Functional Abstractions over Imperative Infrastructure

COS 326
David Walker
Princeton University

– *Abstractions involve using your imagination*

Welcome to the Infinite!

```
module type INFINITE =
  sig
    type ‘a stream                       (* an infinite series of values *)

    val const : ‘a -> ‘a stream          (* an infinite series – all the same *)

    val head : ‘a stream -> ‘a           (* get the next value – there always is one! *)
    val tail : ‘a stream -> ‘a stream    (* get all the rest *)

    val map : (‘a -> ‘b) -> ‘a stream -> ‘b stream


    ...
end

module Inf : INFINITE = ... ?
```

Welcome to the Infinite!

– *Demo!*

Consider this definition:

```
type ‘a stream =
  Cons of ‘a * (‘a stream)
```

We can write functions to extract the head and tail of a stream:

```
let head(s:’a stream):’a =
  match s with
  | Cons (h,_) -> h

let tail(s:’a stream):’a stream =
  match s with
  | Cons (_,t) -> t
```

But there's a problem…

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

How do I build a value of type 'a stream?

attempt: Cons (3, ______) .... Cons (3, Cons (4, ___))

There doesn't seem to be a base case (e.g., Nil)

Since we need a stream to build a stream,
what can we do to get started?

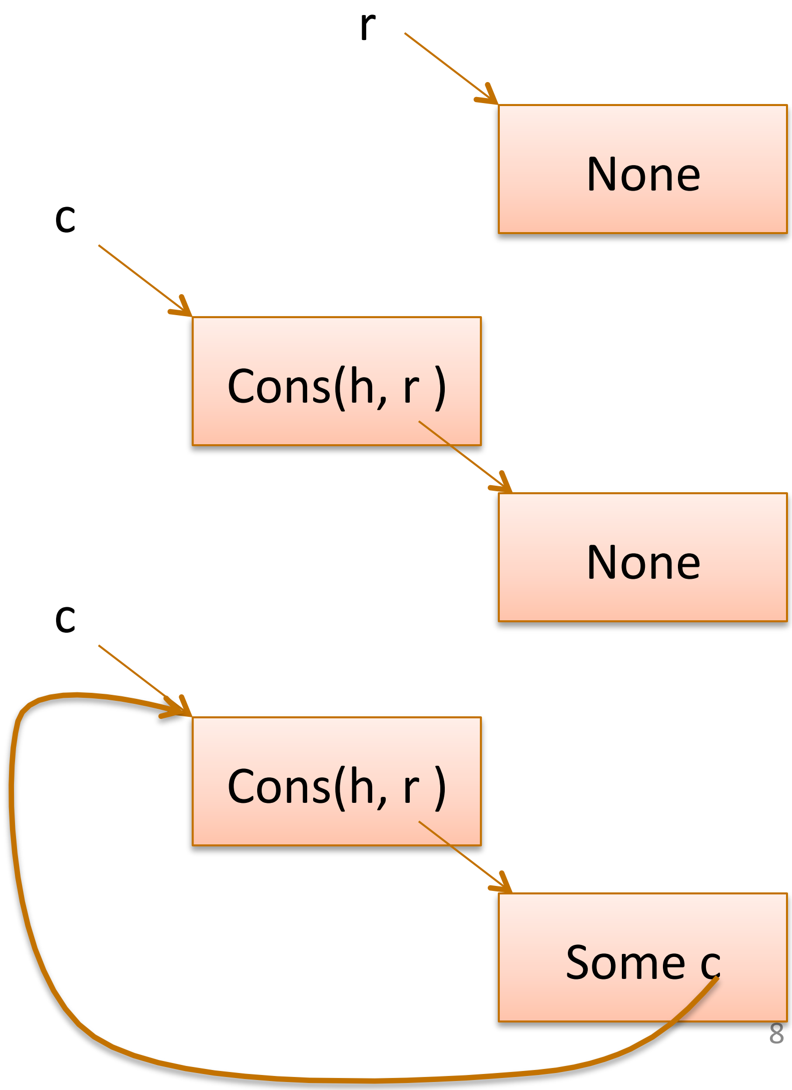
One idea

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

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

What happens?
```

```
# let rec ones = Cons(1,ones);;
val ones : int stream =
 Cons (1,
  Cons (1,
   Cons (1,
    Cons (1, ...
))))
# ^CInterrupted
```

An alternative would be to use refs

```
type 'a stream =
  Cons of 'a * ('a stream) option ref

let circular_cons h =
  let r = ref None in
  let c = Cons(h,r) in
  (r := (Some c); c)
```



This works …

but has a serious drawback

An alternative would be to use refs

