Lazy Programming

15-150

Lecture 20: November 18, 2025

Stephanie Balzer
Carnegie Mellon University

Today

So far we have only dealt with finite data structures.

→ But how to represent infinite data structures?

Examples:

- Natural numbers, primes
- Keystrokes made on a keyboard
- My email inbox (😅)
- Video / audio streams

→ To program infinite data structures, we use the notion of a **delayed computation**.

→ Delayed computations also facilitate **demand-driven** (aka **lazy**) programming in a call-by-value language such as SML.

Delayed computation

Idea:

- Encapsulate computation to **suspend** it.
- Execute computation by explicitly **forcing** it.

Can we do that in SML? 🤔

Let's take a step back and ask ourselves the following question:

What is the difference between the following two expressions?

`e` and `fn x => e x`

Here, SML will evaluate `e`.

Here, SML will only evaluate `e`, when the lambda is applied to an argument.

Delayed computation

Idea:

- Encapsulate computation to **suspend** it.
- Execute computation by explicitly **forcing** it.

Can we do that in SML? 🤔

Let's take a step back and ask ourselves the following question:

What is the difference between the following two expressions?

`e` and `fn x => e x`

- Lambdas allow us to suspend computations!
- Lambdas are values (even if encapsulated computation diverges).

Delayed computation

Idea:

- Encapsulate computation to **suspend** it.
- Execute computation by explicitly **forcing** it.

Can we do that in SML? 🤔

For example, given

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

`e` and `fn x => e x`

- Lambdas allow us to suspend computations!
- Lambdas are values (even if encapsulated computation diverges).

Delayed computation

Idea:

- Encapsulate computation to **suspend** it.
- Execute computation by explicitly **forcing** it.

Can we do that in SML? 🤔

For example, given

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

`g 3` loops, but `fn x => (g 3) x` is a value

- Lambdas allow us to suspend computations!
- Lambdas are values (even if encapsulated computation diverges).

Delayed computation

Idea:

- Encapsulate computation to **suspend** it.
- Execute computation by explicitly **forcing** it.

Can we do that in SML? 🤔

- Yes, using lambdas to represent infinite, possibly diverging computations.
- We call such lambdas **suspensions**:

