15–150: Principles of Functional Programming

Lazy Functional Programming

Michael Erdmann*

Spring 2025

1 Topics

- Lazy programming (in a call-by-value language)
- Delaying computation
- Demand-driven manipulation of infinite data

2 Introduction

SML is a call-by-value language: functions evaluate their arguments. So an attempt to build an infinite list, such as the following code fragment, is doomed to failure (actually, will loop forever):

```
(* repeat : 'a list -> 'a list *)
fun repeat L = L @ repeat L
```

Even though SML tells us this function is well-typed, if we apply it to a list we won't get a value. For example

```
repeat [0] ==> [0] @ repeat [0]
           ==> [0] @ ([0] @ repeat [0])
           ==> . . . forever
```

Although this is a silly example, it illustrates a problem. We may need to build and manipulate infinite data structures, and deal with such data in a demand-driven manner: just evaluate as much as we need, while leaving the rest for later if we need it. The call-by-value behavior of functions needs to be side-stepped somehow, if we want to prevent eager evaluation of the rest of the data.

In fact we can simply use *functions* themselves to delay computation. Function values (e.g., lambda expressions) *are* values: SML doesn't evaluate *inside* a function body until, if and when, the function is applied to an argument value.

*Adapted from a document by Stephen Brookes.

3 Representing Infinite Data

In a functional programming language, one way to represent an infinite list of data of some type `t` is as a function `int -> t`. However, this kind of representation is not well-tailored for lazy demand-driven computation, particularly when future data may depend on external input. Moreover, such a representation may create a lot of tedious index manipulation.

Today we will introduce another representation for potentially infinite lists, which we call *streams*. We will implement operations on streams that facilitate solving non-trivial demand-driven problems.

The main idea is that a `t stream` of values of type `t` is really a delayed computation. We delay a computation by making it be the body of a function known as a *suspension*, sometimes also called a *thunk*. When the function is called on its argument, it computes the *head* of the stream, along with another suspension for computing the next element of the stream, and so forth.

Definition: A *suspension* of a type τ is a function of type `unit -> `τ. We say a suspension is *forced* when it is applied to argument `()`.

The reason we care about suspensions is that we want to delay evaluation. In an eager language like SML, writing `e` (e.g., in the REPL) causes evaluation of `e`. Writing `(fn () => e)` prevents immediate evaluation of `e` since lambda expressions are already values. Only when forced will the suspension cause evaluation of `e`.

Streams: We now define streams as follows:

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

Observe that for any type `t`, a `t stream` is basically a suspension of a `t front`, which looks much like a `t list`, only now as deferred computation. Specifically, a (nonempty) `t front` is a "cons" of an element of type `t` and another `t stream`. (Aside: Alternatively, we could have defined `type 'a stream = unit -> 'a front` without a `Stream` constructor. We chose the current design for visual clarity when debugging code.)

We need a function for turning a suspension of a front into a stream, and for exposing the first element of a stream by forcing the underlying suspension:

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

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

We also have an empty stream, which is essentially just a suspension of an empty front:

```
val empty : 'a stream = Stream(fn () => Empty)
```

Finally, we have a function for adding an element to the beginning of a stream, returning the resulting stream:

```
(* cons : 'a * 'a stream -> 'a stream *)
fun cons (x, s) = Stream(fn () => Cons(x, s))
```

3.1 Examples

Here are some examples that construct values of type `int stream`:

```
(* zeros : unit -> int front *)
fun zeros () = Cons(0, delay zeros)

(* Zeros : int stream *)
val Zeros = delay zeros
```

`Zeros` is an infinite stream of 0s.

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

(* Ones : int stream *)
val Ones = delay ones
```

`Ones` is an infinite stream of 1s.

```
(* streamify : 'a list -> 'a stream *)
fun streamify [ ] = empty
  | streamify (x::L) = cons(x, streamify L)
```

The function `streamify` turns a list into a finite stream with the same elements as in the list (and in the same order).

