

A Lazy Fix!

COS 326

David Walker

CALL-BY-VALUE VS CALL-BY-NAME

Recall Single Step Operational Semantics

$$(\text{rec } f \ x = e) \ v2 \ \rightarrow \ e[\text{rec } f \ x = e/f][v2/x]$$

you can only call a function
if you have reduced its
argument to a value first

"call by value" semantics

$$\frac{e2 \rightarrow e2'}{v1 \ e2 \rightarrow v1 \ e2'}$$

$$\frac{e1 \rightarrow e1'}{e1 \ e2 \rightarrow e1' \ e2}$$

Other rules explain how
to reduce the function
and its argument to values

A different choice

$$\frac{}{(\text{rec } f \ x = e) \ e2 \ \rightarrow \ e[\text{rec } f \ x = e/f][e2/x]}$$

you can call a function
even if its argument is not
a value

"call by name" semantics

$$\frac{e1 \rightarrow e1'}{e1 \ e2 \rightarrow e1' \ e2}$$

Other rules explain how
to reduce the function,
but not its argument to values

Call by name can be cheap

```
let f x = 3  
let _ = f (expensive_computation)
```

if you never use your
argument, call-by-name
never evaluates it.

Saves time!

But functions usually use their arguments ...

```
let f x = x + x + x + x  
  
let _ = f (expensive)
```

--> (call-by-name)

```
let _ = (expensive) + (expensive) +  
        (expensive) + (expensive)
```

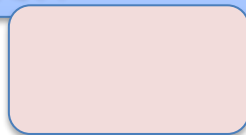
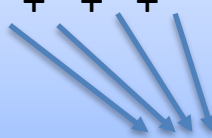
if you use your argument
many times, you wind up
recomputing the expensive
thing each time

Lazy computation

```
let f x = x + x + x + x  
let _ = f (expensive)
```

--> (lazy)

```
let _ = + + +
```



type 'a t = Done of 'a | Compute of unit -> 'a
type 'a lazy_t = ('a t) ref

does not compute the
answer right away.
When used, computes the
answer once and saves the
result so that future uses
are just a lookup

OCAML SUPPORT

Lazy Programming

```
type 'a lazy_t
```

```
lazy : 'a -> 'a lazy_t
```

```
force : 'a lazy_t -> 'a
```

```
open Lazy
```

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

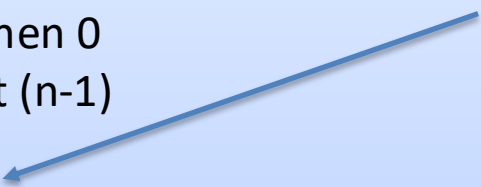
```
let x = lazy (count 10000000)
```

```
let a = (fun x -> 0) x (* fast *)
```

```
let b = (fun x -> force x) (* slow *)
```

```
let c = (fun x -> force x + force x) (* no slower than b *)
```

not a normal call-by-value
function call
ie: requires compiler
support to implement



USEFUL APPLICATIONS: INFINITE DATA STRUCTURES

Streams

Sometimes it is useful to define the entirety of an infinite data set *now*
and sample finite parts of it *later* ...