A **suspension** of type `t` is a function `f` of type

```
f: unit -> t
```

Delayed computation

A **suspension** of type `t` is a function `f` of type

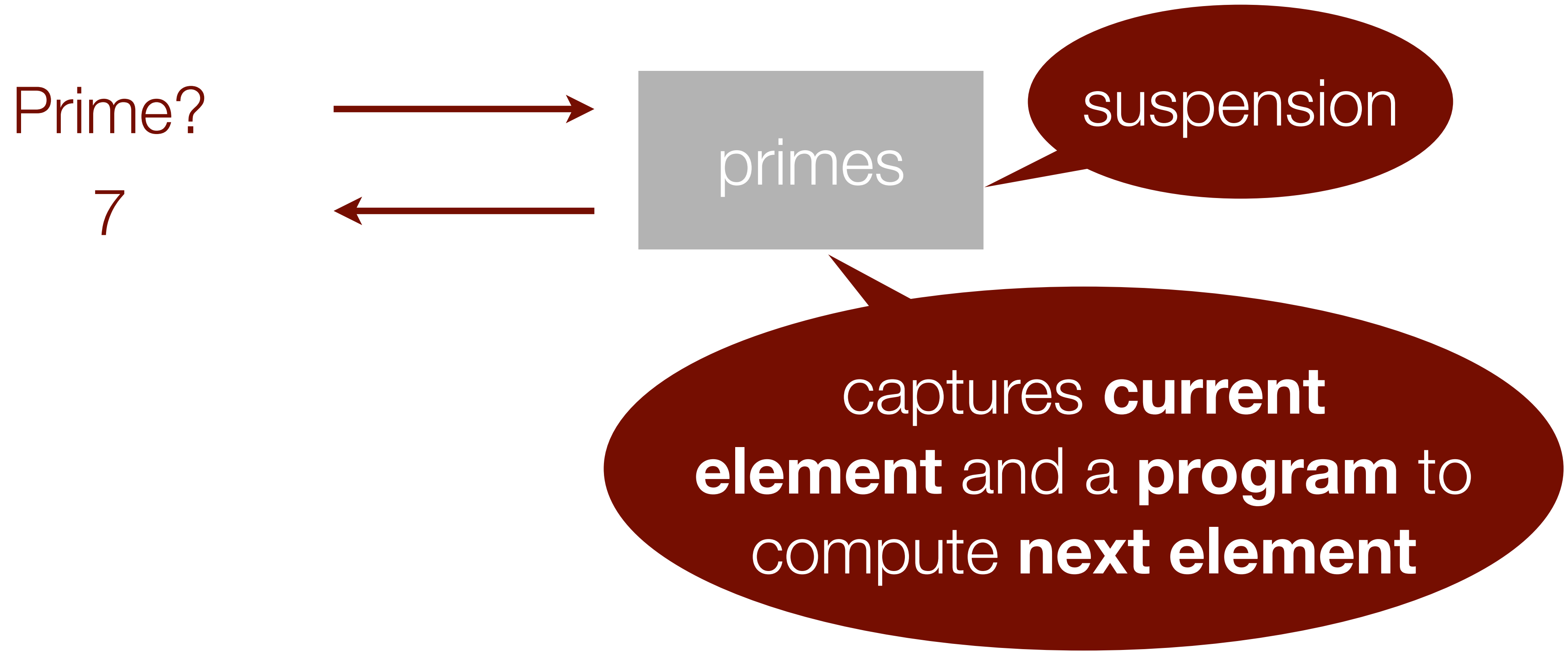
```
f: unit -> t
```

such that for `e: t`, `f` is `fn () => e`.

- A suspension is **forced**, when it is applied, i.e., `f ()`.
- The suspension `f` is a **lazy** representation of `e` because `e` won't be evaluated until `f` is forced.
- Let's use suspensions to represent (possibly infinite) **streams** of data.

Streams*

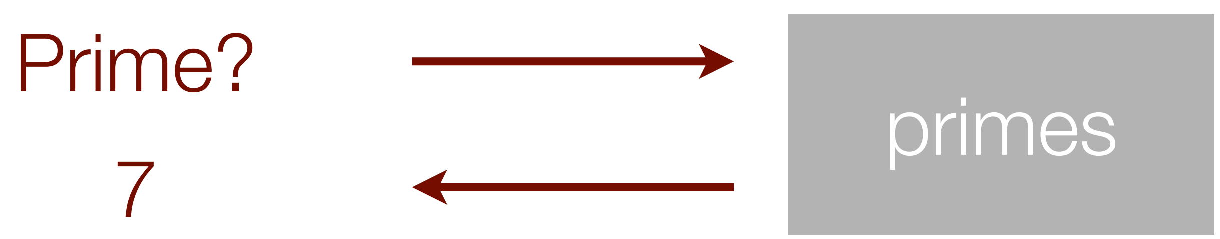
Streams are data structures that are being continuously created, e.g.,



* (Note, different from SML's built-in I/O streams.)

Streams

Streams are data structures that are being continuously created, e.g.,



- We can think of streams as being generated by state machines:
  - only when "kicked" (forcing suspension) they yield element
  - advancing state for computation of next element.
- Streams are defined **coinductively**.

* (Note, different from SML's built-in I/O streams.)

Intermezzo: induction versus coinduction

if you'd like to know

aka, we don't expect you to know

Intermezzo: induction versus coinduction

The data types (e.g., lists, trees) encountered so far were defined **inductively**.

We can view **inductive** and **coinductive** types as **duals** of each other:

- Inductive data types are constructed **upfront** and are thus **finite**.
- Coinductive data types are computed **on demand** (i.e., lazily) and may thus be **infinite**.
- Inductive data types facilitate **proofs by induction**
  - show that all possible ways of construction satisfy property
- Coinductive data types facilitate **proofs by coinduction**
  - show containment of element by consistent behavior

Intermezzo: induction

We can also define corresponding lazy versions!

The data types (e.g., lists, trees) encountered so far were defined **inductively**.

We can view **inductive** and **coinductive** types as **duals** of each other:

- Inductive data types are constructed **upfront** and are thus **finite**.
- Coinductive data types are computed **on demand** (i.e., lazily) and may thus be **infinite**.
- Inductive data types facilitate **proofs by induction**
  - show that all possible ways of construction satisfy property
- Coinductive data types facilitate **proofs by coinduction**
  - show containment of element by consistent behavior

Let's implement streams

- First, we define a signature, capturing streams abstractly.
- Then, we implement them in a corresponding structure.

Stream signature

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

streams with elements of type `'a`

```
end
```

Stream signature

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





end
```

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                                                  (* concrete *)
```

Forcing ("kicking") a stream yields a value of type `'a front`,

```
end
```

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                                                  (* concrete *)
```

Forcing ("kicking") a stream yields a value of type `'a front`,

comprising the current element

```
end
```

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                                                  (* concrete *)
```

Forcing ("kicking") a stream yields a value of type `'a front`,

comprising the current element and the rest of the stream,

```
end
```

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty                   (* concrete *)
```

- Forcing ("kicking") a stream yields a value of type `'a front`,
- comprising the current element and the rest of the stream,
- or `Empty`, in case the stream is finite.