```
(* natsFrom : int -> (unit -> int front) *)
fun natsFrom n () = Cons(n, delay (natsFrom (n+1)))

(* Nats : int stream *)
val Nats = delay (natsFrom 0)
```

The function `natsFrom` constructs a suspension of a front, that represents the natural numbers from some number upward. `Nats` is the stream of all natural numbers, in ascending order.

```
(* lazyappend : 'a list * 'a stream -> 'a stream *)
fun lazyappend ([ ], s) = s
  | lazyappend (x::xs, s) = cons(x, lazyappend(xs,s))
```

The function `lazyappend` appends a list to a stream, returning a stream of all the list elements followed by all the original stream elements.

```
(* repeat : 'a list -> 'a stream                                    *)
(* REQUIRES: xs is not nil                                         *)
(* ENSURES:  take(repeat xs, n) returns a value for all n >= 0.    *)
(*           (see definition of take below)                        *)
fun repeat xs = lazyappend(xs, delay(fn () => expose(repeat xs)))
```

The function `repeat` expects a list and returns a stream that represents infinite appends of the list to itself. For example, `repeat [1,2,3]` is the stream consisting of $1, 2, 3, 1, 2, 3, \ldots$, repeating forever.

What happens if we evaluate `repeat [ ]`?

What happens if we evaluate `expose(repeat [ ])`?

(Ignore the value restriction when answering those two questions.)

3.2 Viewing a Stream

Here is a useful function that shows us a "view" of a stream: we can look at the first `n` items in the stream, as an ordinary finite list.

```
(* take  : 'a stream * int -> 'a list
   take' : 'a front  * int -> 'a list
   REQUIRES:  n >= 0
   ENSURES:   take(s,n) returns a list of the first n elements of s in order,
              unless s has fewer than n elements in which case take(s,n)
              raises Subscript.  If computation of any of these n elements
              loops forever or raises an exception, then so will take(s,n).
*)
fun take (s, 0) = nil
  | take (s, n) = take' (expose s, n)
and take' (Empty, _) = raise Subscript
  | take' (Cons(x,s), n) = x::take(s, n-1)
```

Example: `take(Nats, 5) = [0,1,2,3,4]`.

Note: "taking" the first few elements of a stream forces the first few suspensions to be applied to their argument `()` to extract data from the tail of the stream. This is all purely functional, without side-effect. If we evaluate `take(Nats, 2)` after `take(Nats, 5)` we will get the result `[0,1]`, not `[5,6]`. The first take doesn't change the value of the stream.

Observe: The mutual recursion between `take` and `take'` is a useful template for writing stream functions: one function specializes to streams, the other to fronts, and they hand data back and forth between each other. The mutual recursion in the code mirrors the mutual recursion in the `stream` and `front` datatypes. (One does not *need* mutual recursion, but it is a useful template to remember.)

3.3 Example: Reals as Infinite Streams of Integers

A stream of integers could be used to represent a *real number*. We could use the head of the stream to represent the *integer part* of the real number, and then the rest of the stream to represent the decimal digits, with the most significant digit (the tenths part) first, then the hundredths part, and so on. According to this representation, `Ones` represents the real number whose decimal expansion is `1.11111....`

```
val X = lazyappend ([5, 8], repeat [1,4,4])
(* X represents the real number equal to 3227/555 *)
(* This number has decimal expansion 5.8144144144. . . *)

val Y = repeat [0,1,2,3,4,5,6,7,9]
(* Y represents 10/81 *)
(* The decimal expansion for 10/81 is 0.12345679012345679. . . *)

val Z = repeat [0,9]
(* Z represents 10/11 *)
(* The decimal expansion for 10/11 is 0.909090. . . *)
```

3.4 Higher-Order Functions on Streams

Notation: In the rest of this document, `Stream` is a structure ascribing to a `STREAM` signature, both defined in today's lecture code. The types and key functions presented so far appear within that `Stream` structure, as do the following functions.

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

  `map' : ('a -> 'b) -> 'a front -> 'b front`

