Today

• How to represent infinite data in ML using streams as an example

• Big idea: Delay computations and do them only when needed (lazily) – demand-driven computation
Revisiting evaluation in SML

e vs (fn x => e x)
Revisiting evaluation in SML

\[ e \quad \text{vs} \quad (\text{fn } x \Rightarrow e \ x) \]

\(e\) may be a function name or an expression that reduces to a function.

On the left, \(e\) is evaluated right away.

On the right, \(e\) is evaluated only when the lambda expression is applied.
Revisiting evaluation in SML

\[ g \ 3 \quad \text{vs} \quad (\text{fn} \ x \Rightarrow (g \ 3) \ x) \]

where \textbf{fun} \ g \ x = g \ x
Revisiting evaluation in SML

\[ g \ 3 \quad \text{vs} \quad (\texttt{fn} \ x \Rightarrow (g \ 3) \ x) \]

where \texttt{fun} \ \ g \ x = g \ x

\[ g \ 3 \quad \text{loops forever} \]

\[ (\texttt{fn} \ x \Rightarrow (g \ 3)) \ x \quad \text{is a value} \]
SML uses call-by-value evaluation

Functions *always* evaluate their arguments

\[(\text{fn} \ x \ => \ e_1) \ e_2\]

First \(e_2\) is evaluated to obtain a value \(v_2\), and then \([v_2/x] \ e_1\) is evaluated.

ML is an *eager* language
Example: Building an infinite list

repeat: 'a list -> 'a list

fun repeat L = L @ repeat L
Example: Building an infinite list

repeat: 'a list -> 'a list

fun repeat L = L @ repeat L

repeat [0] ==> (fn L => L @ repeat L) [0]
  ==> [[0]/L] L @ repeat L
  ==> [0] @ repeat [0]
  ...

loops for ever
Being lazy

Debe computation until the result is needed

• A function value called a *suspension* (*thunk*)

  \[
  h : \text{unit} \rightarrow t
  \]

  represents a delayed computation for a value of type \(t\)

• To compute this value call \(h()\) to *force* the suspension
signature STREAM =
sig
  type 'a stream (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream
  val empty : 'a stream
  val delay : (unit -> 'a front) -> 'a stream
  val expose : 'a stream -> 'a front

  exception EmptyStream

  val null : 'a stream -> bool
  val head : 'a stream -> 'a
  val tail : 'a stream -> 'a stream

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

  val take : 'a stream * int -> 'a list
end
structure Stream : STREAM =
struct
    datatype 'a stream = Stream of unit -> 'a front
    and 'a front = Empty | Cons of 'a * 'a stream
structure Stream : STREAM =
struct
    datatype 'a stream = Stream of unit -> 'a front
    and 'a front = Empty | Cons of 'a * 'a stream

    val empty = _____________
    fun delay (d) = ______________
    fun expose (Stream(d)) = _____________

Recall the signature:

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

continued on the next page
structure Stream : STREAM =

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

  val empty = Stream(fn () => Empty)
  fun delay (d) = Stream(d)
  fun expose (Stream(d)) = d()
structure Stream : STREAM =
struct
...
  exception EmptyStream

fun null (s) =
structure Stream : STREAM =
struct
...

exception EmptyStream

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

fun head (s) = case (expose s) of
    Empty => raise EmptyStream
    | (Cons(x, _)) => x

fun tail (s) = case (expose s) of
    Empty => raise EmptyStream
    | (Cons(_, s)) => s

continued on the next page
structure Stream : STREAM =
struct

...  
  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)

  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)
structure Stream : STREAM =
struct

  ...

  fun take (s, 0) = nil
  | take (s, n) = take' (expose s, n)

and take' (Empty, _) = raise EmptyStream
  | take' (Cons(x, s), n) = x :: take(s, n-1)

end

continued on the next page
Sieve of Eratosthanes

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

2 is the first prime.
Sieve of Eratosthanes

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

2 is the first prime.
Filter out all multiples of 2.
Sieve of Eratosthanes

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

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

3 is the next prime.
Sieve of Eratosthanes

3 is the next prime.
Filter out all multiples of 3.
Sieve of Eratosthenes

5 is the next prime.
(* notDivides : int -> int -> bool
   REQUIRES: true
   ENSURES: notDivides p q ==> true iff p does not divide q,
            (q is not a multiple of p).
*)
fun notDivides p q = (q mod p <> 0)

(* sieve: int Stream.stream -> int Stream.stream *)
fun sieve s = Stream.delay (fn () => sieve' (Stream.expose s))
and sieve' (Stream.Empty) = Stream.Empty
  | sieve' (Stream.Cons(p, s)) =
      Stream.Cons(p, sieve (Stream.filter (notDivides p) s))

(* All the primes as a stream: *)
val primes = sieve (delay (natsFrom 2))

(* The first 100 primes as a list: *)
val p100 = Stream.take(primes, 100)
fun cons(x,s) = Stream(fn() => Cons(x,s))

fun lazyappend([],s) = s
  | lazyappend(x::xs, s) = cons(x, lazyappend(xs, s))

fun repeat xs = lazyappend(xs,
                          delay (fn () => expose (repeat xs))

Our new repeat has type 'a list -> 'a stream

Contrast it what we tried to do at the beginning of the lecture.

repeat: 'a list -> 'a list

fun repeat L = L @ repeat L
fun append (s1, s2) = delay
          (fn () => append' (expose s1, s2))
and append' (Empty, s2) = expose s2
| append' (Cons(x, s1), s2) =
       Cons(x, append (s1, s2))

fun repeat s = append (s,
                  delay (fn () => expose (repeat s)))