15-150

Principles of Functional Programming

Slides for Lecture 20

Time to be Lazy

April 4, 2024

Michael Erdmann

Lessons:

- Infinite data structures
- Encapsulated Computation

  (We have seen a form of that with failure continuations. Closures are the key tool.)
- Demand-driven (lazy) computation

Examples of (potentially) infinite data:

- Even integers, natural numbers, primes
- All keystrokes you will make on a keyboard.
- Video/Audio streams

Streams

Caution:

We will build our own streams.

These are different from SML's

built-in I/O streams.

Brain Teaser

What is the difference between

`f` and `(fn x => f x)` ?

Brain Teaser

What is the difference between

`f` and **`(fn x => f x)`** ?

`f` is not evaluated in **`(fn x => f x)`** until the lambda expression is called on an argument and the body **`f x`** is evaluated.

If **`f`** is itself an expression, it *may not* be valuable, whereas **`(fn x => f x)`** is valuable.

Brain Teaser (cont)

What is the difference between

`f` and **`(fn x => f x)`** ?

For instance, consider:

```
fun g x = g x
```

Now suppose **`f`** is the expression **`(g 3)`**.

$$\underbrace{\texttt{(g 3)}}_{\text{loops}} \qquad \underbrace{\texttt{(fn x => (g 3) x)}}_{\text{is a value (a closure)}}$$

Brain Teaser (cont)

What is the difference between

`f` and **`(fn x => f x)`** ?

For instance, consider:

`fun g x = g x` `g : 'a -> 'b`

Now suppose **`f`** is the expression **`(g 3)`**.

$\underbrace{\texttt{(g 3)}}_{\text{loops}}$ $\underbrace{\texttt{(fn x => (g 3) x)}}_{\text{is a value (a closure)}}$ `:'a->'b`

here `g : int -> 'a` `(g 3) : 'a -> 'b`

Suspensions

Definition

A *suspension* of type **τ** is a function of type

unit -> τ .

We say a suspension is *forced* when it is applied to argument **()** .

If **e:t**, then **(fn () => e)** is a suspension of type **t**.

We view such a suspension as a *lazy* represention of **e**. The suspension is a function, so **e** will not be evaluated until the suspension is forced.

Streams

We will model (potentially infinite) streams of data much like lists, but lazily:

- Base case: an empty stream
- Inductive case:

A suspension of the following:

A single element

"consed"

onto another stream.

(Suspensions will allow us to build streams with no base cases!)
(never-ending, infinite, yet encapsulated finitely)

Streams

We will model (potentially infinite) streams of data much like lists, but lazily:

$\boxed{\text{Co-Induction}}$

A suspension of the following:

A single element

"consed"

onto another stream.

(Suspensions will allow us to build streams with no base cases!)
(never-ending, infinite, yet encapsulated finitely)

STREAM Signature

First, a signature to describe streams abstractly.

(Then we will implement them.)

```
signature STREAM =
sig
  type 'a stream  (* abstract *)
```

Streams are abstract, modeled by the type `'a stream`.

```
end
```

```
signature STREAM =
sig
   type 'a stream  (* abstract *)
   datatype 'a front = Empty | Cons of 'a * 'a stream
                                 (* NOTE: concrete *)
```

Streams are abstract and lazy.
We may want to look at their elements.
The type `'a front` represents the result of performing just enough computation to expose the first element of a stream, as well as obtain the rest of the stream.

We say that a `front` is a "view" of a `stream`.

`Cons` models the (co)inductive nature of streams.

`Empty` models the result of exposing an empty stream.

```
end
```

```
signature STREAM =
sig
  type 'a stream  (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  val expose : 'a stream -> 'a front
```

This function performs the computations needed to see the first element of a **`stream`**.
The function returns the corresponding **`front`** (which could be **`Empty`**).

