15-150 Fall 2019
Thursday, 21 November

Stephen Brookes
lazy stuff

To show the power of laziness we tackle some problems that manipulate infinite data

• generating factorials lazily

• prime number sieves

• the Hamming sequence
• **datatype** 'a lazylist = Cons of 'a * (unit -> 'a lazylist)

• **val** nats : int -> int lazylist

• **val** lazymap : ('a -> 'b) -> ('a lazylist -> 'b lazylist)

• **val** lazyzip : 'a lazylist * 'b lazylist -> ('a * 'b) lazylist

• **val** lazyfilter : ('a -> bool) -> 'a lazylist -> 'a lazylist

• **val** show : int -> 'a lazylist -> 'a list

as defined earlier
• Expressions $e_1$, $e_2 : t$ lazylist are equal if and only if

\[ \forall \ n \geq 0. \ \text{show} \ n \ e_1 = \text{show} \ n \ e_2 \]

• $\text{Cons}(x_1, h_1) = \text{Cons}(x_2, h_2)$ iff $x_1 = x_2$ and $h_1() = h_2()$

For all $m \geq 0$, $\text{nats} \ (m+1) = \text{lazymap} \ (\text{fn} \ x \ => \ x+1) \ (\text{nats} \ m)$.

$\text{show} \ n \ (\text{nats} \ (m+1)) = [m+1, \ ..., \ m+n]$ \n
$\text{show} \ n \ (\text{lazymap} \ (\text{fn} \ x \ => \ x+1) \ (\text{nats} \ m))$

$= \text{map} \ (\text{fn} \ x \ => \ x+1) \ (\text{show} \ n \ (\text{nats} \ m))$

$= \text{map} \ (\text{fn} \ x \ => \ x+1) \ [m, \ ..., \ m+n-1]$

$= [m+1, \ ..., \ m+n]$
recursive laziness

lazy lists can be generated recursively

facts : unit -> int lazylist

fun facts() = 
  Cons(1, fn () =>
    lazymap (op *) (lazyzip (facts(), nats 2)))

facts() represents 1, 2, 6, 24, 120, ...

For all $n \geq 1$, show $n \cdot \text{facts}() = [1!, \ldots, n!]$
prime numbers

• A prime is an integer $\geq 2$ divisible only by 1 and itself.

• There are *infinitely many* prime numbers.

  2, 3, 5, 7, 11, ...

• We can write an ML program that builds a lazy list of all the primes.

  Cons(2, fn () => ...???...)

  How?
  
  We need an inductive way to generate primes...
The **Sieve of Eratosthenes** is a simple, ancient algorithm for finding all prime numbers up to any given limit.

It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2.

The Wikipedia entry describes an algorithm for producing a list of the first N primes.

This algorithm also suggests a way to produce a lazy list of all the primes...
A prime sieve

sift : int -> int lazylist -> int lazylist
fun sift x = lazyfilter (fn y => y mod x <> 0)

sieve : int lazylist -> int lazylist
fun sieve (Cons(x, h)) =
  Cons(x, fn () => sieve (sift x (h( ))))

primes : int lazylist
val primes = sieve(nats 2)
sift and sieve

• Suppose $\text{Cons}(x, h)$ represents the integers $\geq 2$ not divisible by the first $n$ primes

• Then $x$ is the $n+1$th prime

• And $\text{sift} \ x \ (h( ))$ represents the integers $\geq 2$ not divisible by the first $n+1$ primes

• So $\text{sieve(nats 2)}$ represents the primes

For all $n \geq 1$, show $n \cdot (\text{sieve(nats 2)}) = [p_1, \ldots, p_n]$

where $p_1, \ldots, p_n$ are the first $n$ primes
results

- show 1000 primes;
val it = [2,3,5,7,11,13,17,19,23,29,31,37,...] : int list

- List.rev it;
val it = [7919,7907,7901,7883,7879,7877,7873,7867,...] : int list

- show 10000 primes;
val it = [2,3,5,7,11,13,17,19,23,29,31,37,...] : int list

- List.rev it;
val it = [104729,104723,104717,104711,104707,104701,...] : int list
fun is_prime n = 
  foldr (fn (x, t) => (n mod x <> 0) andalso t) 
    true 
    (upto 2 (n div 2));

val primes = lazyfilter is_prime (nats 2)

show 1000 primes = ???

