Code Review 7: April 3, 2020

Infinite Streams and Lazy Evaluation

Hakeem Angulu, hangulu@college.harvard.edu

Learning Goals

- Conceptualize infinite data streams
- Understand the purposes of memoization
- Appreciate the power of Lazy evaluation

Streams

A stream is an **infinite sequence** of data. When we say "infinite," we mean "possibly infinite," in that there is no defined end. It can be thought of as a `List` that keeps giving us more elements based on a predefined rule, but **never before we need or ask for them**.

Why do we use streams?

One of the most clear advantages is flexibility. Infinite data structures and lazy evaluation mean you don't have to worry about how large your data is ahead of time. Think about a social network like Facebook: they have constant queries to their system, handling data that seemingly never stops, but needs to be operated on. Streams are how they deal with that problem.

Another is delayed computation. Sometimes operations are incredibly computationally expensive to do on entire `Lists`. When only part of the `List` is needed at a time, this situation may be a good candidate for the use of streams.

Let's examine the stream module from Lab 14:

```
module LazyStream =
  struct

    type 'a stream_internal = Cons of 'a * 'a stream
     and 'a stream = unit -> 'a stream_internal ;;

    (* Extracting the head and tail of a lazy stream *)
    let head (s : 'a stream) : 'a =
      let Cons(h, _t) = s() in h ;;

    let tail (s : 'a stream) : 'a stream =
```

```
    let Cons(_h, t) = s() in t ;;

  (* Extracting the first n elements of a stream into a list *)
  let rec first (n : int) (s : 'a stream) : 'a list =
    if n = 0 then []
    else head s :: first (n - 1) (tail s) ;;

  (* Mapping a function lazily over a stream *)
  let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
    fun () -> Cons(f (head s), smap f (tail s)) ;;

  (* Mapping a binary function over two streams *)
  let rec smap2 f s1 s2 =
    fun () -> Cons(f (head s1) (head s2),
                   smap2 f (tail s1) (tail s2)) ;;
end ;;
```

Exercise 1: What is the difference between a `stream` and a `stream_internal`?

The `stream_internal` stores the actual data, while the `stream` provides the functionality of delaying and forcing computation when necessary.

Exercise 2: Split a stream into two streams, where the first element goes at the front of the first stream. Think of this as an "unzipping" process.

```
let rec split (s: 'a stream) : 'a stream * 'a stream =
   let t = tail (tail s) in
   (fun () -> Cons (head s, fst (split t))),
          (fun () -> Cons (head (tail s), snd (split t))) ;;
```

Lazy Evaluation

Lazy evaluation is one of the most powerful and most used processes in OCaml. It allows us to operate on streams while **memoizing** our results to save time. For regular streams, you'll often have to recompute computationally expensive results every time you call a function, but OCaml's inbuilt Lazy module memoizes the results upon evaluation so that next time the function is called with those parameters, the result is simply returned from memory.

Lazy evaluation more broadly is an evaluation strategy which delays the evaluation of an expression until its value is needed and which avoids repeated evaluations, typically through memoization. This is also known as the call-by-need evaluation strategy.

This is in contrast to other evaluation strategies, including the **strict** or **eager** evaluation strategies we are used to in OCaml and other programming languages.

The built-in `Lazy` module introduces a new type – `'a Lazy.t` – of delayed elements of type `'a`, and a new function `Lazy.force : 'a Lazy.t -> 'a` that forces a delayed computation to occur, saving the result if this is the first time the value was forced and simply returning the saved value on later requests.

This replaces much of the functionality that we've seen in the other implementation of streams. One defining the types using the Lazy module is below, from Lab 15:

```
type 'a stream_internal = Cons of 'a * 'a stream
 and 'a stream = 'a stream_internal Lazy.t ;;
```

How do we get from an `'a stream_internal` to an `'a stream`? `'a stream` to `'a stream_internal`?

Syntactically, it works the same as before, except instead of using `fun() ->` as a wrapper, we use `lazy`. For example:

```
let rec ones = lazy (Cons (1, ones))
```

Instead of forcing computation with `s()`, we use `Lazy.force`.

Exercise 3: Redefine the split function using the Lazy module (as in Lab 15).

```
let rec split (s: 'a stream) : 'a stream * 'a stream =
  let t = tail (tail s) in
  lazy(Cons head s, fst (split t)),
  lazy(Cons head (tail s), snd (split t)) ;;
```

The solutions to all of the above exercises will be available by Sunday at noon.

For this and future code reviews, please do not hesitate to reach out to me with any questions or concerns. My email can be found at the top of this document.