

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

What I'm calling a `venturi` `venturi :: Ord a => [a] -> [a]` merges an infinite list of infinite lists into one list, under the assumption that each list, and the heads of the lists, are in increasing order. I wrote this as an experiment in coding data structures in Haskell's lazy evaluation model, rather than as explicit data. The majority of the work done by this code is done by `merge`; the multiples of each prime percolate up through a tournament consisting of a balanced tree of suspended `merge` function calls. In order to create an infinite lazy balanced tree of lists, the datatype `data List a = A a (List a) | B` is used as scaffolding. One thinks of the root of the infinite tree as starting at the leftmost child, and crawling up the left spine as necessary. After some pondering, the `List` data structure for merging is really ingenious! :) Here's a try to explain how it works: The task is to merge a number of sorted and infinite lists when it's known that their heads are in increasing order. In particular, we want to write `primes = (2:) $ diff $ venturi $ map multiple primes`. Thus, we have a (maybe infinite) list `xss =` of infinite lists with the following properties: all sorted `xss` sorted (`map head xss`) where `sorted` is a function that returns `True` if the argument is a sorted list. A first try to implement the merging function is `venturi xss = foldr1 merge xss = xs1 `merge` (xs2 `merge` (xs3 `merge` ...` where `merge` is the standard function to merge to sorted lists. However, there are two problems. The first problem is that this doesn't work for infinite lists since `merge` is strict in both arguments. But the property `head xs1 < head xs2 < head xs3 < ...` we missed to exploit yet can now be used in the following way `venturi xss = foldr1 merge' xss` `merge' (x:xt) ys = x : merge xt ys`. In other words, `merge'` is biased towards the left element `merge' (x:_) _ = x : _` which is correct since we know that `(head xs < head ys)`. The second problem is that we want the calls to `merge` to be arranged as a balanced binary tree since that gives an efficient heap. It's not so difficult to find a good shape for the infinite tree, the real problem is to adapt `merge'` to this situation since it's not associative: `merge' (merge' (x:_) _) = x : merge' (x:_) _`. The problem is that the second form doesn't realize that `y` is also smaller than the third argument. In other words, the second form has to treat more than one element as privileged, namely `x1, x2, ...` and `y`. This can be done with the aforementioned list data structure `data People a = VIP a (People a) | Crowd`. The people (VIPs and crowd) are assumed to be `_sorted_`. Now, we can start to implement `merge' :: Ord a => People a -> People a`. Hi, .. replying to a two-years-old post here, :) and after consulting the full VIP version in `haskellwiki/Prime_Numbers#Implicit_Heap` ... It is indeed the major problem with the merged multiples removing code (similar one to Richard Bird's code from Melissa O'Neill's JFP article) - the linear nature of `foldr`, requiring an `(:: a-b-b)` merge function. To make it freely composable to rearrange the list into arbitrary form tree it must indeed be type uniform `(:: a-a-a)` first, and associative second. The structure of the folded tree should be chosen to better suit the primes multiples production. I guesstimate the total cost as `Sum (1/p)*d`, where `p` is a generating prime at the leaf, and `d` the leaf's depth, i.e. the amount of `merge` nodes its produced multiple must pass on its way to the top. The structure used in your VIP code, `1+(2+(4+(8+...)))`, can actually be improved upon with another, `(2+4)+(4+8)+(8+16)+...`, for which the estimated cost is about 10%-12% lower. This can be expressed concisely as the following: `primes :: () -> [a] -> [a]` `primes () = 2:primes' where primes' = ++ drop 2 `minus` comps` `mults = map (p- fromList) $ primes' (comps,_) = tfold mergeSP (pairwise mergeSP mults) fromList (x:xs)` `= (,xs) tfold f (a: ~(b: ~(c:xs))) = (a `f` (b `f` c)) `f` tfold f (pairwise f xs) pairwise f (x:y:ys) = f x y : pairwise f y` `mergeSP (a,b) ~(c,d) = let (bc,b') = spMerge b c in (a ++ bc, merge b' d) where spMerge u@(x:xs) w@(y:ys) = case compare x y of LT - (x:c,d) where (c,d) = spMerge xs w EQ - (x:c,d) where (c,d) = spMerge xs ys GT - (y:c,d) where (c,d) = spMerge u ys spMerge u = (, u) spMerge w = (, w) with "merge" and "minus" defined in the usual way. Its run times are indeed improved 10%-12% over the VIP code from the haskellwiki page. Testing was done by running the code, interpreted, inside GHCi. The ordered split pairs representing ordered lists here as pairs of a known, finite (so far) prefix and the rest of list, form a _monoid_ under mergeSP. Or with wheel, primes :: () -> [a] -> [a] primes () = 2:3:5:7:primes' where primes' = ++ drop 2 (rollFrom 11) `minus` comps mults = map (p- fromList $ map (p*) $ rollFrom p) $ primes' (comps,_) = tfold mergeSP (pairwise mergeSP mults) fromList (x:xs) = (,xs) rollFrom n = let x = (n-11) `mod` 210 (y,_) = span (< x) wheelSums in roll n $ drop (length y) wheelSums = roll 0 wdiffs roll = scanl (+) wheel = wdiffs ++ wheel wdiffs = 2:4:2:4:6:2:6:4:2:4:6:6:2:6:4:2:6:4:6:8:4:2:4:2: 4:8:6:4:6:2:4:6:2:6:6:4:2:4:6:2:6:4:2:4:2:10:2:10:wdiffs Now _this_, when tested as interpreted code in GHCi, runs about 2.5x times faster than Priority Queue based code from Melissa O'Neill's ZIP package mentioned at the haskellwiki/Prime_Numbers page, with about half used memory reported, in producing 10,000 to 300,000 primes. It is faster than BayerPrimes.hs from the ZIP package too, in the tested range, at about 35 lines of code in total. ~~ This is a repost, with apologies to anyone who sees this twice (I've replied to a two years old thread, and it doesn't show up in GMANE as I thought it would). ~~`