Caution: Since exposing a stream value involves computation, the computation might not terminate.
This is different from looking at list values.

```
end
```

```
signature STREAM =
sig
  type 'a stream  (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  val expose : 'a stream -> 'a front
  val delay : (unit -> 'a front) -> 'a stream
```

This function expects a suspension of a **front** and creates the corresponding **stream** for it.

Notice that the function expects a suspension of a front, not merely the front. Why?

The reason is contained in the earlier brain teaser: SML evaluates arguments *eagerly*.

If we had `delay : 'a front -> 'a stream`, then `delay(e)` would evaluate `e`, but we want the computation represented by `e` to be lazy, so need `delay(fn () => e)`.

```
signature STREAM =
sig
  type 'a stream  (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  val expose : 'a stream -> 'a front
  val delay : (unit -> 'a front) -> 'a stream

  val empty : 'a stream
```

This is one representation of a stream containing no elements.

In particular, we will ensure that `expose(empty)` $\Longrightarrow$ `Empty`.

Caution: One can imagine other “empty stream”s, for instance a stream value `s` such that `expose(s)` loops forever.

```
end
```

```
signature STREAM =
sig
  type 'a stream  (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  val expose : 'a stream -> 'a front
  val delay : (unit -> 'a front) -> 'a stream

  val empty : 'a stream
  val cons : 'a * 'a stream -> 'a stream
```

A function useful for constructing streams eagerly, i.e., e.g., when elements are already known.

```
end
```

```
signature STREAM =
sig
  type 'a stream  (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  val expose : 'a stream -> 'a front
  val delay : (unit -> 'a front) -> 'a stream

  val empty : 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val null : 'a stream -> bool
```

A function to test whether a stream is empty.

Caution: Under the hood, this may involve stream exposures, so might not terminate.

```
end
```

```
signature STREAM =
sig
  type 'a stream  (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  val expose : 'a stream -> 'a front
  val delay : (unit -> 'a front) -> 'a stream

  val empty : 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val null : 'a stream -> bool
  val take : ('a stream * int ) -> 'a list
```

`take(s,n)` returns the first `n` elements of stream `s`, as a list; raises `Subscript`, if any exposure encounters `Empty`.

Caution: As always, stream exposures can loop forever.

```
end
```

```
signature STREAM =
sig
  type 'a stream  (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  val expose : 'a stream -> 'a front
  val delay : (unit -> 'a front) -> 'a stream

  val empty : 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val null : 'a stream -> bool
  val take : ('a stream * int ) -> 'a list

  val map : ('a -> 'b) -> 'a stream -> 'b stream
```

This is lazy!!! So `map f s` returns a stream `s'`, but does not apply `f` to any element of `s`, not until someone exposes `s'`.

```
end
```

```
signature STREAM =
sig
  type 'a stream  (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  val expose : 'a stream -> 'a front
  val delay : (unit -> 'a front) -> 'a stream

  val empty : 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val null : 'a stream -> bool
  val take : ('a stream * int ) -> 'a list

  val map : ('a -> 'b) -> 'a stream -> 'b stream
  val filter: ('a -> bool) -> 'a stream -> 'a stream

end
```

Again: This is lazy!!!

```
signature STREAM =
sig
   type 'a stream  (* abstract *)
   datatype 'a front = Empty | Cons of 'a * 'a stream

   val expose : 'a stream -> 'a front
   val delay : (unit -> 'a front) -> 'a stream

   val empty : 'a stream
   val cons : 'a * 'a stream -> 'a stream
   val null : 'a stream -> bool
   val take : ('a stream * int ) -> 'a list

   val map : ('a -> 'b) -> 'a stream -> 'b stream
   val filter: ('a -> bool) -> 'a stream -> 'a stream
  (* ... more functions: append, tabulate, zip ... *)
end
```

Stream Structure

Time to implement streams.