  ```
  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)
  ```

  If value `f : t1 -> t2` is total and value `s : t1 Stream.stream` has elements $d_0, d_1, \ldots, d_n, \ldots$, then `(Stream.map f s)` has type `t2 Stream.stream` and elements $\texttt{f}(d_0), \texttt{f}(d_1), \ldots, \texttt{f}(d_n), \ldots$.

  Again, notice the mutual recursion between a function that operates on streams and one that operates on fronts. Also, observe that `(map f s)` returns a delayed computation. Let's call it `ms`. Not until someone, if ever, evaluates `expose(ms)` does any actual mapping occur, and then only to the first stream element, with further mapping of the rest of the stream itself delayed until requested via an `expose` of the rest of the mapped stream. Etc.

  Just for reference, here is an implementation of `map` that does not use mutual recursion:

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

- `zip : 'a stream * 'b stream -> ('a * 'b) stream`

  `zip' : 'a front * 'b front -> ('a * 'b) front`

```
fun zip (s1, s2) = delay (fn () => zip'(expose s1, expose s2))
and zip' (_, Empty) = Empty
  | zip' (Empty, _) = Empty
  | zip' (Cons(x, s1), Cons(y, s2)) = Cons((x, y), zip(s1, s2))
```

  If value `A : t1 Stream.stream` has elements $a_0, a_1, \ldots, a_n, \ldots$
  and value `B : t2 Stream.stream` has elements $b_0, b_1, \ldots, b_n, \ldots$,
  then `Stream.zip(A,B)` evaluates to a value `Z : (t1 * t2) stream`,
  with elements $(a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n), \ldots$.

  Observe that if either or both of streams `A` and `B` is finite,
  then the resulting stream `Stream.zip(A,B)` has the length
  of the shorter of `A` and `B`.

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

  `filter' : ('a -> bool) -> 'a front -> 'a front`

```
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 value `p : t -> bool` is total and value `s` is a stream of type `t Stream.stream`, then `Stream.filter p s` computes the stream of elements of `s` that satisfy `p`.

  If `s` has an element satisfying `p`, and `p` is total, then `Stream.expose(Stream.filter p s)` returns a front `Stream.Cons(x, rest)`, where `x` is the first element of `s` that satisfies `p` and `rest` is a stream representing the remaining elements of `s` satisfying `p`.

  What happens if we evaluate `Stream.expose(Stream.filter (fn _ => false) Ones)`?

It is natural to see the first two functions (`map` and `zip`) as "extensions" of the corresponding functions on ordinary lists, because one can prove the following results:

For all non-negative `n`, and all suitably typed values `f`, `s` and `t`, with `f` total:

```
Stream.take(Stream.map f s, n) ≅ List.map f (Stream.take(s, n))
Stream.take(Stream.zip (s, t), n) ≅ List.zip (Stream.take(s, n), Stream.take(t, n))
```

3.5 Equivalence of Streams

When should we say that two stream expressions are extensionally equivalent? In what sense are

```
Stream.map (f o g) s
Stream.map f (Stream.map g s)
```

extensionally equivalent, assuming as usual that `f` and `g` are total functions whose types allow them to be composed?

A notion of equivalence that makes sense for streams is that stream values `X` and `Y` are *extensionally equivalent* iff, for all $\texttt{n} \geq 0$, `Stream.take(X, n)` and `Stream.take(Y, n)` are equivalent. This definition also allows for the possibility that an attempt to extract an element may loop forever (as in the `filter` example when there is no element satisfying the predicate). So two streams whose first 42 elements are equivalent and whose 43rd elements result in infinite loops when exposed are also regarded as equivalent.

