Functional Abstractions
over Imperative Infrastructure

and

Lazy Evaluation

COS 326

Andrew W. Appel
Princeton University

slides copyright 2013-2015 David Walker and Andrew W. Appel
– Abstractions involve using your imagination
module type INFINITE =
  sig
    type ‘a stream (* an infinite series of values *)
    val const : ‘a -> ‘a stream (* an infinite series – all the same *)
    val nats : () -> int stream (* all of the natural numbers *)
    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!

– How would you implement an infinite data structure?
  (think about it)
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…

```ml
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?
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,
          Cons (1,
            Cons (1, ...
))))))
#
One idea

```ocaml
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,
          Cons (1,
            Cons (1,
              Cons (1,
              Cons (1,
                ...

OCaml builds this!
```
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)
```

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)
```

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

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

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:

```plaintext
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 just int stream
What if we changed the definition of streams one more time?

```ml
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:

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

What we had before.

Augmented as a *mutually recursive* type definition.
How would we define head, tail, and map?

```ocaml
type 'a str = Cons of 'a * ('a stream)
and 'a stream = unit -> 'a str
```
How would we define head, tail, and map?

```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?

```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
```
How would we define head, tail, and map?

```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
```
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 =
How would we define head, tail, and map?

```plaintext
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))  
```
Call-by-name Evaluation

How would we define head, tail, and map?

```ocaml
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?

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

But we don’t infinite loop, because the typechecker saves us: Cons (x,y) is a str not a stream.
How would we define head, tail, and map?

```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 =
  fun () -> Cons(f (head s), map f (tail s))
```

Importantly, map must return a function, which delays evaluating the recursive call to map.
Now we can use map to build other infinite streams:

```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 ;;
```

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:

```ocaml
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:

```ocaml
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\textsuperscript{th} fib and then the 11\textsuperscript{th} fib, we are re-calculating the fibs starting from 0, when we could \textit{cache} or \textit{memoize} the result of previous fibs.
From call-by-name evaluation to

LAZY EVALUATION
Memoizing Streams

We can take advantage of refs to memoize:

```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
```

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.
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; x)
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 ->
        (s := Evaluated (f()); tail s) ;;
type 'a thunk =
  Unevaluated of (unit -> 'a) | Evaluated of 'a

type 'a lazy_t = ('a thunk) ref

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

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

define 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

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

let thunk f = ref (Unevaluated f)

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

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

What’s the interface?

type 'a lazy = 'a thunk ref

val thunk : (unit -> 'a) -> 'a lazy = ...

val force: 'a lazy -> 'a = ...

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

and 'a stream = ('a str) lazy

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:

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

and this takes care of wrapping a “ref (Unevaluated (fun () -> ...))” around the whole thing.

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 g n s =
  if n>0 then
    match Lazy.force s with Cons (x,r) ->
    (print_int x; print_string "\n"; g (n-1) r)
  else ();;

g 10 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).
Mathematical background: $\lambda$-calculus

Notation: use $(\lambda x. E)$ instead of $(\text{fun } x \to E)$

Rules:

$$(\lambda x. A) B \Rightarrow A[B/x] \quad \text{(\(\beta\)-reduction)}$$

\[
\begin{align*}
A & \Rightarrow A' \\
A B & \Rightarrow A' B \\
B & \Rightarrow B' \\
A B & \Rightarrow A B'
\end{align*}
\]

(context rules)

\[
(\lambda x. A) \Rightarrow (\lambda x. A')
\]

$2*3 \Rightarrow 5 \quad \text{(\(\delta\)-reduction)}$
Mathematical background: \( \lambda \)-calculus

\[
(\lambda \, x \, . \, A) \, B \quad \mapsto \quad A[B/x]
\]

\[
2 \times 3 \quad \mapsto \quad 5
\]

\[
\begin{align*}
A & \quad \mapsto \quad A' \\
\frac{A \, B}{A \, B} & \quad \mapsto \quad A' \, B \\
\frac{B}{B} & \quad \mapsto \quad B'
\end{align*}
\]

a legal reduction sequence
\[(\lambda \, x \, . \, (\lambda \, y \, . \, f \, (f \, y)) \, (x+1)) \, (2 \times 3) \quad \mapsto \quad (\lambda \, x \, . \, f \, (f \, (x+1))) \, (2 \times 3) \quad \mapsto \quad f(f(2 \times 3 + 1) \mapsto f(f(5+1) \mapsto f(f \, 6))
\]

call-by-value reduction
\[(\lambda \, x \, . \, (\lambda \, y \, . \, f \, (f \, y)) \, (x+1)) \, (2 \times 3) \quad \mapsto \quad (\lambda \, x \, . \, (\lambda \, y \, . \, f \, (f \, y)) \, (x+1)) \, 5 \mapsto (\lambda \, y \, . \, f \, (f \, y)) \, (5+1)) \mapsto (\lambda \, y \, . \, f \, (f \, y)) \, 6 \mapsto f \, (f \, 6)
\]

call-by-name reduction
\[(\lambda \, x \, . \, (\lambda \, y \, . \, f \, (f \, y)) \, (x+1)) \, (2 \times 3) \quad \mapsto \quad (\lambda \, y \, . \, f \, (f \, y)) \, ((2 \times 3) + 1) \mapsto f \, (f \, ((2 \times 3) + 1)) \mapsto f \, (f \, (5+1)) \mapsto f \, (f \, 6)
\]

Church-Rosser theorem (1934):
No matter which reduction order you use, you’ll get to the same answer.
Call-by-name, call-by-value, lazy evaluation

call-by-value reduction
\[
(\lambda x. (\lambda y. f (f y)) (x+1)) (2*3) \Rightarrow (\lambda x. (\lambda y. f (f y)) (x+1)) 5 \Rightarrow \\
(\lambda y. f (f y)) (5+1)) \Rightarrow (\lambda y. f (f y)) 6 \Rightarrow f (f 6)
\]

(like ordinary ML)

call-by-name reduction
\[
(\lambda x. (\lambda y. f (f y)) (x+1)) (2*3) \Rightarrow (\lambda y. f (f y)) ((2*3)+1) \Rightarrow f (f ((2*3)+1)) \\
\Rightarrow f (f (5+1)) \Rightarrow f (f 6)
\]

(like streams WITHOUT thunks)

lazy evaluation: (using thunks, updated with “memorized” computed values)
To represent this, you can’t just use textual strings, you need pointers.
No wonder nobody thought of it until AFTER computers were invented.
Consider this lambda-term:

\[(\lambda y. A )\ (\lambda x. x)\ 3\]  \hspace{1cm} \text{where } A \text{ is some expression}

Reducing \((\lambda x. x)\ 3\) takes one step, but pretend that it takes many steps (i.e., is expensive).

**WHICH IS BETTER?**

**Call-by-value:**

\[(\lambda y. A )\ (\lambda x. x)\ 3\]  \rightarrow  \ (\lambda y. A )\ 3\  \rightarrow  A[3/y] \rightarrow \ldots \rightarrow \ldots

**Call-by-name:**

\[(\lambda y. A )\ (\lambda x. x)\ 3\]  \rightarrow  A[((\lambda x. x)\ 3)/y] \rightarrow \ldots \rightarrow \ldots
WHICH IS BETTER?

Depends! if \( A == (y+y) \), then:

CBV, 3 steps:

\[
(\lambda y. y+y)((\lambda x. x) 3) \rightarrow (\lambda y. y+y) 3 \rightarrow 3+3 \rightarrow 6.
\]

CBN, 4 steps:

\[
(\lambda y. A)((\lambda x. x) 3) \rightarrow ((\lambda x. x) 3)+((\lambda x. x) 3) \rightarrow 3+((\lambda x. x) 3) \rightarrow 3+3 \rightarrow 6.
\]

Depends! if \( A == 4 \), then:

CBV, 2 steps:

\[
(\lambda y. 4)((\lambda x. x) 3) \rightarrow (\lambda y. 4) 3 \rightarrow 4.
\]

CBN, 1 step:

\[
(\lambda y. 4)((\lambda x. x) 3) \rightarrow 3.
\]
Call-by-name vs. call-by-value

WHICH IS BETTER?

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 (which doesn’t mean it’s always fastest).
In general:
LAZY can be asymptotically faster than CBN.
CBN is never asymptotically faster than LAZY.
CBN terminates if-and-only-iff LAZY terminates.
(Thus) LAZY is also a most-general strategy.

However:
It’s hard to express LAZY using the lambda-notation as on the previous slides, because it’s inherently about pointer-sharing (DAGs representing common subexpressions), which is hard to represent in textual lambda calculus.
END