Here, we will implement some key functions,
look at some examples,
implement a couple higher-order functions,
then build a stream containing all the primes.

```
structure Stream : STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
```

We define the representation of an `'a stream` to be (a `Stream` constructor wrapped around) the suspension of an `'a front`.

We do not really need to define a datatype with a `Stream` constructor, but doing so helps when debugging. After all, a suspension is simply a function. By wrapping `Stream` around suspensions, we can more readily see what the function means to encode (we could also look at the function type).

There is one additional subtlety: `'a stream` refers to `'a front`. Let's see how to deal with that.

```
end
```

```
structure Stream : STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
  and 'a front = Empty | Cons of 'a * 'a stream
```

The **and** keyword allows us to define mutually recursive datatypes.

(recall that we can also define mutually recursive functions with **and**).

As a reminder: the datatype declaration for `'a front` was specified concretely in signature **STREAM**, so we need to implement it as it was specified.

```
end
```

```
structure Stream : STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
  and 'a front = Empty | Cons of 'a * 'a stream
```

A type picture to describe what is happening:

$$\underbrace{\texttt{Stream}\ \underbrace{\texttt{(fn () =>}\ \underbrace{\texttt{Cons(}\underbrace{\texttt{x}}_{\texttt{t}}\texttt{, }\underbrace{\texttt{s}}_{\texttt{t stream}}\texttt{))}}_{\texttt{t front}}}_{\texttt{t front}\ \text{suspension}}}_{\texttt{t stream}}$$

(Here `t` is the stream's element type.)

```
end
```

```
structure Stream : STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
  and 'a front = Empty | Cons of 'a * 'a stream

  fun delay (d) = Stream(d)
```

> `delay : (unit -> 'a front) -> 'a stream`
>
> `delay` simply wraps a `Stream` constructor around a suspension of a front.

```
end
```

```
structure Stream : STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
  and 'a front = Empty | Cons of 'a * 'a stream

  fun delay (d) = Stream(d)
  fun expose (Stream(d)) = d ()
```

`expose : 'a stream -> 'a front`

`expose(s)` forces the underlying suspension in `s`.

```
end
```

```
structure Stream : STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
  and 'a front = Empty | Cons of 'a * 'a stream

  fun delay (d) = Stream(d)
  fun expose (Stream(d)) = d ()

  val empty = Stream (fn () => Empty)
```

`empty` is the suspension of `Empty` (which is a `front`), with the `Stream` constructor turning that into a `stream`.

```
end
```

```
structure Stream : STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
  and 'a front = Empty | Cons of 'a * 'a stream

  fun delay (d) = Stream(d)
  fun expose (Stream(d)) = d ()

  val empty = Stream (fn () => Empty)
  fun cons (x, s) = Stream (fn () => Cons(x, s))
```

Given a known element **x** and a stream **s**, **cons(x,s)** creates a new stream, consisting of **x** followed by all the elements of **s**.

```
end
```

```
structure Stream : STREAM =
struct
  datatype 'a stream = Stream of unit -> 'a front
  and 'a front = Empty | Cons of 'a * 'a stream

  fun delay (d) = Stream(d)
  fun expose (Stream(d)) = d ()

  val empty = Stream (fn () => Empty)
  fun cons (x, s) = Stream (fn () => Cons(x, s))

  fun map ...    (* we will implement this soon *)

  fun filter ... (* we will implement this soon *)

  ...  (* other functions *) ...
end
```

Example #1

For all these examples, assume (a) we are writing code outside the **`Stream`** structure and (b) **`structure S = Stream`**.

