Streams, Laziness and Memoization

CS3100 Fall 2019

Review

Previously

- Modular Programming
  - Namespacing, Abstraction, Code Reuse
  - Structures, Signatures, Functors

This lecture

- Streams: Programming with infinite data structures
- Laziness: Call-by-need evaluation

Recursive values

- In OCaml, we can define recursive functions.
  - we can also define **recursive values**

In [1]:

```
(* Infinite list of ones *)
let rec ones = 1::ones
```

Out[1]:

```
val ones : int list = [1; <cycle>]
```

In [2]:

```
(* Infinite list of alternating 0s and 1s *)
let rec zero_ones = 0::1::zero_ones
```

Out[2]:

```
val zero_ones : int list = [0; 1; <cycle>]
```

Even though the list is **infinite**, the data structure uses **finite** memory.

Infinite data structures

Infinite data structures are not just an intellectual curiosity.

- Infinite sequences such as primes and fibonacci numbers.
- Streams of input read from file or socket.
- Game trees which may be infinite
  - Every possible move leads to branch in the tree.
  - Imagine game trees where a piece could chase the other around forever.

Limitations of cyclic structures

Suppose we want to convert the infinite list `zero_ones` to string, the obvious solutions don't work.

In [3]:

```
let zero_ones_string = List.map string_of_int zero_ones
```

```
Stack overflow during evaluation (looping recursion?).
Raised by primitive operation at file "list.ml", line 88, c
haracters 20-23
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
Called from file "list.ml", line 88, characters 32-39
```

List to Streams

We can start with the list type

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

and make a **stream** type.

In [4]:

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

Out[4]:

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

There is no `Nil` since the streams are infinite.

Doesn't quite work

In [5]:

```
let rec zero_ones = Cons (0, Cons (1, zero_ones))
```

Out[5]:

```
val zero_ones : int stream = Cons (0, Cons (1, <cycle>))
```

In [6]:

```
let rec to_string (Cons(x,xs)) = Cons(string_of_int x, to_string xs)
```

Out[6]:

```
val to_string : int stream -> string stream = <fun>
```

In [7]:

```
to_string zero_ones
```

```
Stack overflow during evaluation (looping recursion?).
Raised by primitive operation at file "[6]", line 1, charac
ters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
Called from file "[6]", line 1, characters 55-67
```

Pausing the execution

- We need a way to pause the execution rather than recursively applying to the rest of the list.
- Use **thunks**: `unit -> 'a` functions.

In [8]:

```
let v = failwith "error"
```

```
Exception: Failure "error".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "[8]", line 1, characters 8-24
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

Pausing the execution

In [9]:

```
let f = fun () -> failwith "error"
```

Out[9]:

```
val f : unit -> 'a = <fun>
```

In [10]:

```
f ()
```

```
Exception: Failure "error".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

Streams again

In [11]:

```
type 'a stream = Cons of 'a * (unit -> 'a stream)
```

Out[11]:

```
type 'a stream = Cons of 'a * (unit -> 'a stream)
```

In [12]:

```
let rec zero_ones = Cons (0, fun () -> Cons (1, fun () -> zero_ones))
```

Out[12]:

```
val zero_ones : int stream = Cons (0, <fun>)
```

In [13]:

```
let hd (Cons (x, _)) = x
```

Out[13]:

```
val hd : 'a stream -> 'a = <fun>
```

In [14]:

```
let tl (Cons (_, xs)) = xs ()
```

Out[14]:

```
val tl : 'a stream -> 'a stream = <fun>
```

More Stream functions

In [15]:

```
let rec take n s =
  if n = 0 then []
  else (hd s)::(take (n-1) (tl s))
```

Out[15]:

```
val take : int -> 'a stream -> 'a list = <fun>
```

In [16]:

```
take 10 zero_ones
```

Out[16]:

```
- : int list = [0; 1; 0; 1; 0; 1; 0; 1; 0; 1]
```

In [17]:

```
let rec drop n s =
  if n = 0 then s
  else drop (n-1) (tl s)
```

Out[17]:

```
val drop : int -> 'a stream -> 'a stream = <fun>
```

In [18]:

```
drop 1 zero_ones
```

Out[18]:

```
- : int stream = Cons (1, <fun>)
```

Higher order functions on streams

In [19]:

```
let rec map f s = Cons (f (hd s), fun () -> map f (tl s))
```

Out[19]:

```
val map : ('a -> 'b) -> 'a stream -> 'b stream = <fun>
```

In [20]:

```
let zero_ones_str = map string_of_int zero_ones
```

Out[20]:

```
val zero_ones_str : string stream = Cons ("0", <fun>)
```

In [21]:

```
take 10 zero_ones_str
```

Out[21]:

```
- : string list = ["0"; "1"; "0"; "1"; "0"; "1"; "0"; "1";
"0"; "1"]
```

