Streams

A *stream* is an infinite list. Sometimes these are also called sequences, delayed lists, or lazy lists. We can try to define a stream by analogy to how we can define (finite) lists. Recall that definition:

```
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist
```

We could try to convert that into a definition for streams:

```
(* doesn't actually work *)
type 'a stream =
  | Cons of 'a * 'a stream
```

Note that we got rid of the `Nil` constructor, because the empty list is finite, but we want only infinite lists.

The problem with that definition is that it's really no better than the built-in list in OCaml, in that we still can't define `nats`:

```
# let rec from n = Cons (n, from (n+1));;

# let nats = from 0;;
Stack overflow during evaluation (looping recursion?).
```

As before, that definition attempts to go off and compute the entire infinite sequence of naturals.

What we need is a way to *pause* evaluation, so that at any point in time, only a finite approximation to the infinite sequence has been computed. Fortunately, we already know how to do that!

Consider the following definitions:

```
# let f1 = failwith "oops";;
Exception: Failure "oops".

# let f2 = fun x -> failwith "oops";;
val f2 : 'a -> 'b = <fun>

# f2 ();;
Exception: Failure "oops".
```

The definition of `f1` immediately raises an exception, whereas the definition of `f2` does not.

Why? Because `f2` wraps the `failwith` inside an anonymous function. Recall that, according to the dynamic semantics of OCaml, **functions are already values**. So no computation is done inside the body of the function until it is applied. That's why `f2 ()` raises an exception.

We can use this property of evaluation—that functions delay evaluation—to our advantage in defining streams: let's wrap the tail of a stream inside a function. Since it doesn't really matter what argument that function takes, we might as well let it be unit. A function that is used just to delay computation, and in particular one that takes unit as input, is called a *thunk*.

```
(* An ['a stream] is an infinite list of values of type ['a].
 * AF:  [Cons (x, f)] is the stream whose head is [x] and tail is [f()].
 * RI:  none.
 *)
type 'a stream =
  Cons of 'a * (unit -> 'a stream)
```

This definition turns out to work quite well. We can define `nats`, at last:

```
# let rec from n = Cons (n, fun () -> from (n+1));;
val from : int -> int stream = <fun>

# let nats = from 0;;
val nats : int stream = Cons (0, <fun>)
```

We do not get an infinite loop or a stack overflow. The evaluation of `nats` has paused. Only the first element of it, `0`, has been computed. The remaining elements will not be computed until they are requested. To do that, we can define functions to access parts of a stream, similarly to how we can access parts of a list:

```
(* [hd s] is the head of [s] *)
let hd (Cons (h, _)) = h

(* [tl s] is the tail of [s] *)
let tl (Cons (_, tf)) = tf ()

(* [take n s] is the list of the first [n] elements of [s] *)
let rec take n s =
  if n=0 then []
  else hd s :: take (n-1) (tl s)

(* [drop n s] is all but the first [n] elements of [s] *)
let rec drop n s =
  if n = 0 then s
  else drop (n-1) (tl s)
```

It is informative to observe the types of those functions:

```
val hd : 'a stream -> 'a
val tl : 'a stream -> 'a stream
val take : int -> 'a stream -> 'a list
val drop : int -> 'a stream -> 'a stream
```

Note how, in the definition of `tl`, we must apply the function `tf` to `()` to obtain the tail of the stream. That is, we must *force* the thunk to evaluate at that point, rather than continue to delay its computation.

We can use `take` to observe a finite prefix of a stream. For example:

```
# take 10 nats;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```