Here is how we might implement an infinite stream, all of whose elements are **`1`**:

```
fun ones'() = S.Cons(1, S.delay ones')
val ones = S.delay ones'
```

Observe:
```
ones'  : unit -> int S.front
ones   : int S.stream
```

Example #2

Here is how we might implement an infinite stream consisting of all the natural numbers:

```
fun nat' x () =
        S.Cons (x, S.delay (nat'(x+1)))

val nats = S.delay (nat' 0)
```

Observe:

```
   nat' : int -> unit -> int S.front

   nats : int S.stream
```

(example #2 continued)

```
fun nat' x () = S.Cons (x, S.delay (nat'(x+1)))
val nats = S.delay (nat' 0)
```

Consider now:

```
val S.Cons(x, rest) = S.expose nats
val S.Cons(y, _) = S.expose rest
```

To what values are **`x`** and **`y`** bound?

What does **`rest`** represent?

(example #2 continued)

```
fun nat' x () = S.Cons (x, S.delay (nat'(x+1)))
val nats = S.delay (nat' 0)
```

Consider now:

```
val S.Cons(x, rest) = S.expose nats
val S.Cons(y, _) = S.expose rest
```

To what values are **x** and **y** bound?

What does **rest** represent?

Answers: **0/x** **1/y**

rest is a stream consisting of all natural numbers greater than **0**.

(example #2 continued)

```
fun nat' x () = S.Cons (x, S.delay (nat'(x+1)))
val nats = S.delay (nat' 0)
```

Consider now:

```
val S.Cons(x, rest) = S.expose nats
val S.Cons(y, _) = S.expose rest

val S.Cons(z, _) = S.expose nats
```

To what value is **z** bound?

(example #2 continued)

```
fun nat' x () = S.Cons (x, S.delay (nat'(x+1)))
val nats = S.delay (nat' 0)
```

Consider now:

```
val S.Cons(x, rest) = S.expose nats
val S.Cons(y, _) = S.expose rest

val S.Cons(z, _) = S.expose nats
```

To what value is **z** bound?

Answer: **0/z** (Same as **x**.)

Memoization

- Suppose exposing a stream element takes 1 month to compute.
- Each time we expose that same stream element, we will force the same suspension and therefore require 1 month of computation time.
- Some lazy languages, like Haskell, remember the value computed, so that it does not need to be re-computed on subsequent re-exposures, merely looked up. That is called *memoization*.
- We will see one way to do that Tuesday.

Stream Equivalence

- Recall the function

```
take : ('a stream * int ) -> 'a list
```

- We say that two streams **X** and **Y** produced by the same structure **Stream : STREAM** are extensionally equivalent ($X \cong Y$) if and only if

$$\texttt{Stream.take(X,n)} \cong \texttt{Stream.take(Y,n)}$$

for all integers $n \geq 0$.

Productive Streams

- We say a stream **`s`** of type **`t Stream.stream`** is *productive* if and only if **`Stream.expose(s)`** returns one of the following:
  a) **`Stream.Empty`** or
  b) **`Stream.Cons(x,s')`**, with **`x`** a value of type **`t`** and **`s'`** itself a productive stream.

Productive & Infinite Streams

- We say a stream **`s`** of type **`t Stream.stream`** is *productive* if and only if **`Stream.expose(s)`** returns one of the following:
    a) **`Stream.Empty`** or
    b) **`Stream.Cons(x,s')`**, with **`x`** a value of type **`t`** and **`s'`** itself a productive stream.

- Now we can formally define the intuitive notion of an infinite stream: A stream is *infinite* if it is productive and if successive exposures never encounter **`Stream.Empty`**.

Some more Stream functions

Let us implement a few more functions inside the structure **`Stream`**.

