CS51 **Spring 2024**

Code Review 7: Lazy Programming and Infinite Data Structures

1 Lazy Evaluation

Definition 1.1 (Lazy Evaluation)**.** `OCaml` is an **eager** language, meaning that expressions are evaluated immediately when **encountered**.

Lazy Evaluation is a method in which expressions are not evaluated until their values are **needed**.

We **delay** a computation until it's needed, and when we actually need the value of the computation, we **force** the computation to occur.

Please read the discussion in section 17.1 of the textbook.

Example 1.2 (Delaying Computation using Functions)**.** One way to delay computation is to use *functions*.

Suppose we have the following code:

```
print_endline "Hello"; 5 + 1 ;;
```

Since OCaml is an eager language, this computation is evaluated eagerly; in the `REPL`, `"Hello"` is printed out and the value of the expression `5 + 1` is computed.

We can **delay** the computation of this expression by wrapping it around a function.

```
fun () -> print_endline "Hello"; 5 + 1
```

This works because **functions are values in OCaml**; the function body isn't evaluated when we define a function.

We can **force** the computation by **applying** the function (i.e `f ()`).

Remark 1.3 (Expressions and Values)**.** It's important to note the distinction between **expressions** and **values**.

Values are the building blocks of computation. Expressions are combinations of values. You can think of a value to an expression as an atom to a molecule.

We *reduce* expressions to *values*. All values are expressions, but not all expressions are values. If an expression is well-defined, the `OCaml` interpreter will reduce the expression to a value.

That's why, since functions are themselves values, we can delay a computation C by wrapping it around a function (`fun x -> C`).

2 Streams

Definition 2.1 (Stream)**.** In computer science, a stream is a sequence of elements made available over time. [1]

In this class, we'll define a stream as a sequence of elements of type `'a` like an "infinite list".

[1] https://en.wikipedia.org/wiki/Stream_(computing)

Note that the idea of streams can also be applied to define "infinite" versions of "finite" data structure such as trees, which you'll see in the problem set.

Here's how someone might naively define a stream:

Listing 1: Naive Stream Implementation (Incorrect)

```
type 'a stream = Cons of 'a * 'a stream ;;

let head (Cons (hd, _tl) : 'a stream) : 'a = hd ;;

let tail (Cons (_hd, tl) : 'a stream) : 'a stream = tl ;;

let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
  match s with
  | Cons (hd, tl) -> Cons (f hd, smap f tl) ;;
```

Here, we define a stream as similarly to how lists are defined, but leave out the base case (i.e a value constructor for `Nil`).

Consider, if wrote the following code:

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

let twos = smap succ ones ;;
Stack overflow during evaluation (looping recursion?).
```

The problem with this definition is that **it's impossible to eagerly evaluate over an infinite sequence of elements**, so we need to use a mechanism to **delay computation**. As mentioned above, **one way** we can delay computation is to use functions.

Listing 2: Stream Implementation (Correct)

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


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 ;;

let rec smap (f : 'a -> 'b) (s : 'a stream)
           : ('b stream) =
  fun () -> Cons (f (head s), smap f (tail s)) ;;
```

Here, we delay computation of stream values until we need them by defining a stream as a function of type `unit -> 'a stream_internal`.

Remark 2.2 (Tips for Working with Streams)**.** Many students find it uncomfortable at first to solve problems with streams. Here are some tips:

- Streams can often be thought of "infinite versions" of "finite" data structures. The distinction is that with streams, we delay computation. Students often find it easier to think about what you would do with the finite data structure. It helps to think about how to write a function with the finite data structure and then generalize to a function on the infinite stream. For example, to define `smap`, we could think about how we defined `map` for lists and generalize to streams.
- As we have said throughout this course, **think about the typing of functions you write**. If we define streams as functions, and your function returns a stream, then your function must return a function with the appropriate type.

Example 2.3 (Using Streams)**.** Here's an example of how we can define and use our definition of streams.

```
let rec ones : int stream =
  fun () -> Cons (1, ones) ;;

  let rec twos : int stream =
    smap ((+) 1) ones ;;

let threes =
  smap2 (+) ones twos ;;

let rec nats =
  fun () -> Cons (0, smap succ nats) ;;
```

Another way to implement `nats` could be:

```
let rec nats : int stream =
    let helper (n : int) : int stream =
      fun () -> Cons (n, helper (n + 1)) in
helper 0 ;;
```

3 Memoization

Definition 3.1 (Memoization)**.** In addition to delaying computation, we might to *optimize* computations by **memorizing** the results of previous computations.