Haskell-Cafe mailing list Haskell-C...@haskell.org

<http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

wheelSums = roll 0 wdiffs roll = scanl (+) wheel = wdiffs ++ wheel wdiffs =
2:4:2:4:6:2:6:4:2:4:6:6:2:6:4:2:6:4:6:8:4:2:4:2: 4:8:6:4:6:2:4:6:2:6:6:4:2:4:6:2:6:4:2:4:2:10:2:10:wdiffs Apparently
that works too, but I meant it to be: wdiffs = 2:4:2:4:6:2:6:4:2:4:6:6:2:6:4:2:6:4:6:8:4:2:4:2:
4:8:6:4:6:2:4:6:2:6:6:4:2:4:6:2:6:4:2:4:2:10:2:10: :)
Haskell-Cafe mailing list Haskell-C...@haskell.org <http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Daniel Fischer - 2010/01/30 19:19

Now `_this_`, when tested as interpreted code in GHCi, runs about 2.5x times faster than Priority Queue based code from Melissa O'Neill's ZIP package mentioned at the haskellwiki/Prime_Numbers page, with about half used memory reported, in producing 10,000 to 300,000 primes. It is faster than `BayerPrimes.hs` from the ZIP package too, in the tested range, at about 35 lines of code in total. That's nice. However, the important criterion is how compiled code (-O2) fares. Do the relations continue to hold? How does it compare to a bitsieve?

Haskell-Cafe mailing list Haskell-C...@haskell.org
<http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

Haskell-Cafe mailing list Haskell-Cafe <at haskell.org
<http://www.haskell.org/mailman/listinfo/haskell-cafe>
Haskell-Cafe mailing list Haskell-C...@haskell.org <http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Eugene Kirpichov - 2010/01/30 19:19

development of the famous classic Turner's sieve, through the postponed filters, through to Euler's sieve, the merging sieve (i.e. Richard Bird's) and on to the tree-fold merging, with `wheel`. I just wanted to see where the simple normal (i.e. `_beginner_`-friendly) functional code can get, in a natural way. It's not about writing the fastest code in `_advanced_` Haskell. It's about having clear and simple code that can be understood at a glance - i.e. contributes to our understanding of a problem - faithfully reflecting its essential elements, and because of `_that_`, fast. It's kind of like `_not_` using mutable arrays in a quicksort. Seeing claims that it's `_either_` Turner's `_or_` the PQ-based code didn't feel right to me somehow, especially the claim that going by primes squares is a pleasing but minor optimization, what with the postponed filters (which serves as the framework for all the other variants) achieving the orders of magnitude speedup and cutting the Turner's $O(n^2)$ right down to $O(n^{1.5})$ just by doing that squares optimization (with the final version hovering around 1.24..1.17 in the tested range). The Euler's sieve being a special case of Eratosthenes's, too, doesn't let credence to claims that only the PQ version is somehow uniquely authentic and faithful to it. Turner's sieve should have been always looked at as just a specification, not a code, anyway, and actually running it is ridiculous. Postponed filters version, is the one to be used as a reference point of the basic `_code_`, precisely because it `_does_` use the primes squares optimization, which `_is_` essential to any basic sieve.