(Since we are within the structure we do *not* use the qualified names "**`Stream.`**" or "**`S.`**")

`null`

`'a stream -> bool`

```
fun null s = (case (expose s) of
                    Empty => true
                  | _     => false)
```

Observe that **`null`** must expose stream **`s`**, in order to try to determine whether the stream is **`empty`**, by checking whether the corresponding front is **`Empty`**. This exposure could take a long time, possibly even never terminating, depending on what **`s`** is.

map

('a -> 'b) -> 'a stream -> 'b stream

```
fun map f s = delay(fn () => map' f (expose s))

and map' f Empty = Empty
  | map' f (Cons(x,s')) = Cons(f x, map f s')
```

map

```
('a -> 'b) -> 'a stream -> 'b stream
```

```
fun map f s = delay(fn () => map' f (expose s))
and map'  f Empty = Empty
  | map'  f (Cons(x,s')) = Cons(f x, map f s')
```

I find it convenient mentally to have two
mutually recursive functions when working with streams.
One function focuses on streams, the other on fronts.
Other implementations are possible (see next slide).

map

```
('a -> 'b) -> 'a stream -> 'b stream
```

```
fun map f s = delay(fn () => map' f (expose s))
and map' f Empty = Empty
  | map' f (Cons(x,s')) = Cons(f x, map f s')
```

Alternate implementation:

```
fun map f s =
      delay (fn () =>
        (case (expose s) of
            Empty => Empty
          | Cons(x,s') => Cons (f x, map f s')))
```

map

`('a -> 'b) -> 'a stream -> 'b stream`

```
fun map f s = delay(fn () => map' f (expose s))
and map' f Empty = Empty
  | map' f (Cons(x                s')
```

Notice the laziness here!

`map` does not actually call **`f`** on any elements of **`s`**. Instead, **`map`** delays such work. When/if someone exposes the mapped stream, **`f`** will be applied to the first element of **`s`** by **`map'`**.

map

`('a -> 'b) -> 'a stream -> 'b stream`

```
fun map f s = delay(fn () => map' f (expose s))

and map' f Empty = Empty
  | map' f (Cons(x,s')) = Cons(f x, map f s')
```

`map` does not actually call **`f`** on any elements of **`s`**. Instead, **`map`** delays such work. When/if someone exposes the mapped stream, **`f`** will be applied to the first element of **`s`** by **`map'`**.

Then **`map'`** applies **`map f`** to the rest of the stream **`s'`**, so as to delay further calls until someone needs the elements.

`filter`

`('a -> bool) -> 'a stream -> 'a stream`

Similar to **`map`**, much like **`List.filter`** and **`List.map`** are similar.

There is one subtlety.

filter

```
('a -> bool) -> 'a stream -> 'a stream
```

```
fun filter p s =
      delay (fn () => filter' p (expose s))

and filter' p Empty = Empty
   | filter' p (Cons(x,s')) =
        if (p x) then Cons (x, filter p s')
        else filter' p (expose s')
```

If someone exposes a filtered stream,
code must look for the first element satisfying `p`.
That may entail looking at multiple elements of `s`,
so may need to call `filter'` repeatedly.

filter

```
('a -> bool) -> 'a stream -> 'a stream
```

```
fun filter p s =
     delay (fn () => filter' p (expose s))

and filter' p Empty = Empty
  | filter' p (Cons(x,s')) =
      if (p x) then Cons (x, filter p s')
      else filter' p (expose s')
```

If someone exposes a filtered stream,
code must look for the first element satisfying **`p`**.
That may entail looking at multiple elements of **`s`**,
so may need to call **`filter'`** repeatedly.

filter

```
('a -> bool) -> 'a stream -> 'a stream
```

```
fun filter p s =
     delay (fn () => filter' p (expose s))

and filter' p Empty = Empty
  | filter' p (Cons(x,s')) =
      if (p x) then Cons (x, filter p s')
      else filter' p (expose s')
```

If someone exposes a filtered stream,
code must look for the first element satisfying `p`.
That may entail looking at multiple elements of `s`,
so may need to call `filter'` repeatedly.

Can loop forever!

Example #3

```
val evens = S.map (fn n => 2*n) nats

val [0,2,4] = S.take (evens, 3)
```

Observe: **`evens : int S.stream`**

The stream **`evens`** is a lazy piece of code (sometimes called a *thunk*) that knows how to compute the even natural numbers from the stream **`nats`**, which itself is a thunk that knows how to compute natural numbers. In particular, the number **`4`** does not appear explicitly in **`nats`** or **`evens`**, but eventually is computed by the exposures implicit in the code for **`take`**.