show 5000 primes = ???
• Our sieve program is written in **Standard ML**

  Standard ML of New Jersey v110.81 [built: Tue Aug 8 15:38:04 2019]

• Can also be written (more succinctly) in **Haskell**

```haskell
primes = filterPrime [2..]
  where filterPrime (p:xs) =
        p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

“This example is contrived to illustrate Haskell syntax, including list comprehensions, ... Do not take it seriously as a prime generator.”
historical remark

• Our prime sieve function is, arguably, only loosely based on Eratosthenes

Standard ML of Ancient Greece v110.81 [built: Tue Aug  8 15:38:04 BC 256]

• There’s room for improvement…

The Genuine Sieve of Eratosthenes
Melissa O’Neill,
Harvey Mudd University

A much beloved and widely used example showing the elegance and simplicity of lazy functional programming represents itself as “The Sieve of Eratosthenes”. This paper shows that this example is not the sieve, and presents an implementation that actually is.

JFP 2009

Haskell
The **Sieve of Eratosthenes** is a simple, ancient algorithm for finding all **prime numbers** up to any given limit.

It does so by iteratively marking as **composite** (i.e., not prime) the multiples of each prime, starting with the first prime number, 2.

The multiples of a given prime are generated as a sequence of numbers starting from that prime, with **constant difference between them** equal to that prime.

This (additive operation) is the sieve's key distinction from another algorithm known as **trial division** that sequentially tests each candidate number for divisibility by each prime.

---

**our sieve function actually does this!**
sifting ideas

• Maintain a *priority queue* that represents the *composite* numbers built from the primes found so far
  
  • smallest composite will be easy to find(!)

• If current candidate is less than smallest composite, it must be the next prime (Insert its multiples into the priority queue)
  
  * For prime $p$, we only need to include composites starting from $p^2$. Smaller composites $2p, 3p, \ldots$ have a smaller prime factor, so get handled earlier.

• Otherwise it’s a composite (Adjust the priority queue)
  
  * No need to keep composites smaller than the current candidate.
priority queue

type PQ = (int * int lazylist) list;

(* Invariant: a sorted list of (k, L) pairs, ordered by k *)

exception EmptyPQ;

val empty : PQ = [ ]

(* minKey : PQ -> int *)
fun minKey [ ] = raise EmptyPQ
| minKey ((k, L) :: kvs) = k
d

(* minKeyValue : PQ -> int * int lazylist *)
fun minKeyValue [ ] = raise EmptyPQ
| minKeyValue ((k, L) :: kvs) = (k, L)

A list of (int * int lazylist) pairs, ordered by int component
For each prime p so far, there’s a pair (m, L) representing the multiples of p, starting from m
insertion

(* insert : int * int lazylist -> PQ -> PQ *)
fun insert (k, L) [ ] = [(k, L)]
|     insert (k, L) ((k', L') :: kvs) =
     if k < k' then ((k, L) :: (k', L') :: kvs) else ((k', L') :: (insert (k, L) kvs))

preserves invariant

deletion

(* deleteMinAndInsert : int * int lazylist -> PQ -> PQ *)
fun deleteMinAndInsert (k, L) [ ] = [(k, L)]
| deleteMinAndInsert (k, L) ((k', L') :: kvs) = insert (k, L) kvs

preserves invariant
adjust

(* adjust : int -> PQ -> PQ *)

fun adjust y pq =
  let
    val (n, Cons (n', ns)) = minKeyValue pq
  in
    if n <= y
    then adjust y (deleteMinAndInsert (n', ns ( )) pq)
    else pq
  end

removes composites smaller than y

preserves invariant
**insertprime**

(* insertprime : int * int lazylist -> PQ -> PQ *)

