Lecture 18: Streams

Before we define streams, let us revisit and constrast some properties tuples and lists.

Lists are ...

- type-homogeneous;
- of variable length (e. g. a list of a given type can be extended by adding elements to it, and any list can have any length);
- of finite length.

Tuples are ...

- not necessarily type-homogeneous;
- of fixed arity (for a given type);
- of finite arity.

While these datastructures are very general, and many problems can be solved with their help, there are desireable properties that neither lists, nor tuples have.

For example, we can not access the i^{th} element of the tuple using a simple expression, say, of the form #n, where n would be a variable (but remember that we have projection operators defined for any fixed, positive n compatible with the type of the tuple). We can use the `List.nth` operator to access the n^{th} element of a list.

Another drawback of these two datastructures is their finiteness. This statement might sound surprising initially, since any datastructure that we want to represent must be finite for obvious reasons. There are applications, however, in which it is more convenient to structure the computation assuming that we have one or more infinite datasources, from which we can extract as much data as we need for the problem at hand. If we are looking, for example, for the smallest prime number with a particular property, we might not know how far in the infinite sequence of ordered prime numbers we need to search to find it. In such a situation, it might make sense to conceive of our computation as using a potentially infinite datasource that produces prime numbers on demand. A stream is such a datasource.

We stress that "infinite" and "infinity" refer to potential (theoretical) infinity, and not actual infinity. Due to the obvious limitations of various resources (time, memory, and others), no computation will produce an infinite sequence of values. Our streams are infinite in potential, not in fact.

Before we define the stream datatype, let us think for a minute about how we could specify an infinite stream of values. It is immediately clear that we can not actually enumerate these values, so we are left with the alternative of providing a method for computing them. This means that our streams must rely on an infinity of function calls. This is only possible if (at least some of) the functions involved are directly or indirectly recursive. Now, the infinite sequence of function calls can not run to the "end" (there is no end of infinity, of course); at any given time, only a finite number of such calls must have been initiated. Thus me must have a mechanism for (temporarily) stopping further recursive calls. If we understand how we can suspend, and later resume, the sequence of computations that generates the stream values, then we can write streams in SML.

Consider the following definition:

```
datatype 'a stream = Null | Cons of 'a * (unit -> 'a stream)
```

First, note that this definition is very similar to our definition of custom lists.

We see that a stream can be either empty (`Null`), or it can consist of a pair. The first element of the pair is a value (the head of the stream), while the second element of the pair is a 0-argument function that produces a stream. The stream that the 0-argument function returns is the tail (the rest) of our original stream.

Commenting in general about 0-argument functions, you might remark that they are pretty much useless in the absence of side-effects: while a function whose domain is other than `unit` admits in general many values for its arguments, a 0-argument function can only be called with `unit`. Since we

know what the argument will be, why would we not evaluate the body of the function, rather than delaying the evaluation until the function is called with the - otherwise useless - `unit` argument?

If you made this comment, you would be right, as it is often useful just to evaluate the body of a 0-argument function without any further delay (remember, we assume that there are no side-effects!). Here, however, it is precisely the delaying ("lazy") property of this postponed function evaluation that we are interested in. By relying on it, we will be able to stop an infinite sequence of recursive calls in its tracks.

This being said, let us now define the simplest stream - an infinite stream that consists of the same repeated value, ad infinitum:

```
fun const(c: 'a) = Cons(c, fn() => const c)
```

But how do we use such a stream? Will the suspended computations ever be trigerred? Here are some functions that work on streams:

```
exception Empty

(* Returns the first element of a stream. *)
fun hd(s: 'a stream): 'a =
  case s of
    Null       => raise Empty
  | Cons(h, _) => h

(* Returns the stream that results after removing the first element. *)
fun tl(s: 'a stream): 'a stream =
  case s of
    Null       => raise Empty
  | Cons(_, t) => t()

(* Applies a function to every element of a stream. *)
fun map (f: 'a -> 'b) (s: 'a stream): 'b stream =
  case s of
    Null  => Null
  | Cons(h, t) => Cons(f h, fn () => map f (t()))


(* Returns the ordered list of the first n elements of the stream.  *)
fun takeN(s: 'a stream, n: int): 'a list =
  case (s, n) of
    (_, 0)          => []
  | (Null, _)       => raise Empty
  | (Cons(h, t), n) => h :: (takeN (t(), n - 1))


(* Produces a stream of values that satisfy a predicate. *)
fun filter (f: 'a -> bool) (s: 'a stream): 'a stream =
  case s of
    Null => Null
  | Cons(h, t) => if f(h) then Cons(h, fn () => filter f (t()))
                          else filter f (t())
```