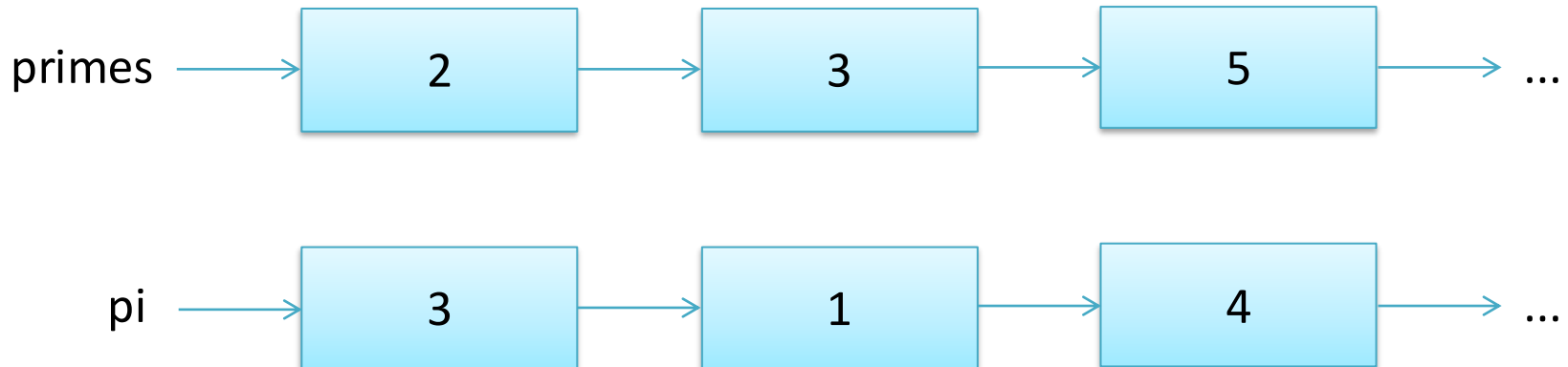
Streams

Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...



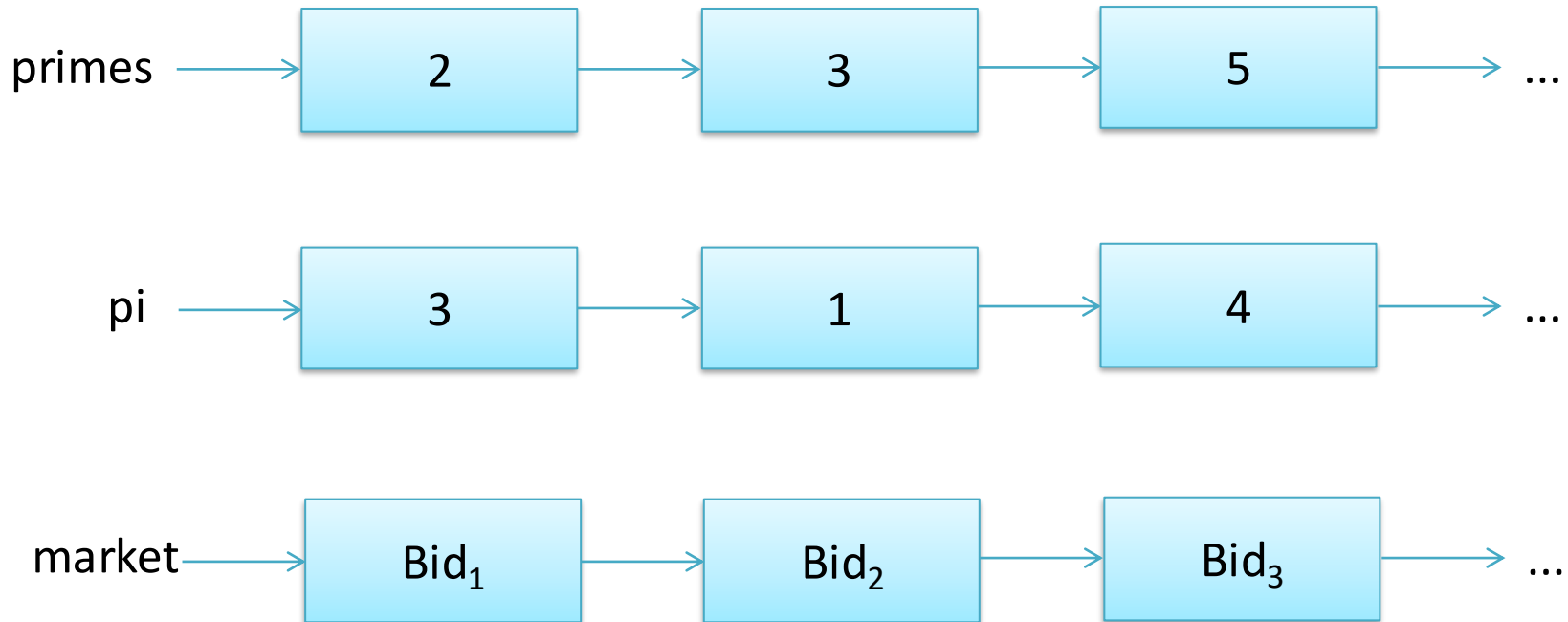
Streams

Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...



Streams

Sometimes it is useful to define the entirety of an infinite data set *now* and sample finite parts of it *later* ...