```haskell
fun insertprime (p, xs) = insert (p * p, lazymap (fn x => x * p) xs)
```

when xs is nats (p+1),
this will insert composites of p
starting from $p^2$
(*/ siever : int lazylist -> PQ -> int lazylist *)*/

fun siever (Cons (y, k)) pq =
    let
        val ys = k ( )
    in
        if minKey pq <= y
            then siever ys (adjust y pq)
            else Cons (y, fn ( ) => siever ys (insertprime (y , ys) pq))
    end
genuine sieve

(* g_sieve : int lazylist -> int lazylist *)

fun g_sieve (Cons (x, h)) =
  let
    val xs = h ()
  in
    Cons (x, fn ( ) => siever xs (insertprime (x, xs) empty))
  end;

(* g_primes : int lazylist *)

val g_primes = g_sieve (nats 2)
results

- show 1000 g_primes;
  val it = [2,3,5,7,11,13,17,19,23,29,31,37,...]
    : int list

- List.rev it;
  val it = [7919,7907,7901,7883,7879,7877,7873,7867,...]
    : int list

(as before)
results

- show 10000 g_primes;

uncaught exception Overflow [overflow] raised at: <file stdIn>

OOPS!
What went wrong?
- open IntInf;

- show 10000 g_primes;

val it = [2,3,5,7,11,13,17,19,23,29,31,37,...] : int list

- List.rev it;

val it = [104729,104723,104717,104711,104707,104701,...] : int list
results

- show 60000 g_primes;
val it = [2,3,5,7,11,13,17,19,23,29,31,37,...]
  : int list

- List.rev it;

- val it = [746773,746749,746747,746743,746737,746723,...]
  : int list

Much faster than our earlier version

(maybe Melissa O’Neill has a point!)
what happens

g_sieve (nats 2)
  = Cons (2, fn () => siever (nats 3) (insertprime (2, nats 3) empty))
  = Cons (2, fn () => siever (nats 3) [ (4, [6…]) ])

siever (nats 3) [ (4, [6…]) ]
  = Cons(3, fn () => siever (nats 4) [ (4, [6…]), (9, [12…]) ])

siever (nats 4) [ (4, [6…]), (9, [12…]) ]
  = siever (nats 5) [ (6, [8…]), (9, [12…]) ]
  = Cons(5, fn () => siever (nats 6) [ (6, [8…]), (9, [12…]), (25, [30…]) ])

siever (nats 6) [ (6, [8…]), (9, [12…]), (25, [30…]) ]
  = siever (nats 7) [ (8, [10…]), (9, [12…]), (25, [30…]) ]
what happens

• Each time we call `siever (nats n) pq`
  we’ve already “produced” the primes less than n
  and pq represents the multiples of those primes

• n is prime if it is less than the smallest integer in pq

• Otherwise n must be composite (non-prime)
exercise

Implement this prime sieve without representing the composites as lazy lists

• Instead of a pair \((m, L) : \text{int} \times \text{int lazylist}\) where \(L\) represents the successive multiples of \(p\) after \(m\), just use the pair \((p, m) : \text{int} \times \text{int}\)

• Maintain the list of \((p, m)\) pairs in increasing order of \(m\)

• To adjust, instead of replacing \((m, \text{Cons}(m', h))\) by \((m', \text{h( ))}\) replace \((p, m)\) by \((p, m+p)\)
Hamming numbers

… are numbers whose only prime factors are 2, 3 and 5. They are named after [Richard Hamming] but became famous (or notorious) after [Edsger Dijkstra] posed the question of how to efficiently enumerate them in numeric order.

The problem is frequently trotted out to explain why "Functional Programming is better".

Richard Hamming

Edsger Dijkstra
solution

• Each Hamming number has the form $2^i3^j5^k$ ($i, j, k \geq 0$)

• The smallest is $1 (= 2^03^05^0)$

• We can *generate* the Hamming numbers recursively, using lazy merge and lazy map

  ```
  merge : int lazylist * int lazylist -> int lazylist

  lazymap : (int -> int) -> int lazylist -> int lazylist
  ```
fun merge (Cons(x, h), Cons(y, k)) =
  case Int.compare(x, y) of
    LESS    => Cons(x, fn () => merge (h(), Cons(y, k)))
  | EQUAL   => Cons(x, fn () => merge(h(), k( )))
  | GREATER => Cons(y, fn () => merge (Cons(x, h), k( )))

merge : int lazylist * int lazylist -> int lazylist

REQUIRES  L represents an increasing list of x’s,
          R represents an increasing list of y’s

ENSURES  merge(L, R) represents the increasing list of x’s and y’s
fun times k = lazymap (fn x => k*x)

fun ham() = Cons(1, fn () =>
    merge (times 2 (ham ()),
    merge (times 3 (ham ()),
    times 5 (ham ()))))

- show 500 (ham());
val it = [1,2,3,4,5,6,8,9,10,12,15,16,...] : int list

- List.rev it;
val it =
[937500,933120,921600,911250,900000,885735,884736,874800,864000,843750,839808,829440,...] : int list