```
type 'a stream =
  Cons of 'a * ('a stream) option ref

let circular_cons h =
  let r = ref None in
  let c = Cons(h,r) in
  (r := (Some c); c)
```

This works .... but has a serious drawback...

when we try to get out the tail, it may not exist.

Back to our earlier idea

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

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

```
# let rec ones = Cons(1,ones);;
val ones : int stream =
 Cons (1,
  Cons (1,
   Cons (1,
    Cons (1, ...
))))
# ^CInterrupted
```

The only "problem" here is that ML evaluates our code just a little bit too *eagerly*. We want it to "wait" to evaluate the right-hand side only when necessary ...

Back to our earlier idea

One way to implement "waiting" is to wrap a computation up in a function and then call that function later when we want to.

Another attempt:

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

let rec ones =
  fun () -> Cons(1,ones)

let head (x) =
  match x () with
    Cons (hd, tail) -> hd
;;

head (ones);;
```

Darn. Doesn't type check!
It's a function with type
unit -> int stream
not a stream

Lazy Evaluation

What if we changed the definition of streams?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

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

Or, the way we’d normally write it:

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

Lazy Evaluation

How would we define head, tail, and map?

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

Lazy Evaluation

How would we define head, tail, and map?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let head(s:’a stream):’a =
```

Lazy Evaluation

How would we define head, tail, and map?

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

let head(s:'a stream):'a =
 match s() with
 | Cons(h,_) -> h
```

Lazy Evaluation

How would we define head, tail, and map?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let head(s:’a stream):’a =
 match s() with
 | Cons(h,_) -> h

let tail(s:’a stream):’a stream =
 match s() with
 | Cons(_,t) -> t
```

Lazy Evaluation

How would we define head, tail, and map?

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

let rec map (f:'a->'b) (s:'a stream) : 'b stream =
```

Lazy Evaluation

How would we define head, tail, and map?

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

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

Lazy Evaluation

How would we define head, tail, and map?

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str

let rec map (f:’a->’b) (s:’a stream) : ’b stream =
  fun () -> Cons(f (head s), map f (tail s))
```

Importantly, map returns a function which delays evaluating the recursive call to map.

Lazy Evaluation

Now we can use map to build other infinite streams:

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

let rec ones = fun () -> Cons(1,ones) ;;
let inc x = x + 1
let twos = map inc ones ;;
```

head twos
--> head (map inc ones)
--> head (fun () -> Cons (inc (head ones), map inc (tail ones)))
--> match (fun () -> ...) () with Cons (hd, _) -> h
--> match Cons (inc (head ones), map inc (tail ones)) with Cons (hd, _) -> h
--> match Cons (inc (head ones), fun () -> ...) with Cons (hd, _) -> h
--> ... --> 2

Another combinator for streams:

```
let rec zip f s1 s2 =
  fun () ->
   Cons(f (head s1) (head s2),
        map f (tail s1) (tail s2)) ;;

let threes = zip (+) ones twos ;;

let rec fibs =
  fun () ->
   Cons(0, fun () ->
           Cons (1,
                 zip (+) fibs (tail fibs)))
```

Unfortunately

This is not very efficient:

```
type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = unit -> ‘a str
```

Every time we want to look at a stream (e.g., to get the head or tail), we have to re-run the function.

So when you ask for the 10^{th} fib and then the 11^{th} fib, we are re-calculating the fibs starting from 0, when we could *cache* or *memoize* the result of previous fibs.

Memoizing Streams

We can take advantage of refs to memoize:

```
type 'a thunk =
  Unevaluated of unit -> 'a | Evaluated of 'a

type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) thunk ref
```

When we build a stream, we use an Unevaluated thunk to be lazy. But when we ask for the head or tail, we remember what Cons-cell we get out and save it to be re-used in the future.

Memoizing Streams

```
type 'a thunk =
  Unevalu   ed of uni            luated  of 'a ;;
type 'a la         ('

typ
and 'a

le
  matc
  |                          > h
  | Uneval          f
    (s :=    aluated (f()); head s) ;;
```

Common pattern!

