Lazy Evaluation & Infinite Data

COS 326
Speaker: Andrew Appel
Princeton University

Some ideas in this lecture borrowed from Brigitte Pientka, McGill University

slides copyright 2020 David Walker and Andrew Appel
permission granted to reuse these slides for non-commercial educational purposes
AN INFINITE DATA STRUCTURE: STREAMS
Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...
Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...
Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...
Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...
Consider this definition:

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

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

```ocaml
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...

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

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

Cons (3, Cons (4, ___))  
Cons (3, ___)
But there’s a problem...

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

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

```
Cons (3, Cons (4, ___))
```

```
Cons (3, ___)
```

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?
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

```ocaml
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)
```

.... 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's look at creating the stream of all natural numbers:

```
let rec nats i = Cons(i, nats (i+1))
```

# let n = nats 0;;
Stack overflow during evaluation (looping recursion?).

OCaml evaluates our code just a little bit too **eagerly**. We want to evaluate the right-hand side **only when necessary** ...
Another 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:

```ocaml
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
```

Are there any problems with this code?

Darn. Doesn’t type check! It’s a function with type `unit -> int stream` not just `int stream`
What if we changed the definition of streams one more time?

```ocaml
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:

```ocaml
let rec ones () = Cons(1,ones)
```
How would we define head, tail, and map of an 'a stream?

```ocaml
let rec head = function
  | Cons of _, _ -> head (tail (Cons of _))
  | _ -> None

let rec tail = function
  | Cons of _ -> head
  | _ -> None

let rec map = function
  | Cons of a, b -> Cons of map a, map b
  | _ -> None
```

- `head`: Returns the first element of the stream.
- `tail`: Returns the rest of the stream.
- `map`: Applies a function to each element of the stream.

### Additional Note
Adding `Cons` is necessary to handle the case where the stream is empty, ensuring that the functions work correctly even when passed an empty stream.
Functional Implementation

How would we define head, tail, and map of an ’a stream?

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

let head(s:’a stream):’a = ...
```
How would we define head, tail, and map of an 'a stream?

```ml
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
```
How would we define head, tail, and map of an 'a stream?

```ocaml
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 =
  ...
```
How would we define head, tail, and map of an 'a stream?

```ocaml
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
```
Functional Implementation

How would we define head, tail, and map of an ’a stream?

```
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 = ...
```
How would we define head, tail, and map of an 'a stream?

```ocaml
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))
```
How would we define head, tail, and map of an 'a stream?

```ocaml
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))
```

Rats!
Infinite looping!
How would we define head, tail, and map of an ’a stream?

```
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))
```

Doesn’t type check!
Cons (x,y) is a str not a stream
How would we define head, tail, and map of an 'a stream?

```ocaml
type 'a str = Cons of 'a * ('a str)
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 must return a function, which delays evaluating the recursive call to map.
How would we define head, tail, and map of an ’a stream?

```ocaml
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
```
Functional Implementation

**Type Definition**

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

**Recursive Functions**

```ml
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 twos = map (fun x -> x+1) ones
```

**Expression Evaluation**

```ml
head twos
---> head (map inc ones)
---> head (fun () -> Cons (inc (head ones), map inc (tail ones)))
---> match (fun () -> ...) () with Cons (h, _) -> h
---> match Cons (inc (head ones), map inc (tail ones)) with Cons (h, _) -> h
---> match Cons (inc (head ones), fun () -> ...) with Cons (h, _) -> h
---> ... ---> 2
```
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str

let rec zip f s1 s2 = 
  fun () -> 
    Cons(f (head s1) (head s2),
        zip f (tail s1) (tail s2))
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str

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

let threes = zip (+) ones twos
Functional Implementation

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

let rec zip f s1 s2 =
  fun () ->
    Cons(f (head s1) (head s2),
         zip 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:

```haskell
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.
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.

```
let x = head s
let y = head s
```

rerun the *entire* underlying function as opposed to fetching the first element of a list.

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

This is really, really inefficient:

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

So when you ask for the 10\textsuperscript{th} fib and then the 11\textsuperscript{th} fib, we are recalculating the fibs starting from 0...

It takes exponential time, the same way this function does:

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

If we could \textit{cache} or \textit{memoize} the result of previous fibs...
LAZY EVALUATION
Lazy Data

We can take advantage of mutation to memoize:

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

type 'a lazy_t = 'a thunk ref
```

Initially:

- Unevaluated

After evaluating once:

- Evaluated

fun x -> ....
Lazy Data

We can take advantage of mutation to memoize:

```ocaml
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
```
Lazy Data

```ml
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 head(s:'a stream):'a =
```
type 'a thunk =
    Unevaluated of (unit -> 'a) | Evaluated of 'a

type 'a lazy_t = 'a thunk ref

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

let rec head(s:'a a stream):'a =
    match !s with
    | Evaluated (Cons(h,_)) ->
    | Unevaluated f ->
Lazy Data

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

type 'a lazy_t = 'a thunk ref

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

let rec head(s:'a a stream):'a =
  match !s with
  | Evaluated (Cons(h,_)) -> h
  | Unevaluated f ->
Lazy Data

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 head(s:'a stream):'a =
   match !s with
   | Evaluated (Cons(h,_)) -> h
   | Unevaluated f ->
      let x = f() in (s := Evaluated x; head s)
type 'a thunk =
    Unevaluated of (unit -> 'a) | Evaluated of 'a

let rec tail(s:'a stream) : 'a stream =
    match !s with
    | Evaluated (Cons(_,t)) -> t
    | Unevaluated f ->
        (let x = f () in s := Evaluated x; tail s)
Lazy Data

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 tail(s:'a stream) : 'a stream =
  match !s with
  | Evaluated (Cons(_,t)) -> t
  | Unevaluated f ->
    let x = f() in (s := Evaluated x; tail s)

let rec head(s:'a stream):'a =
  match !s with
  | Evaluated (Cons(h,_)) -> h
  | Unevaluated f ->
    let x = f() in (s := Evaluated x; head s)
Lazy Data

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

type 'a lazy_t = 'a thunk

and 'a stream = Cons of 'a * ('a stream)

let rec tail (s: 'a stream) : 'a stream =
  match !s with
  | Evaluated (Cons(_,t)) -> t
  | Unevaluated f ->
    let x = f() in (s := Evaluated x; tail s)

let rec head (s: 'a stream): 'a =
  match !s with
  | Evaluated (Cons(h, _)) -> h
  | Unevaluated f ->
    let x = f() in (s := Evaluated x; head s)

Common pattern!

Dereference & check if evaluated:
• If so, take the value.
• If not, evaluate it & take the value
```
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 ->
    let v = f() in
    (t:= Evaluated v ; v)

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

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

```ocaml
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 lazy f = ref (Unevaluated f)

let rec ones =
    lazy (fun () -> Cons(1,ones))
```
What’s the interface?

type ’a lazy_t

val lazy : (unit -> ’a) -> ’a lazy_t

val force : ’a lazy_t -> ’a

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

let rec ones =
  lazy(fun () -> Cons(1,ones))
What’s the interface?

defined

```ocaml
type 'a lazy_t

val lazy : (unit -> 'a) -> 'a lazy_t

val force : 'a lazy_t -> 'a

let rec zip f s1 s2 = lazy (fun () ->
  match force s1, force s2 with
  Cons (x1,r1), Cons (x2,r2) -> Cons (f x1 x2,
    zip f r1 r2)
)

let type 'a str = Cons of 'a * ('a stream)

and 'a stream = ('a str) lazy_t
```
OCaml’s Builtin Lazy Constructor

If you use Ocaml’s built-in Lazy.t, then you can write:

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

and this takes care of wrapping a “ref (Unevaluated (fun () -> ...))” around the whole thing. It has the effect of suspending the computation until you use Lazy.force

So for example:

```ocaml
let rec fibs =
    lazy (Cons(0,
               lazy (Cons(1,
                         zip (+) fibs (tail fibs))))))
```
type 'a str = Cons of 'a * 'a stream
and 'a stream = ('a str) Lazy.t;;

let rec zip f (s1: 'a stream) (s2: 'a stream) : 'a stream =
  lazy (match Lazy.force s1, Lazy.force s2 with
    Cons (x1,r1), Cons (x2,r2) ->
      Cons (f x1 x2, zip f r1 r2))

let tail (s: 'a stream) : 'a stream =
  match Lazy.force s with Cons (x,r) -> r

let rec fibs : int stream =
  lazy (Cons(0, lazy (Cons (1, zip (+) fibs (tail fibs))))));;

let rec printn n s =
  if n>0 then
    match Lazy.force s with
      Cons (x,r) -> (printf "%d\n" x; printn (n-1) r)

let _ = printn 10 fibs
EVALUATION ORDER:
CALL-BY-VALUE VS
CALL-BY-NAME VS
LAZY (a.k.a. CALL-BY-NEED)
OCaml is Call-by-value

let x = e1 in e2

Evaluation strategy:
• evaluate e1 until you get a value
• bind that value to x
• evaluate e2 until you get a value

Example
let x = 2 + 3 in x - 7
--> let x = 5 in x - 7
--> 5 - 7
--> -2
OCaml is Call-by-value

let x = e1 in e2

Evaluation strategy:
• evaluate e1 until you get a value
• bind that value to x
• evaluate e2 until you get a value

e1 e2

Evaluation strategy:
• evaluate e1 until you get a value (fun x -> e)
• evaluate e2 until you get a value (v)
• substitute v for x in e to get e’
• continue evaluating e’ until you get a value
OCaml is Call-by-value

let x = e1 in e2

Evaluation strategy:
• evaluate e1 until you get a value
• bind that value to x
• evaluate e2 until you get a value

e1 e2

Evaluation strategy:
• evaluate e1 until you get a value (fun x -> e)
• evaluate e2 until you get a value (v)
• substitute v for x in e to get e’
• continue evaluating e’ until you get a value

Is this the only way we could evaluate these expressions?
Is this the most efficient way we could evaluate these expressions?
OCaml is Call-by-value

Evaluation strategy:
• evaluate e1 until you get a value
• bind that value to x
• evaluate e2 until you get a value

Evaluation strategy:
• evaluate e1 until you get a value (fun x -> e)
• evaluate e2 until you get a value (v)
• substitute v for x in e to get e’
• continue evaluating e’ until you get a value

Is this the only way we could evaluate these expressions? **No!**
Is this the most efficient way we could evaluate these expressions? **No!**
Call-by-Name

let x = e1 in e2

Evaluation strategy:
• bind that expression e1 to x
• continue to evaluate e2

Example

let x = 2 + 3 in x - 7
--> (2 + 3) - 7
--> 5 - 7
--> -2
**Call-by-Name**

Evaluation strategy:
- bind that expression `e1` to `x`
- continue to evaluate `e2`

```
let x = e1 in e2
```

Call-by-name can avoid work sometimes:
```
let x = work () in 7
--> 7
```
Call-by-Name

Evaluation strategy:
• bind that expression e₁ to x
• continue to evaluate e₂

let x = e₁ in e₂

Call-by-name can avoid *A LOT* of work sometimes:

```
let x = loop_forever () in 7
```

--> 7
Call-by-Name

let x = e1 in e2

Evaluation strategy:
• bind that expression e1 to x
• continue to evaluate e2

But sometimes it does more work than necessary

let x = work () in x + x
---> (work ()) + (work ())
In general:
CBV can be asymptotically faster than CBN (by exponential factor at least!)
CBN can be asymptotically faster than CBV (by exponential factor at least!)

However:
CBV can diverge (infinite-loop) where CBN terminates but not vice versa!
If CBN diverges, then ANY strategy diverges

Therefore:
CBN is the “most general” strategy, in the sense that it terminates as often as possible. Though it definitely isn’t necessarily fastest!

by the way, guess who figured all this out:
Alonzo Church and his graduate students, Princeton University, 1930s
Lazy evaluation is like call-by-name but it avoids repeatedly executing e₁ by using *memoization* – it computes an answer once and then remembers the result if x is ever needed a 2nd or 3rd time.

The operational semantics notation is less compact when it comes to describing lazy computations because we have to keep track of the imperative state used for memoization. So I won’t try here.
In general:

**LAZY** can be asymptotically faster than **CBN**.

- thanks to memoization – no repeated calls

**CBN** is never asymptotically faster than **LAZY**.

**CBN** terminates if-and-only-if **LAZY** terminates.

(Thus) **LAZY** is also a most-general strategy.

In practice:

- Data structures used to memoize computations take up space
  - thunks hang on to data structures, making it tough to reason about

- Much optimization needed for **CBN** to approach **CBV** performance

- But laziness (“deferred, call-by-need computation”) can be useful
  - we can program with selective laziness in call-by-value languages
Historical note

The term “thunk” was introduced in 1960 to describe a way of implementing call-by-name function-parameters in Algol 60, by P. Z. Ingerman of U. Penn.

An early form of “closures” (for environments of thunks) was invented by: Ned Irons ’58 of Princeton University and Wally Feurzig of U. Chicago.

Ned Irons was one of the most brilliant programmers of his time.
Wally Feurzig was a pioneer in A.I. and in C.S. education, a collaborator of Turing award winners Marvin Minsky (PhD Princeton 1951) and John McCarthy (PhD Princeton 1951).

Summary

By default, OCaml (and Java, C, etc) is an eager language
- but you can use thunks or “lazy” to suspend computations
- use “force” to run the computation when needed

By default, Haskell is a lazy language
- the implementers (eg: Simon Peyton Jones) would probably make it eager by default if they had a do-over
- working with infinite data is generally more pleasant
- but difficult to reason about space and time

Lazy evaluation makes it possible to build *infinite data structures*.
- can be modelled using functions
- but adding refs allows memoization