Example #4

```
val ns = S.filter (fn n => n < 0) nats
```

Observe: `ns : int S.stream`

Questions:

1. Is `ns` a value?
2. If so, how long does it take to compute it?
3. Does `S.expose ns` reduce to a value?

Example #4

```
val ns = S.filter (fn n => n < 0) nats
```

Recall:

```
fun filter p s =
      delay (fn () => filter' p (expose s))
```

Questions:

1. Is `ns` a value?
2. If so, how long does it take to compute it?
3. Does `S.expose ns` reduce to a value?

Answers:

1. **Yes.**
2. **The call to `S.filter` returns almost instantaneously.**
3. **No, `S.expose ns` loops forever.**

```
fun filter p s =
     delay (fn () => filter' p (expose s))

and filter' p Empty = Empty
  | filter' p (Cons(x,s')) =
      if (p x) then Cons (x, filter p s')
      else filter' p (expose s')
```

loops forever!

Example #5 : All the primes

Inspired by the Sieve of Erathosthenes

Example #5 : All the primes

Inspired by the Sieve of Erathosthenes

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

Write down all the natural numbers greater than **1**.

Example #5 : All the primes

Inspired by the Sieve of Erathosthenes

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

Find leftmost element (**2** currently).

Example #5 : All the primes

Inspired by the Sieve of Erathosthenes

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, ...

Cross-off all multiples of that leftmost element.

Example #5 : All the primes

Inspired by the Sieve of Erathosthenes

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

3, 5, 7, 9, 11, 13, 15, 17,…

Repeat the process with the remaining numbers.

Example #5 : All the primes

Inspired by the Sieve of Erathosthenes

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

3, 5, 7, ~~9~~, 11, 13, ~~15~~, 17,…

Example #5 : All the primes

Inspired by the Sieve of Erathosthenes

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

3, 5, 7, ~~9~~, 11, 13, ~~15~~, 17, …

5, 7, 11, 13, 17, …

Keep repeating this process.

Example #5 : All the primes

Inspired by the Sieve of Erathosthenes

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

3, 5, 7, ~~9~~, 11, 13, ~~15~~, 17, …

5, 7, 11, 13, 17, …

The diagonal of leftmost elements constitutes all primes.

Example #5 : All the primes

```
fun notDivides p q = (q mod p <> 0)
```

`notDivides p q` returns **`false`** if **`q`** is a multiple of **`p`**, and **`true`** otherwise.

Example #5 : All the primes

```
fun notDivides p q = (q mod p <> 0)


fun sieve s = S.delay (fn () => sieve'(S.expose s))
```

`sieve` delays the actual sieving.

Example #5 : All the primes

```
fun notDivides p q = (q mod p <> 0)

fun sieve s = S.delay (fn () => sieve'(S.expose s))
and sieve' (S.Empty) = S.Empty
```

We don’t really need this clause,
since there are infinitely many primes.

Example #5 : All the primes

```
fun notDivides p q = (q mod p <> 0)

fun sieve s = S.delay (fn () => sieve'(S.expose s))
and sieve' (S.Empty) = S.Empty
  | sieve' (S.Cons(p, s)) =
     S.Cons(p, sieve (S.filter (notDivides p) s))
```

`sieve'` filters out multiples of the element **`p`**
that it finds at the head of its front,
recursively constructs a stream of all larger primes,
and adds **`p`** to the front of that.

Example #5 : All the primes

```
fun notDivides p q = (q mod p <> 0)

fun sieve s = S.delay (fn () => sieve'(S.expose s))
and sieve' (S.Empty) = S.Empty
  | sieve' (S.Cons(p, s)) =
     S.Cons(p, sieve (S.filter (notDivides p) s))

val primes = sieve (S.delay (nat’ 2))
```

All the primes represented lazily.