Higher order functions on streams

In [22]:

```
(** [filter p s] returns a new stream where every element [x] in [s]
    such that [p x = true] is removed *)
let rec filter p s =
  if p (hd s) then filter p (tl s)
  else Cons (hd s, fun () -> filter p (tl s))
```

Out[22]:

```
val filter : ('a -> bool) -> 'a stream -> 'a stream = <fun>
```

In [23]:

```
let s' = filter ((=) 0) zero_ones in
take 10 s'
```

Out[23]:

```
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1]
```

Higher order functions on streams

In [24]:

```
let rec zip f s1 s2 = Cons (f (hd s1) (hd s2), fun () -> zip f (tl s1) (
```

Out[24]:

```
val zip : ('a -> 'b -> 'c) -> 'a stream -> 'b stream -> 'c
stream = <fun>
```

In [25]:

```
zip (fun x y -> (x,y)) zero_ones zero_ones_str
```

Out[25]:

```
- : (int * string) stream = Cons ((0, "0"), <fun>)
```

Primes

- **Sieve of Eratosthenes**: Neat way to compute primes.
- Start with a stream `s` of `[2;3;4;.....]`.
- At each step,
  - `p = hd s` is a prime.
  - return a new stream `s'` such that $\forall x. x \bmod p \notin s'$

- In the first step,
  - `prime = 2`
  - `new stream = [3;5;7;9;11;13;15;17;....]`
- In the second step,
  - `prime = 3`
  - `new stream = [5;7;11;13;17;19;23;....]`

Primes

In [26]:

```
let rec from n = Cons (n, fun () -> from (n+1));;
from 2
```

Out[26]:

```
val from : int -> int stream = <fun>
```

Out[26]:

```
- : int stream = Cons (2, <fun>)
```

In [27]:

```
let primes_stream =
  let rec primes s = Cons (hd s, fun () ->
    primes @@ filter (fun x -> x mod (hd s) = 0) (tl s))
  in primes (from 2)
```

Out[27]:

```
val primes_stream : int stream = Cons (2, <fun>)
```

In [28]:

```
take 100 @@ primes_stream
```

Out[28]:

```
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 5
3; 59; 61; 67; 71;
 73; 79; 83; 89; 97; 101; 103; 107; 109; 113; 127; 131; 13
7; 139; 149; 151;
 157; 163; 167; 173; 179; 181; 191; 193; 197; 199; 211; 22
3; 227; 229; 233;
 239; 241; 251; 257; 263; 269; 271; 277; 281; 283; 293; 30
7; 311; 313; 317;
 331; 337; 347; 349; 353; 359; 367; 373; 379; 383; 389; 39
7; 401; 409; 419;
 421; 431; 433; 439; 443; 449; 457; 461; 463; 467; 479; 48
7; 491; 499; 503;
 509; 521; 523; 541]
```

Fibonacci sequence

- Let's consider Fibonacci sequence
  - `s1 = [1;1;2;3;5;8;13;...]`
- Let's consider the tail of `s1`
  - `s2 = [1;2;3;5;8;13;....]`
- Let's zip `s1` and `s2` by adding together the elements:
  - `s3 = [2;3;5;6;13;21;...]`
  - `s3` is nothing but the tail of tail of fibonacci sequence.
- If we were to prepend `[1;1]` to `s3` we will have the fibonacci sequence.

Fibonacci sequence

In [29]:

```
let rec fibs =
  Cons (1, fun () ->
    Cons (1, fun () ->
      zip (+) fibs (tl fibs)))
```

Out[29]:

```
val fibs : int stream = Cons (1, <fun>)
```

In [30]:

```
take 10 fibs
```

Out[30]:

```
- : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55]
```

Fibonacci sequence

- Each time we force the computation of the next element, we compute the fibonacci of previous element twice.
  - Not immediately apparent, but this is equivalent to:

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

There is an exponential increase in the running time of `fib(n)` for each increase in `n`.