It is impossible to implement an equality-check for infinite streams, even for `int stream`s. Unlike ordinary lists of integers, we cannot use the built-in `=` operator of SML to check for equality. Instead, one generally must establish stream equivalence by proof.

3.6 Lazy Fusion Equivalences

Given the notion of stream equivalence above, the following *lazy fusion* results are provable:

> For all types `t1, t2, t3`, all total function values `f:t2->t3` and `g:t1->t2`, and all values `s : t1 Stream.stream`,
>
> `Stream.map (f o g) s` $\cong$ `Stream.map f (Stream.map g s)`.

The proof proceeds by showing that (under the given assumptions) for all $\texttt{n} \geq 0$,

`Stream.take(Stream.map (f o g) s, n)` $\cong$ `Stream.take(Stream.map f (Stream.map g s), n)`.

In fact, we can appeal to the earlier result linking `Stream.map` and `List.map`, and the familiar fusion property of `List.map`, i.e., that

`List.map (f o g) (Stream.take(s, n))` $\cong$ `List.map f (List.map g (Stream.take(s, n)))`

to derive the lazy fusion equivalence stated above.

Note that we can paraphrase this lazy fusion equivalence, taking advantage of extensionality. The following is an equivalent statement to the one given above:

> For all types `t1, t2, t3`, and all total function values `f:t2->t3` and `g:t1->t2`,
>
> `Stream.map (f o g)` $\cong$ `(Stream.map f) o (Stream.map g)`.

3.7 Productive and Infinite Streams

Exposing a stream can lead to infinite looping or a raised exception. The following definition describes streams for which this does *not* occur. We say a stream `s` of type `t Stream.stream` is *productive* if and only if `Stream.expose(s)` returns either `Stream.Empty` or `Stream.Cons(x,s')`, with `x` a value of type `t` and `s'` again a productive stream.

We have already spoken of infinite streams intuitively. More formally, we say a stream constructed with structure `Stream` is *infinite* if and only if it is productive and successive exposures never encounter `Stream.Empty`.

4 More Examples

This section presents two examples illustrating how to program with streams. The art is in the design of suspensions (often using recursion), and making the control flow sufficiently lazy that we never prematurely reveal the head-tail (i.e., front) structure of part of a stream until we need to.

To improve readability, let us suppose we have defined an abbreviation:

```
structure S = Stream
```

with `Stream` being a structure ascribing to a `STREAM` signature, as defined in today's lecture code.

4.1 Streams of Factorials

Useful facts about factorials:

- $0! = 1$
- $k! * (k+1) = (k+1)!$ when $k \geq 0$.

The following lazy way to generate factorials is based on these useful facts. The first fact tells us that the first element of the stream should be 1. The second fact tells us that we can obtain the later factorials by zipping each integer $k+1$ with the factorial of its predecessor and multiplying. So we embed a recipe for generating factorials starting from 0!, by zipping and multiplying, in the tail suspension.

```
(* factorials : unit -> int front *)
fun factorials () = S.Cons(1, S.map (op * )
                                   (S.zip (S.delay factorials, S.delay (natsFrom 1))))

(* Factorials : int stream *)
val Factorials = S.delay factorials
```

`Factorials` is a stream consisting of all factorials $n!$, for $n \geq 0$.
In other words, for all non-negative `n`, `S.take(Factorials, n+1)` evaluates to the list

`[0!, 1!, . . ., n!].`

```
S.take (Factorials, 10) ==> [1,1,2,6,24,120,720,5040,40320,362880].
```

4.2 Stream of Prime Numbers

This code is inspired by the Sieve of Eratosthenes, an ancient algorithm for enumerating the primes.

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

(*
   sieve  : int S.stream -> int S.stream
   sieve' : int S.front  -> int S.front
*)
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(natsFrom 2)
```

`primes` is a stream representing all prime numbers, in ascending order. Here are the first 400:

```
  S.take (primes, 400) ==>

  [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]
```