CS51 **Spring 2024**

Code Review 7: Lazy Programming and Infinite Data Structures

1 Practice Problems

Problem 1.1 (Laziness). Define a function `(&&&) : bool Lazy.t -> bool Lazy.t -> bool`. It should behave like a short circuit Boolean AND. That is, `lb1 &&& lb2` should first force `lb1`. If the value is false, the function should return false. Otherwise, it should force lb2 and return its value.

```
let (&&&) (lb1 : bool Lazy.t) (lb2 : bool Lazy.t) : bool =
  let lb1 = Lazy.force lb1 in
  if lb1 = false then false
  else Lazy.force lb2 ;;
```

Problem 1.2 (Defining Streams). [1]

1. Define a value `pow2 : int stream` whose elements are the powers of two.

```
    let rec pow2 : int stream =
        lazy (Cons (1, smap (( * ) 2) pow2)) ;;
```

2. Define a stream whose elements are the lowercase letters of the alphabet on repeat: a, b, c ....., z, a, b, c ..., z ... *Hint* : The chr and code function in OCaml's Char module might be helpful

```
     let rec alpha_stream : char stream =
      let alpha_size = 26 in
      let next_char (chr : char) : char =
        let cde = Char.code chr in
        let a_code = Char.code 'a' in
        Char.chr (((cde + 1) - a_code) mod alpha_size + a_code) in
      lazy (Cons ('a', smap next_char alpha_stream)) ;;
```

3. Suppose we have the following type:

```
type flip = Heads | Tails
```

Define a stream of pseudorandom coin flips. Hint: `Random.int bound` function returns a random integer between 0 (inclusive) and bound (exclusive).

```
let rec random_stream : flip stream =
    let rec generate () : flip stream =
        if Random.int 2 = 0 then lazy (Cons (Heads, generate ()))
        else  lazy (Cons (Tails, generate ()))
    in generate () ;;
```

[1] https://courses.cs.cornell.edu/cs3110/2021sp/textbook/adv/exercises.html

Note the subtelty here. One might write a solution like this.

```
let rec random_stream : flip stream =
  lazy (Cons ((if Random.int 2 = 0 then Heads else Tails),
     random_stream)) ;;
```

This is incorrect. Problem with this is that you end up getting the same coin flip every time.

Problem 1.3 (Stream Function Exercises)**.**

1. Define a function `expo_term : float -> float stream`, that returns the stream of the Taylor expansion of e^x. The Taylor expansion of e^x is defined as:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

One possible implementation:

```
let expo_term (x : float) : float stream =
 let rec factorial (n : int) : int =
   if n = 0 then 1
   else n * factorial (n - 1) in
 let rec power (x' : float) (n : int) : float =
   if n = 0 then 1.
   else x' *. power x' (n - 1) in
 let rec generate (n : int) : float stream =
   lazy (Cons ((power x n) /. (float_of_int) (factorial n),
      generate (n + 1)))
 in generate 0 ;;
```

Another implementation

```
let expo_term (x : float) : float stream =
  let rec generate (prev : float) (n : int) : float stream =
    lazy (
    let new_el = if n = 0 then 1. else  (x /.  (float_of_int n)) *.
        prev in
    Cons ( new_el , generate new_el (n + 1))) in
  generate 1. 0 ;;
```

Consider which implementation is better. It might look like implementation (1) is worse because computing the power and factorial for each element in the stream is inefficient, but is it so bad? If we re-use the stream, then we will store the computed values of the stream.

Implementation (2) seems like it saves time the first time we perform computations on the stream.

2. Define a function `total : int stream -> int stream`, that takes in a stream `<a; b; c; ...>` and outputs a stream of the running total of the input elements, i.e., `<a; a +. b; a +. b +. c; ...>`. [2]

[2] https://courses.cs.cornell.edu/cs3110/2021sp/textbook/adv/exercises.html

```
let total (s : int stream) : int stream =
  let rec generate (prev_sum : int) (s' : int stream) : int stream
     =
    lazy (
      let curr_sum = head s' + prev_sum in
      Cons (curr_sum , generate curr_sum (tail s'))) in
  generate 0 s ;;
```

3. Define a function `infinite_delay : ('a -> 'a) list -> 'a list -> 'a Lazy.t stream` that takes a list of functions and arguments to those functions (of the same size), and returns a repeating stream of delayed computations (functions applied to corresponding argument). For instance, consider the function list to be `[f; g; h]` and the argument list to be `[x;y;z]`. The output stream should represent the delayed computations of `f x, g y, h z, f x, g y, ...`. Though if we force computations in the stream, each computation should only be computed once.

```
let infinite_delay (funcs : ('a -> 'a) list) (args : 'a list) : 'a
   Lazy.t stream =
  if List.length funcs <> List.length args then raise (
     Invalid_argument "funcs and args
  aren't same length")
  else
  (let delayed_comps = List.map2 (fun f x -> lazy (f x)) funcs args
      in
   let rec generate_seq (lst : 'a Lazy.t list) : 'a Lazy.t stream =
    let rec aux (l : 'a Lazy.t list) : 'a Lazy.t stream =
      match l with
      | [] -> generate_seq lst
      | h :: t -> lazy (Cons (h, aux t)) in
    aux lst in
  generate_seq delayed_comps) ;;
```

One might come up with this solution (incorrect):

```
let rec infinite_delay (funcs : ('a -> 'a) list) (args : 'a list) :
    'a Lazy.t stream =
  let rec aux (f : ('a -> 'a) list) (a : 'a list) : 'a Lazy.t
     stream =
    match funcs, args with
    | [], [] -> infinite_delay funcs args
    | [], _
    | _, [] -> raise (Invalid_argument "function and arg lists not
       of eq length")
    | fh :: ft, ah :: at -> lazy (Cons (lazy (fh ah), aux ft at) )
       in
  aux funcs args;;
```

The problem here is that you are not performing each computation **exactly once**, as two of the same computation would belong to *two different thunks* everytime they appear in the list.

4. Define a function `compute : ('a -> 'a list) -> 'a list -> unit -> 'a` that returns a generator that repeatedly returns the next computation each time the function is called. For example, if we have a function list `[f; g; h]` and `[x; y; z]`, then this function should return `f x`, then `g y`, then `h z`, `f x`, and so on.

```
let rec compute (funcs : ('a -> 'a) list) (args : 'a list) : unit
   -> 'a =
  let comp_stream = ref (infinite_delay funcs args) in
  fun () ->
    let Cons (h, t) = Lazy.force !comp_stream in
    comp_stream := t; Lazy.force h ;;
```

Problem 1.4 (Lazy Trees)**.**

1. Define an `'a tree` type for infinite trees. Each node in the tree can have any number of children.

```
type 'a tree_internal =
  Node of 'a * 'a tree list and
'a tree = 'a tree_internal Lazy.t ;;
```

2. Write a function numNodes that takes an `'a tree` and a positive integer d and returns the number of nodes in the `'a tree` above depth d. We say that the root node of the tree is at depth 0, its children are at depth 1, its grandchildren are at depth 2, and so on.

```
let rec numNodes (tr : 'a tree) (d : int) : int =
    if d = 0 then 1
    else
    let Node (h, children) = Lazy.force tr in
    1 + List.fold_left (fun acc x -> numNodes x (d - 1) + acc) 0
       children ;;
```