Dereference & check if evaluated:

- If so, take the value.
- If not, evaluate it & take the value

Memoizing Streams

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

type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) lazy_t;;

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

let head(s:'a stream):'a =
  match force s with
  | Cons(h,_) -> h ;;

let tail(s:'a stream):'a =
  match force s with
  | Cons(_,t) -> t ;;
```

Memoizing Streams

```
type ‘a thunk =
  Unevaluated of unit -> ‘a | Evaluated of ‘a

type ‘a str = Cons of ‘a * (‘a stream)
and ‘a stream = (‘a str) thunk ref;;

let rec ones =
  ref (Unevaluated (fun () => Cons(1,ones))) ;;
```

Memoizing Streams

```
type 'a thunk =
  Unevaluated of unit -> 'a | Evaluated of 'a

type 'a str = Cons of 'a * ('a stream)
and 'a stream = ('a str) thunk ref;;

let thunk f = ref (Unevaluated f)

let rec ones =
  thunk (fun () => Cons(1,ones))
```

OCaml's Builtin Lazy Constructor

If you use Ocaml's built-in lazy_t, then you can write:

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

and this takes care of wrapping a "ref (Unevaluated (fun () => ...))" around the whole thing.

So for example:

```
let rec fibs =
  lazy (Cons(0,
        lazy (Cons(1,zip (+) fibs (tail fibs)))))
```

A note on laziness

- By default, Ocaml is an eager language, but you can use the "lazy" features to build lazy datatypes.

- Other functional languages, notably Haskell, are lazy by default. *Everything* is delayed until you ask for it.
  - generally much more pleasant to do programming with infinite data.
  - but harder to reason about space and time.
  - and has bad interactions with side-effects.

- The basic idea of laziness gets used a lot:
  - e.g., Unix pipes, TCP sockets, etc.

Summary

You can build *infinite data structures*.

- Not really infinite – represented using cyclic data and/or lazy evaluation.

Lazy evaluation is a useful technique for delaying computation until it's needed.

- Can model using just functions.
- But behind the scenes, we are *memoizing* (caching) results using refs.

This allows us to separate model generation from evaluation to get "scale-free" programming.

- e.g., we can write down the routine for calculating pi regardless of the number of bits of precision we want.
- Other examples: geometric models for graphics (procedural rendering); search spaces for AI and game theory (e.g., tree of moves and counter-moves).

End

More fun with streams:

```
let rec filter p s =
    if p (head s) then
      lazy (Cons (head s,
                  filter p (tail s)))
    else (filter p (tail s))
  ;;

let even x = (x mod 2) = 0;;
let odd x = not(even x);;

let evens = filter even nats ;;
let odds = filter odd nats ;;
```

Sieve of Eratosthenes

```
let not_div_by n m =
    not (m mod n = 0) ;;

let rec sieve s =
  lazy (Cons (head s,
              sieve (filter (not_div_by (head s))
  (tail s))))
  ;;


let primes = sieve (tail (tail nats)) ;;
```

Taylor Series

```
let rec fact n = if n <= 0 then 1 else n * (fact
   (n-1)) ;;

let f_ones = map float_of_int ones ;;

(* The following series corresponds to the Taylor
 * expansion of e:
 *   1/1! + 1/2! + 1/3! + ...
 * So you can just pull the floats off and start
   adding
 * them up. *)
let e_series =
   zip (/.) f_ones (map float_of_int (map fact
   nats)) ;;

let e_up_to n =
    List.fold_left (+.) 0. (first n e_series) ;;
```

Pi

```
(* pi is approximated by the Taylor series:
 *   4/1 - 4/3 + 4/5 - 4/7 + ...
 *)
let rec alt_fours =
  lazy (Cons (4.0,
  lazy (Cons (-4.0, alt_fours))));;

let pi_series = zip (/.) alt_fours (map
  float_of_int odds);;

let pi_up_to n =
  List.fold_left (+.) 0.0
      (first n pi_series) ;;
```

Integration to arbitrary precision…

```
let approx_area (f:float->float)(a:float)(b:float) =
    (((f a) +. (f b)) *. (b -. a)) /. 2.0 ;;

let mid a b = (a +. b) /. 2.0 ;;

let rec integrate f a b =
  lazy (Cons (approx_area f a b,
              zip (+.) (integrate f a (mid a b))
                       (integrate f (mid a b) b))) ;;

let rec within eps s =
    let (h,t) = (head s, tail s) in
    if abs(h -. (head t)) < eps then h else within eps t ;;

let integral f a b eps = within eps (integrate f a b) ;;
```

Thought Exercises

- Do other Taylor series using streams:
  - e.g., $\cos(x) = 1 - (x^2/2!) + (x^4/4!) - (x^6/6!) + (x^8/8!) \ldots$

- You can model a wire as a stream of booleans and a combinational circuit as a stream transformer.
  - define the "not" circuit which takes a stream of booleans and produces a stream where each value is the negation of the values in the input stream.
  - define the "and" and "or" circuits which take streams of booleans and produce a stream of the logical-and/logical-or of the input values.
  - better: define the "nor" circuit and show how "not", "and", and "or" can be defined in terms of "nor".
  - For those of you in EE: define a JK-flip-flop

- How would you define infinite trees?

END