Haskell-Cafe mailing list Haskell-Cafe <at haskell.org
<http://www.haskell.org/mailman/listinfo/haskell-cafe>
Haskell-Cafe mailing list Haskell-C...@haskell.org <http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

If you mean actual performance for a particular task, you should measure the performance in realistic conditions. Namely, if you're implementing a program that needs efficient generation of primes, won't you compile it with -O2? If I realistically needed primes generated in a real life setting, I'd probably had to use some C for that. If OTOH we're talking about a tutorial code that is to be as efficient as possible without losing its clarity, being a reflection of essentials of the problem, then any overly complicated advanced Haskell wouldn't be my choice either. And seeing that this overly-complicated (IMO), steps-jumping PQ-based code was sold to us as the only faithful rendering of the sieve, I wanted to see for myself whether this really holds water.

Haskell-Cafe mailing list Haskell-C...@haskell.org
<http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

Haskell-Cafe mailing list Haskell-Cafe <at haskell.org>
<http://www.haskell.org/mailman/listinfo/haskell-cafe>
Haskell-Cafe mailing list Haskell-C...@haskell.org <http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Daniel Fischer - 2010/01/30 19:19

not a code, anyway, and actually running it is ridiculous. Postponed filters version, is the one to be used as a reference point of the basic `_code_`, precisely because it `_does_` use the primes squares optimization, which `_is_` essential to any basic sieve.

Haskell-Cafe mailing list Haskell-C...@haskell.org
<http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Daniel Fischer - 2010/01/30 19:19

And seeing that this overly-complicated (IMO), steps-jumping PQ-based code was sold to us as the only faithful rendering of the sieve, I wanted to see for myself whether this really holds water. I can understand that very well.

Haskell-Cafe mailing list Haskell-C...@haskell.org
<http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

Turner's sieve should have been always looked at as just a specification, not a code, anyway, and actually running it is ridiculous. Postponed filters version, is the one to be used as a reference point of the basic `_code_`, precisely because it `_does_` use the primes squares optimization, which `_is_` essential to any basic sieve.

Haskell-Cafe mailing list Haskell-Cafe <at haskell.org>
<http://www.haskell.org/mailman/listinfo/haskell-cafe>

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

And seeing that this overly-complicated (IMO), steps-jumping PQ-based code was sold to us as the only faithful rendering of the sieve, I wanted to see for myself whether this really holds water. I can understand that very well. :)
Haskell-Cafe mailing list Haskell-C...@haskell.org
http://www.haskell.org/mailman/listinfo/haskell-cafe

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Daniel Fischer - 2010/01/30 19:19

```
or primes = 2: 3: sieve (tail primes) 5 where sieve fs (p:ps) x = , a!i] ++ sieve ((2*p,q):fs') ps (q+2)
where q = p*p mults = |(s,y)<- fs] fs' = a = accumArray (a b-False) True (x,q-2)
Umm, really? I'd think if you see what that does, you won't have difficulties with a mutable array sieve.
```

Haskell-Cafe mailing list Haskell-C...@haskell.org
http://www.haskell.org/mailman/listinfo/haskell-cafe

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Daniel Fischer - 2010/01/30 19:19

Am Dienstag 29 Dezember 2009 14:58:10 schrieb Will Ness: If I realistically needed primes generated in a real life setting, I'd probably had to use some C for that. If you need the utmost speed, then probably yes. If you can do with a little less, my STUArray bitsieves take about 35% longer than the equivalent C code and are roughly eight times faster than O'NeillPrimes. I can usually live well with that. Wow! That's fast! (that's the code from haskellwiki's primes page, right?) No, it's my own code. Nothing elaborate, just sieving numbers $6ki\frac{1}{2}1$, twice as fast as the haskellwiki code (here) and uses only 1/3 the memory. For the record:

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

Haskell-Cafe mailing list Haskell-Cafe <at haskell.org
http://www.haskell.org/mailman/listinfo/haskell-cafe
Haskell-Cafe mailing list Haskell-C...@haskell.org http://www.haskell.org/mailman/listinfo/haskell-cafe

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

Haskell-Cafe mailing list Haskell-Cafe <at haskell.org
http://www.haskell.org/mailman/listinfo/haskell-cafe

=====
nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Daniel Fischer - 2010/01/30 19:19

especially the claim that going by primes squares is a pleasing but minor optimization, Which it is not. It is a major optimisation. It reduces the algorithmic complexity *and* reduces the constant factors significantly. D'oh! Thinko while computing sum (takeWhile (<= n) primes) without paper. It doesn't change the complexity, and the constant factors are reduced far less than I thought. The huge performance differences I had must have been due to cache misses (unless you use a segmented sieve, starting at the square reduces the number of cache misses significantly).

----- Haskell-Cafe mailing list Haskell-C...@haskell.org
http://www.haskell.org/mailman/listinfo/haskell-cafe

=====
nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

Am Dienstag 29 Dezember 2009 20:16:59 schrieb Daniel Fischer: especially the claim that going by primes squares is a pleasing but minor optimization, Which it is not. It is a major optimisation. It reduces the algorithmic complexity *and* reduces the constant factors significantly. D'oh! Thinko while computing sum (takeWhile (<= n) primes) without paper. It doesn't change the complexity, and the constant factors are reduced far less than I thought. I do not understand. Turner's sieve is primes = sieve where sieve (p:xs) = p : sieve and the Postponed Filters is primes = 2: 3: sieve (tail primes) where sieve (p:ps) xs = h ++ sieve ps where (h,~(_:t)) = span (< p*p) > Are you saying they both exhibit same complexity? I was under impression that the first shows $O(n^2)$ approx., and the second one $O(n^{1.5})$ (for n primes produced). ----- Haskell-Cafe mailing list Haskell-C...@haskell.org http://www.haskell.org/mailman/listinfo/haskell-cafe

=====
nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Daniel Fischer - 2010/01/30 19:19

primes produced). In Turner/postponed filters, things are really different. Actually, Ms. O'Neill is right, it is a different algorithm. In the above, we match each prime only with its multiples (plus the next composite in the PQ version). In Turner's algorithm, we match each prime with all smaller primes (and each composite with all primes <= its smallest prime factor, but that's less work than the primes). That gives indeed a complexity of $O(\pi(\text{bound})^2)$. In the postponed filters, we match each prime p with all primes <= sqrt p (again, the composites contribute less). I think that gives a complexity of $O(\pi(\text{bound})^{1.5} \log(\log \text{bound}))$ or $O(n^{1.5} \log(\log n))$ for n primes produced.

----- Haskell-Cafe mailing list Haskell-C...@haskell.org
http://www.haskell.org/mailman/listinfo/haskell-cafe

=====
nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Steve - 2010/01/30 19:19

twice as fast as the haskellwiki code (here) and uses only 1/3 the memory. For the record:

nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

Haskell-Cafe mailing list Haskell-Cafe <at haskell.org
<http://www.haskell.org/mailman/listinfo/haskell-cafe>
Haskell-Cafe mailing list Haskell-C...@haskell.org <http://www.haskell.org/mailman/listinfo/haskell-cafe>

=====
nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Daniel Fischer - 2010/01/30 19:19