```
end
```

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty                          (* concrete *)






end
```

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty              (* concrete *)

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

Function **expose** forces the computation yielding the current element and the remainder of the stream.

Caution: **expose** may loop!

```
end
```

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty                   (* concrete *)

  val expose : 'a stream -> 'a front




end
```

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty                   (* concrete *)

  val expose : 'a stream -> 'a front

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


end
```

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty               (* concrete *)

  val expose : 'a stream -> 'a front

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

Function `delay` creates a stream, given a suspension for computing the stream.

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty                 (* concrete *)

  val expose : 'a stream -> 'a front

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

Function `delay` creates a stream, given a suspension for computing the stream.

Suspension required, otherwise SML will evaluate argument!

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty                   (* concrete *)

  val expose : 'a stream -> 'a front

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

Stream signature

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

  datatype 'a front = Cons of 'a * 'a stream
                    | Empty               (* concrete *)

  val expose : 'a stream -> 'a front

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

  (* more functions (see accompanying code) *)
end
```

Stream structure

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

- We find it convenient to wrap a `Stream` constructor around the suspension of an `'a front`.
- The use of the constructor `Stream`, instead of the plain suspension, conveys more readily what the function is about.

```
end
```

```
signature STREAM =
sig
  type 'a stream
  datatype 'a front = Cons of 'a * 'a stream | Empty
  ...
end
```

Stream structure

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

- We find it convenient to wrap a `Stream` constructor around the suspension of an `'a front`.
- The use of the constructor `Stream`, instead of the plain suspension, conveys more readily what the function is about.
- Recall: `'a front` refers to `'a stream`.

```
end
```

```
signature STREAM =
sig
  type 'a stream
  datatype 'a front = Cons of 'a * 'a stream | Empty
  ...
end
```

Stream structure

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

- We find it convenient to wrap a `Stream` constructor around the suspension of an `'a front`.
- The use of the constructor `Stream`, instead of the plain suspension, conveys more readily what the function is about.
- Recall: `'a front` refers to `'a stream`.
- How do we handle that?

```
end
```

Stream structure

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

- Define mutually recursive data structures with keyword `and`.
- Recall: `'a front` is already defined as such in signature.

```
end
```

```
signature STREAM =
sig
  type 'a stream
  datatype 'a front = Cons of 'a * 'a stream | Empty
  ...
end
```

Stream structure

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







end
```

Stream structure

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

  (* delay : (unit -> 'front) -> 'a stream *)
  fun delay (d) = Stream(d)
```

➡ Wraps `Stream` constructor around suspension of `'a front`.

```
end
```

Stream structure

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

  (* delay : (unit -> 'front) -> 'a stream *)
  fun delay (d) = Stream(d)




end
```

Stream structure

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

  (* delay : (unit -> 'front) -> 'a stream *)
  fun delay (d) = Stream(d)

  (* expose : 'a stream -> 'a front *)
  fun expose (Stream(d)) = d ()
```

Forces underlying suspension in input stream.

```
end
```

Stream structure

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

  (* delay : (unit -> 'front) -> 'a stream *)
  fun delay (d) = Stream(d)

  (* expose : 'a stream -> 'a front *)
  fun expose (Stream(d)) = d ()


end
```

Stream structure

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

  (* delay : (unit -> 'front) -> 'a stream *)
  fun delay (d) = Stream(d)

  (* expose : 'a stream -> 'a front *)
  fun expose (Stream(d)) = d ()

  (* more functions (see accompanying code) *)
end
```

Let's practice: stream of 1s

Assume that the following codes is written outside the `Stream` structure, where we abbreviate `Stream` with `S` for space reasons.

→ Let's implement an infinite stream whose elements are 1:

```
(* ones' : unit -> int S.front *)
fun ones' () = S.Cons(1, S.delay ones')

(* int S.stream *)
val ones = S.delay ones'
```

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Let's practice: stream of 1s

Assume that the following codes is written outside the `Stream` structure, where we abbreviate `Stream` with `S` for space reasons.

Let's implement an infinite stream whose elements are 1:

```
(* ones' : unit -> int S.front *)
fun ones' () = S.Cons(1, S.delay ones')