Example #5 : All the primes

```
fun notDivides p q = (q mod p <> 0)

fun sieve s = S.delay (fn () => sieve'(S.expose s))
and sieve' (S.Empty) = S.Empty
  | sieve' (S.Cons(p, s)) =
     S.Cons(p, sieve (S.filter (notDivides p) s))

val primes = sieve (S.delay (nat’ 2))

val p400 = S.take (primes, 400)
```

The first 400 primes in a list.

The first 400 primes

```
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,
 103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,
 197,199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,
 307,311,313,317,331,337,347,349,353,359,367,373,379,383,389,397,401,409,
 419,421,431,433,439,443,449,457,461,463,467,479,487,491,499,503,509,521,
 523,541,547,557,563,569,571,577,587,593,599,601,607,613,617,619,631,641,
 643,647,653,659,661,673,677,683,691,701,709,719,727,733,739,743,751,757,
 761,769,773,787,797,809,811,821,823,827,829,839,853,857,859,863,877,881,
 883,887,907,911,919,929,937,941,947,953,967,971,977,983,991,997,1009,1013,
 1019,1021,1031,1033,1039,1049,1051,1061,1063,1069,1087,1091,1093,1097,1103,
 1109,1117,1123,1129,1151,1153,1163,1171,1181,1187,1193,1201,1213,1217,1223,
 1229,1231,1237,1249,1259,1277,1279,1283,1289,1291,1297,1301,1303,1307,1319,
 1321,1327,1361,1367,1373,1381,1399,1409,1423,1427,1429,1433,1439,1447,1451,
 1453,1459,1471,1481,1483,1487,1489,1493,1499,1511,1523,1531,1543,1549,1553,
 1559,1567,1571,1579,1583,1597,1601,1607,1609,1613,1619,1621,1627,1637,1657,
 1663,1667,1669,1693,1697,1699,1709,1721,1723,1733,1741,1747,1753,1759,1777,
 1783,1787,1789,1801,1811,1823,1831,1847,1861,1867,1871,1873,1877,1879,1889,
 1901,1907,1913,1931,1933,1949,1951,1973,1979,1987,1993,1997,1999,2003,2011,
 2017,2027,2029,2039,2053,2063,2069,2081,2083,2087,2089,2099,2111,2113,2129,
 2131,2137,2141,2143,2153,2161,2179,2203,2207,2213,2221,2237,2239,2243,2251,
 2267,2269,2273,2281,2287,2293,2297,2309,2311,2333,2339,2341,2347,2351,2357,
 2371,2377,2381,2383,2389,2393,2399,2411,2417,2423,2437,2441,2447,2459,2467,
 2473,2477,2503,2521,2531,2539,2543,2549,2551,2557,2579,2591,2593,2609,2617,
 2621,2633,2647,2657,2659,2663,2671,2677,2683,2687,2689,2693,2699,2707,2711,
 2713,2719,2729,2731,2741]
```

(spec for **`sieve`** might be)

```
(* sieve : int Stream.stream -> int Stream.stream

 REQUIRES: s consists, in ascending order, of all
           natural numbers starting with some
           prime q, excluding all multiples of
           primes less than q.

 ENSURES : (sieve s) returns a stream consisting
           of all primes starting with q, in
           ascending order.

*)
```

Inspired by Euclid's Proof

Theorem: There are infinitely many primes.

Proof:

Suppose $p_1, \ldots, p_n$ are all the primes.

Let $P = p_1 * \cdots * p_n$ and $Q = P + 1$.

$Q > P$, so some p_i divides Q and P.

Thus p_i divides 1,

establishing a contradiction. $\boxed{\text{QED}}$

(Euclid proved that for any finite list of primes, there exists a prime outside the list.)

That is all.

Please have a good weekend.

See you Tuesday, when we will talk about mutation.