Her article even adds the primes into the PQ prematurely itself, as soon as the prime is discovered (she fixes that in her ZIP package). With the PQ keeping these prematurely added elements deep inside its guts, the problem might not even manifest itself immediately, but the memory blow-up would eventually hit the wall (having the PQ contain all the preceding primes, instead of just those below the square root of a limit - all the entries above the square root not contributing to the calculation at all). And what we're after here is the insight anyway. Skipping steps of natural development does not contribute to garnering an insight. Most prominent problem with Turner's `/code/_specification_`, is the premature start ups, and divisibility testing of primes (as Melissa O'Neill's analysis shows). Fix one, and you get Postponed Filters (which should really be used as a basis reference point for all the rest). Fix the other - and you've got the Euler's sieve: `primes = 2: 3: sieve (tail primes) where sieve (p:ps) xs = h ++ sieve ps]` where `h,~(_:t) = span (< p*p) xs` Clear, succinct, and plenty efficient for an introductory textbook functional lazy code, of the same order of magnitude performance as PQ code on odds only. Now comparing the PQ performance against `_that_` would only be fare, and IMO would only add to its value - it is faster, has better asymptotics, and is greatly amenable to the wheel optimization right away. prime only with its multiples (plus the next composite in the PQ version). In Turner's algorithm, we match each prime with all smaller primes (and each composite with all primes <= its smallest prime factor, but that's less work than the primes). That gives indeed a complexity of $O(\pi(\text{bound})^2)$. In the postponed filters, we match each prime p with all primes $\leq \sqrt{p}$ (again, the composites contribute less). I think that gives a complexity of $O(\pi(\text{bound})^{1.5} \log(\log \text{bound}))$ or $O(n^{1.5} \log(\log n))$ for n primes produced. Empirically, it was always below 1.5, and above 1.4, and PQ/merged multiples removal around 1.25..1.17. I should really stop doing those calculations in my head, it seems I'm getting too old for that. Doing it properly, on paper, the cost for identifying p_k is

... read more »

Haskell-Cafe mailing list Haskell-C...@haskell.org
<http://www.haskell.org/mailman/listinfo/haskell-cafe>

=====
nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

(again, the composites contribute less). I think that gives a complexity of $O(\pi(\text{bound})^{1.5} \log(\log \text{bound}))$ or $O(n^{1.5} \log(\log n))$ for n primes produced. Empirically, it was always below 1.5, and above 1.4, and PQ/merged multiples removal around 1.25..1.17. I should really stop doing those calculations in my head, it seems I'm getting too old for that. Doing it properly, on paper, the cost for identifying p_k is $\pi(\sqrt{p_k})$

... read more »

=====
nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

.. and that is of course a matter of personal preference. A genius is perfectly capable of making big leaps across

=====
nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Daniel Fischer - 2010/01/30 19:19

Using a feeder, i.e. `primes = 2:3.sieve feederprimes`, where `feederprimes` are generated as above, to reduce memory pressure and GC impact, that fits pretty well with my measurements of the time to find `p_k` for several `k` between 500,000 and 20,000,000. The question of a memory blowup with the treefolding merge still remains. For some reason using its second copy for a feeder doesn't reduce the memory (as reported by standalone compiled program, GHCi reported values are useless) - it causes it to increase twice..... I have a partial solution. The big problem is that the feeder holds on to the beginning of `comps` while the main runner holds on to a later part. Thus the entire segment between these two points must be in memory. So have two lists of composites (yeah, you know that, but it didn't work so far). But you have to force the compiler not to share them: `enter -fno-cse`. The attached code does that (I've also expanded the wheel), it reduces the memory requirements much (a small part is due to the larger ... read more »

V13Primes.hs
3K
Download

=====
nt factor FASTER primes (was: Re: Code and Perf. Data for Prime Finders (was: Genuine Eratosthenes sieve))

Posted by Will Ness - 2010/01/30 19:19

Daniel Fischer <daniel.is.fischer@web.de> writes: But there's a lot of list construction and deconstruction necessary for the Euler sieve. yes. Unless, of course, a smart compiler recognizes there's only one consumer for the values each multiples-list is producing, and keeps it stripped down to a generator function, and its current value. I keep thinking a smart compiler could eliminate all these `span` calls and replace them with just some pointers manipulating... Of course I'm no smart compiler, but I don't see how it could be even possible to replace the `span` calls with pointer manipulation when dealing with lazily generated (infinite, if we're really mean) lists. Even when you're dealing only with strict finite lists, it's not trivial to do efficiently. I keep thinking that storage duplication with `span`, `filter` etc. is not really necessary, and can be replaced with some pointer chasing - especially when there's only one consumer present for the generated values. What I mean is thinking of lists in terms of produce/consumer paradigm, as objects supporting the `{ pull, peek }` interface, keeping the generator inside that would produce the next value on 'pull' request and keep it ready for any 'peek's. Euler's sieve is `sieve (p:ps) xs = h ++ sieve ps (t `minus` map (p*))` where `(h,t) = span (< p*p) xs` Everything lives only through access, so `(sieve (tail primes))` would create an object with the generator which has the 'span' logic inlined: `sieve ps xs = make producer such that p := pull ps`

=====