Memoization is a technique where we avoid recomputation by **remembering** the value the first time it is computed.

3.1 Thunks

Definition 3.2 (Thunk)**.** A **thunk** is a data structure used to store the results of computation.

The advantage of using a **thunk** over a function to delay computation is that we can **memoize** computation in addition to **delaying** computation.

In `OCaml`, here's how we could define a thunk type:

Listing 3: Thunk Type Definition

```
type 'a thunk = 'a thunk_internal ref
 and 'a thunk_internal =
 | Unevaluated of (unit -> 'a)
 | Evaluated of 'a ;;
```

An `'a thunk` delays and memoizes computations that evaluate to a value of type `'a`. We can think of a computation of being in two states, `evaluated` or `unevaluated`.

We can **force** computation by writing a function that evaluates a computation inside a thunk:

Listing 4: force

```
let rec force (t : 'a thunk) : 'a =
  match !t with
  | Evaluated v -> v
  | Unevaluated f ->
      t := Evaluated (f ());
      force t ;;
```

If the computation has already been evaluated, we return the value, otherwise we actually perform the computation by calling `f()`, and then return the value.

Example 3.3 (Fibonacci). Recall the `fib` function that computes the nth Fibonacci number:

```
  let rec fib (n : int) : int =
   if n < 2 then n
   else (fib (n - 1)) + (fib (n - 2)) ;;
```

As n gets larger, computing `fib n` can take a long time; the theoretical runtime of `fib` is $O(2^n)$.

Therefore, we might want to **memoize** repeated Fibonacci calculations.

Suppose we defined a thunk to compute `fib 42`:

```
let t : int thunk = ref (Unevaluated (fun () -> fib 42)) ;;
```

Now, let's see what happens when we compute `fib 42` twice in a row:

```
# Absbook.call_reporting_time force t ;;
   time (msecs): 7141.613960
- : int = 267914296
 # Absbook.call_reporting_time force t ;;
time (msecs): 0.000954
- : int = 267914296
```

We see that the second computation was much faster.

3.2 Lazy Module

Definition 3.4 (Lazy Module). Thunks are built-in the `OCaml`, as the language developers provide the `Lazy` module with the `'a Lazy.t` abstract data type.

The following operations are supported:

- `lazy:'a -> 'a Lazy.t` (note that `lazy` is a keyword, not a function). We use the `lazy` keyword to delay computation of type `'a`. It accepts an expression of type `'a` and returns a value of type `'a Lazy.t` (a thunk like we implemented above).
- `Lazy.force : 'a Lazy.t -> 'a` : Forces the evaluation of a computation and returns a value (has the same semantics as `force` which we defined above).

Example 3.5 (Fibonacci with Lazy Module)**.** Here's how we would delay the computation of `fib 42` using a thunk (**as shown in lab 15**).

Listing 5: Lazy Fibonaci Example

```
1
2  let fib42 : int Lazy.t =
3    lazy (fib 42) ;;
4
5  (*  Force computation by calling Lazy.force fib42
6      # Absbook.call_reporting_time Lazy.force fib42 ;;
7      Elapsed time: 13.380860
8      - : int = 267914296
9      # Absbook.call_reporting_time Lazy.force fib42 ;;
10     Elapsed time: 0.000000
11     - : int = 267914296
12 *)
```

3.3 Native Lazy Streams

Now that we know what thunks and the Lazy Module are, we can re-define streams.

Definition 3.6 (Lazy Streams)**.** Here's how we define streams using the Lazy Module:

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

Similar to the `int` type in Listing 5, `stream_internal` is the value of the result with the deferred computation. In other words, when we call `Lazy.force` on an object of type `stream`, we get a `stream_internal`, which consists of an element of type `'a` and an `'a` stream. `stream` is defined to be the deferred computation (or suspension) of the `stream_internal`.

4 Practice Problems

Problem 4.1 (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.

Problem 4.2 (Defining Streams)**.** [2]

1. Define a value `pow2 : int stream` whose elements are the powers of two.
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
3. Suppose we have the following type:

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

   Define a stream of pseudorandom coin flips.

Problem 4.3 (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:

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

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; ...>`. [3]
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.
4. Define a function `compute : ('a -> 'a list) -> 'a list -> 'a` that takes in a list of computations and 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.

Problem 4.4 (Lazy Trees)**.**

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

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

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.