Before demonstrating how these functions work, let us define two more streams:

```
fun nats(n: int) = Cons(n, fn () => nats(n + 1))
fun fibo(a: int, b: int) = Cons(a, fn () => fibo(b, a + b))
```

Function `nats(n)` generates a sequence of successive integers starting at the initial value `n`. When called with the argument `0`, we obtain the stream of natural numbers.

Function `fibo` produces Fibonacci-like sequences; when called with arguments `0` and `1`, respectively, it generates the usual Fibonacci sequence 0, 1, 1, 2, 3, ...

The reader might have noticed that all three streams we have defined up to now are infinite, but this does not have to be the case. Indeed, it is easy to define finite, non-empty streams:

```
- val s1 = Cons(9, fn () => Cons(8, fn () => Cons(7, fn () => Null)))
val s1 = Cons (9,fn) : int stream
```

And now, let us demonstrate how stream functions work:

```
- takeN(fibo(0, 1), 10);
val it = [0,1,1,2,3,5,8,13,21,34] : int list
-
- hd s1;
val it = 9 : int
-
- tl s1;
```

```
val it = Cons (8,fn) : int stream
-
- hd(tl(tl s1));
val it = 7 : int
-
- takeN(filter (fn n => n mod 2 = 0) (fibo(0, 1)), 4);
val it = [0,2,8,34] : int list
-
- takeN(map (fn n => n * n) s1, 3);
val it = [81,64,49] : int list
```

Returning for a moment to finite streams, let us create a stream from a list of values given as argument:

```
fun fromList(l: 'a list): 'a stream =
  case l of
    []   => Null
  | h::t => Cons(h, fn () => fromList t)

- takeN(fromList [9, 7, 5, 3, 1], 4);
val it = [9,7,5,3] : int list

- takeN(fromList [9, 7, 5, 3, 1], 10);

uncaught exception Empty
  raised at: stdIn:824.30-824.35
```

Note that it is impossible to extract more elements from a finite stream than are available.

Given two streams, what are some meaningful operations that we can define on them? Perhaps surprisingly, one such operation is stream concatenation. Given stream `s1` and `s2`, the concatenation of stream `s1` and `s2` consists of the ordered sequence of all values in `s1`, followed by the ordered sequence of all the values in `s2`. It is obvious that if `s1` is infinite, then no value will ever be extracted from `s2`. On the other hand, streams can be both finite and infinite, so it might be that all values from `s1` are consumed, and values from `s2` will we used:

```
fun concatenate(s1: 'a stream, s2: 'a stream): 'a stream =
  case s1 of
    Null       => s2
  | Cons(h, t) => Cons(h, fn () => concatenate(t(), s2))

- takeN(concatenate(fromList [1,2,3,4,5], fromList [6,7,8,9,10]),  7);
val it = [1,2,3,4,5,6,7] : int list
```

We finish today's discussion of streams by examining an implementation of the Sieve of Eratosthenes, possibly the oldest systematic method (algorithm) for generating the sequence of all prime numbers. The "sieve" can be described as follows:

```
step 1: Generate the sequence of natural numbers starting at 2.
step 2: Position yourself just before the beginning of the sequence.
step 3: Find the next available number in the sequence. Write it down; it is prime.
step 4: Cross out (delete) all multiples of the number identified in step 3.
step 5: Continue with step 3.
```

Of course, one can never work with an actual infinite sequence of numbers, nor can one cross out an infinity of multiples.

We will now implement the Sieve:

```
fun sift (p: int) (s: int stream): int stream =
  filter (fn n => n mod p <> 0) s

fun sieve (s: int stream): int stream =
  case s of
    Null       => Null
  | Cons(s, t) => Cons(s, fn () => sieve(sift s (t())))

val primes = sieve(nats 2);
```

Did you ever wonder what is the 312th prime? You can now find out:

```
- List.hd(rev(takeN(primes, 312)));
val it = 2069 : int
```