Lazy Values

- It would be nice to **save** the results of the execution for previously seen values and reuse them.
  - This is the idea behind lazy values in OCaml.
- Lazy values are the opt-in, explicit, **call-by-need** reduction strategy for OCaml
  - Rest of the language is strict i.e, call-by-value
- Lazy module in OCaml is:

```
module Lazy = struct
  type 'a t = 'a lazy_t
  val force : 'a t -> 'a
end
```

OCaml has syntactic support for lazy values through the `lazy` keyword.

Lazy values

In [31]:

```
let v = lazy (10 + (print_endline "Hello"; 20))
```

Out[31]:

```
val v : int lazy_t = <lazy>
```

In [32]:

```
Lazy.force v
```

```
Hello
```

Out[32]:

```
- : int = 30
```

In [33]:

```
Lazy.force v
```

Out[33]:

```
- : int = 30
```

Lazy fib

In [34]:

```
let fib30lazy = lazy (take 30 fibs |> List.rev |> List.hd)
```

Out[34]:

```
val fib30lazy : int lazy_t = <lazy>
```

In [35]:

```
Lazy.force fib30lazy
```

Out[35]:

```
- : int = 832040
```

In [36]:

```
let fib31lazy = take 31 fibs |> List.rev |> List.hd
```

Out[36]:

```
val fib31lazy : int = 1346269
```

Lazy stream

Let's redefine the stream using lazy values.

In [37]:

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

Out[37]:

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

In [38]:

```
let hd (Cons (x,l)) = x
let tl (Cons (x,l)) = Lazy.force l
let rec take n s =
  if n = 0 then [] else (hd s)::(take (n-1) (tl s))
let rec zip f s1 s2 =
  Cons (f (hd s1) (hd s2), lazy (zip f (tl s1) (tl s2)))
```

Out[38]:

```
val hd : 'a stream -> 'a = <fun>
```

Out[38]:

```
val tl : 'a stream -> 'a stream = <fun>
```

Out[38]:

```
val take : int -> 'a stream -> 'a list = <fun>
```

Out[38]:

```
val zip : ('a -> 'b -> 'c) -> 'a stream -> 'b stream -> 'c
stream = <fun>
```

Fibs Lazy Streams

In [39]:

```
let rec fibslazystream =
  Cons (1, lazy (
    Cons (1, lazy (
      zip (+) fibslazystream (tl fibslazystream)))))
```

Out[39]:

```
val fibslazystream : int stream = Cons (1, <lazy>)
```

In [40]:

```
take 30 fibslazystream
```

Out[40]:

```
- : int list =
[1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233; 377; 610;
987; 1597; 2584;
 4181; 6765; 10946; 17711; 28657; 46368; 75025; 121393; 196
418; 317811;
 514229; 832040]
```

You can see that this is fast!

Memoization

- Lazy values in OCaml are a specific efficient implementation of the general idea of caching called **Memoization**.
  - Add caching to functions to retrieve results fast.

In [41]:

```
let memo f =
  let cache = Hashtbl.create 16 in
  fun v ->
    match Hashtbl.find_opt cache v with
    | None ->
        let res = f v in
        Hashtbl.add cache v res;
        res
    | Some res -> res
```

Out[41]:

```
val memo : ('a -> 'b) -> 'a -> 'b = <fun>
```

Expensive identity

In [42]:

```
let rec spin n = if n = 0 then () else spin (n-1)
```

Out[42]:

```
val spin : int -> unit = <fun>
```

In [43]:

```
let expensive_id x = spin 200000000; x
```

Out[43]:

```
val expensive_id : 'a -> 'a = <fun>
```

In [44]:

```
expensive_id 10
```

Out[44]:

```
- : int = 10
```

Memoizing expensive identity

In [45]:

```
let memoized_expensive_id = memo expensive_id
```

Out[45]:

```
val memoized_expensive_id : '_weak1 -> '_weak1 = <fun>
```

In [46]:

```
memoized_expensive_id 11
```

Out[46]:

```
- : int = 11
```

Memoizing recursive functions

- Memoizing recursive functions is a bit more tricky.
  - We need to tie the **recursive knot**

In [47]:

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

Out[47]:

```
val fib : int -> int = <fun>
```

In [48]:

```
fib 40
```

Out[48]:

```
- : int = 165580141
```

Memoizing recursive functions

Simply doing `let memo_fib = memo fib` will only memoize the outer calls and not the recursive calls.

In [49]:

```
let memo_fib = memo fib
```

Out[49]:

```
val memo_fib : int -> int = <fun>
```

In [50]:

```
memo_fib 40
```

Out[50]:

```
- : int = 165580141
```

Tying the recursive knot

This function should remind you of the definition we used for Y combinator.