(* int S.stream *)
val ones = S.delay ones'
```

current element

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Let's practice: stream of 1s

Assume that the following codes is written outside the `Stream` structure, where we abbreviate `Stream` with `S` for space reasons.

➡ **Let's implement an infinite stream whose elements are 1:**

```
(* ones' : unit -> int S.front *)
fun ones' () = S.Cons(1, S.delay ones')

(* int S.stream *)
val ones = S.delay ones'
```

current element

remains the same in tail

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Let's practice: stream of nats

Assume that the following codes is written outside the `Stream` structure, where we abbreviate `Stream` with `S` for space reasons.

Let's implement an infinite stream of all natural numbers:

```
(* nat' : int -> unit -> int S.front *)
fun nat' x () = S.Cons(x, S.delay (nat' (x+1)))

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

initial element

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Let's practice: stream of nats

Assume that the following codes is written outside the `Stream` structure, where we abbreviate `Stream` with `S` for space reasons.

Let's implement an infinite stream of all natural numbers:

```
(* nat' : int -> unit -> int S.front *)
fun nat' x () = S.Cons(x, S.delay (nat' (x+1)))

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

current element

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Let's practice: stream of nats

Assume that the following codes is written outside the `Stream` structure, where we abbreviate `Stream` with `S` for space reasons.

Let's implement an infinite stream of all natural numbers:

```
(* nat' : int -> unit -> int S.front *)
fun nat' x () = S.Cons(x, S.delay (nat' (x+1)))

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

current element

next element

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Let's practice: stream of nats

```
(* nat' : int -> unit -> int S.front *)
fun nat' x () = S.Cons(x, S.delay (nat' (x+1)))
(* int S.stream *)
val nats = S.delay (nat' 0)
```

Consider now:

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

What values are `x` and `y` bound to? What does `tail` represent?

Recall:
```
(* expose : 'a stream -> 'a front *)
fun expose (Stream(d)) = d ()
```

Let's practice: stream of nats

```
(* nat' : int -> unit -> int S.front *)
fun nat' x () = S.Cons(x, S.delay (nat' (x+1)))
(* int S.stream *)
val nats = S.delay (nat' 0)
```

Consider now:

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

What values are `x` and `y` bound to? What does `tail` represent?

- `x` is bound to 0 and `y` to 1
- `tail` denotes the stream of all natural numbers greater than 0

Let's practice: stream of nats

```
(* nat' : int -> unit -> int S.front *)
fun nat' x () = S.Cons(x, S.delay (nat' (x+1)))
(* int S.stream *)
val nats = S.delay (nat' 0)
```

Consider now:

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

What value is `z` bound to?

z is bound to 0

Memoization for efficiency

- Each time we force the same stream, the element is recomputed.
- **Memoization** allows us to remember a computed value for a stream, so that when forced, the stored value is simply returned.
- On Thursday, we will introduce **reference cells**, which precisely allow us to do that.
  - initially, reference cell contains suspension
  - after force, reference cell contains computed value

When are two streams equivalent?

To define equivalence, we augment our signature with this function:

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

`take(s,n)` returns the first `n` elements of stream `s` as a list.

When are two streams equivalent?

To define equivalence, we augment our signature with this 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, for all integers `n` $\geq$ 0:

```
Stream.take(X,n) ≅ Stream.take(Y,n)
```

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

`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`.

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

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

Find leftmost element (`2` currently).

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

```
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.

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

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.

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

```
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.

Another example: prime numbers

Inspired by the Sieve of Eratosthenes.

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.

Another example: prime numbers

To implement this algorithm, we augment our signature with the following function:

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
```

Moreover, we define locally, the following helper function:

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

returns false if q is a multiple of p

otherwise true

Another example: prime numbers

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
val notDivides p q = (q mod p <> 0)
```

Now, the algorithm:

```
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))
```

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Another example: prime numbers

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
val notDivides p q = (q mod p <> 0)
```

Now, the algorithm:

delays actual sieving

```
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))
```

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Another example: prime numbers

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
val notDivides p q = (q mod p <> 0)
```

Now, the algorithm:

```
fun sieve s = S.delay (fn () => si… pose 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))
```

> not really needed because primes are infinite

Recall:
```
(* delay : (unit -> 'front) -> 'a stream *)
fun delay (d) = Stream(d)
```

Another example: prime numbers

```
val filter : ('a -> bool) -> 'a stream -> 'a stream
val notDivides p q = (q mod p <> 0)
```

Now, the algorithm:

```
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'
```

recursively constructs stream of larger primes, with p at front

filters multiples of current element p

Recall: `(* delay` `-> ` `am *)`
`fun delay`

That's all for today.