In [51]:

```
let fib_norec f n = if n < 2 then 1 else f (n-1) + f(n-2)
```

Out[51]:

```
val fib_norec : (int -> int) -> int -> int = <fun>
```

The idea is to provide an `f` which is the memoized version of

```
let rec f n = if n < 2 then 1 else f (n-1) + f(n-2)
```

We will use a **reference** to tie the knot.

Tying the recursive knot

`memo_rec` will memoize recursive function that take an explicit recursive function argument such as `fib_norec`.

In [52]:

```
let memo_rec f_norec =
  (* define a reference [f] to a function which will never be invoked *)
  let f = ref (fun _ -> assert false) in
  (* memoize the "eta-expanded" [f_norec] function by dereferencing [f].
  let f_rec_memo = memo (fun x -> f_norec !f x) in
                                     (* [f] is not dereferenced yet *)
  f := f_rec_memo; (* update [f] to the recursive memoized function *)
  f_rec_memo
```

Out[52]:

```
val memo_rec : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

In [53]:

```
let fib_memo = memo_rec fib_norec
```

Out[53]:

```
val fib_memo : int -> int = <fun>
```

In [54]:

```
fib_memo 30
```

Out[54]:

```
- : int = 1346269
```

Edit distance

- Memoization is a general solution for **dynamic programming.**
- Let's compute **edit distance** (aka **Levenshtein distance**) between two strings.
- Example:
  - edit_distance("kitten","sitting") = 3
  - kitten -> sitten
  - sitten -> sittin
  - sittin -> sitting

Timing the execution

In [55]:

```
(* Returns the execution time of [f v] in milliseconds *)
let time_it f v =
  let s = Unix.gettimeofday() in
  let res = f v in
  let e = Unix.gettimeofday () in
  (res, (e -. s) *. 1000.)
```

Out[55]:

```
val time_it : ('a -> 'b) -> 'a -> 'b * float = <fun>
```

Edit distance

In [56]:

```
let rec edit_distance ?log (s,t) =
  let open String in
  if log = Some true then print_endline (s ^ " " ^ t);
  match String.length s, String.length t with
  | 0,x | x,0 -> x
  | len_s, len_t ->
    let s' = sub s 0 (len_s - 1) in
    let t' = sub t 0 (len_t - 1) in
    List.fold_left (fun acc v -> min acc v) max_int [
      edit_distance ?log (s',t) + 1; (* insert at end of s *)
      edit_distance ?log (s,t') + 1; (* delete from end of s *)
      edit_distance ?log (s',t') +
        if get s (len_s-1) = get t (len_t-1) then 0 else 1
    ]
```

Out[56]:

```
val edit_distance : ?log:bool -> string * string -> int = <
fun>
```

Edit distance

In [57]:

```
time_it (edit_distance ~log:false) ("OCaml", "ocaml")
```

Out[57]:

```
- : int * float = (2, 2.72679328918457031)
```

Edit distance

In [58]:

```
time_it (edit_distance ~log:false) ("OCaml 4.08", "ocaml 4.08")
```

Out[58]:

```
- : int * float = (2, 7467.10491180419922)
```

Memoize edit distance

In [59]:

```
let rec edit_distance_norec ?log f (s,t) =
  let open String in
  if log = Some true then print_endline (s ^ " " ^ t);
  match String.length s, String.length t with
  | 0,x | x,0 -> x
  | len_s, len_t ->
    let s' = sub s 0 (len_s - 1) in
    let t' = sub t 0 (len_t - 1) in
    List.fold_left (fun acc v -> min acc v) max_int [
      f (s',t) + 1; (* insert at end of s *)
      f (s,t') + 1; (* delete from end of s *)
      f (s',t') +
        if get s (len_s-1) = get t (len_t-1) then 0 else 1
    ]
```

Out[59]:

```
val edit_distance_norec :
  ?log:bool -> (string * string -> int) -> string * string
-> int = <fun>
```

Memoize edit distance

In [60]:

```
let memo_edit_distance = memo_rec (edit_distance_norec ~log:false)
```

Out[60]:

```
val memo_edit_distance : string * string -> int = <fun>
```

In [61]:

```
time_it memo_edit_distance ("OCaml 4.08", "ocaml 4.08")
```

Out[61]:

```
- : int * float = (2, 0.500917434692382812)
```

Fin.