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

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

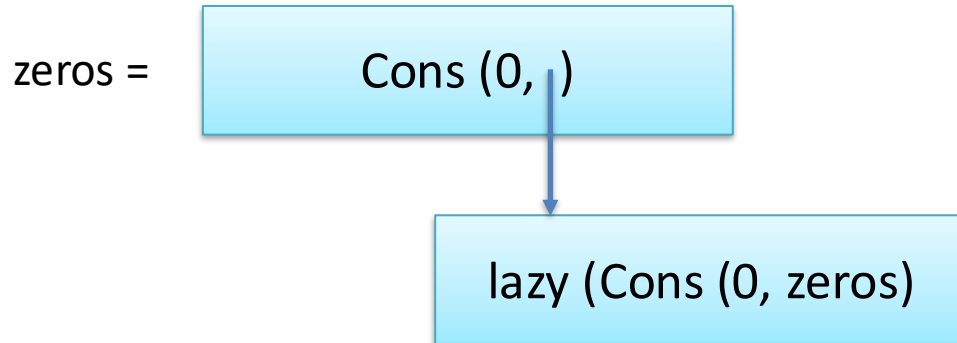
let rec zeros = lazy (Cons (0, zeros))
```

zeros =

lazy (Cons (0, zeros))

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

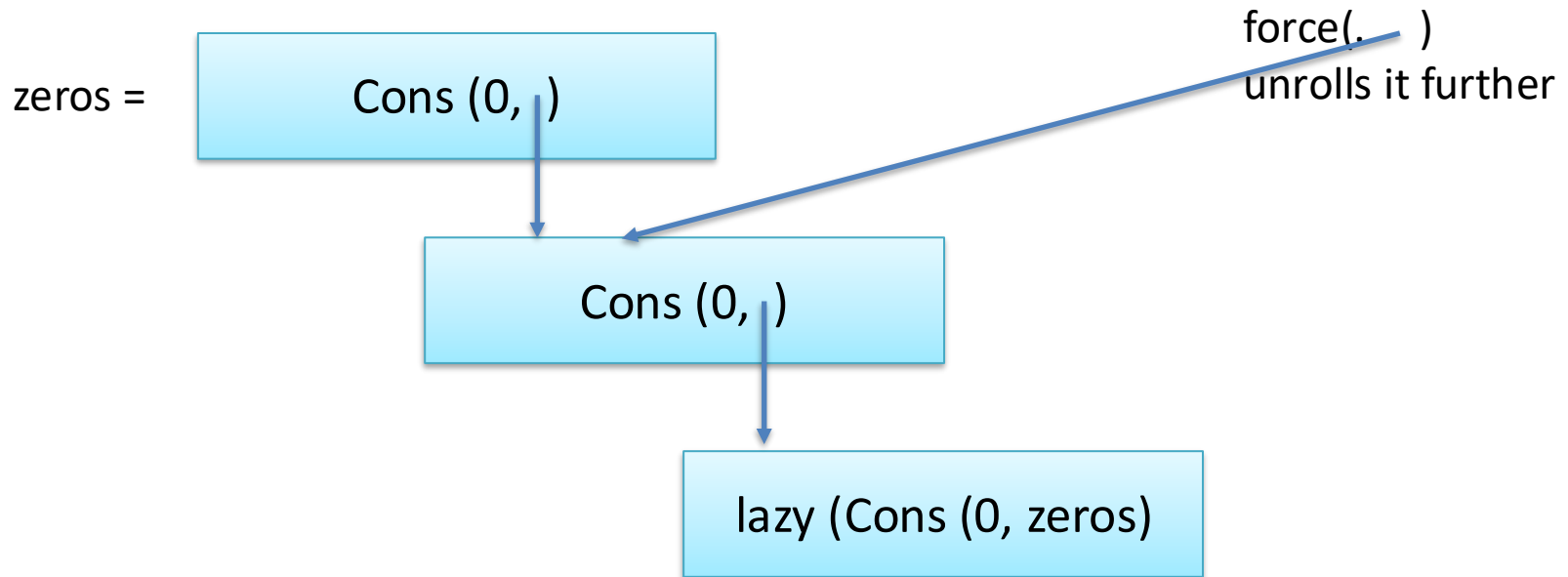
let rec zeros = lazy (Cons (0, zeros))
```



force(zeros)
evaluates the
lazy expression
but generates
another lazy
data structure
inside

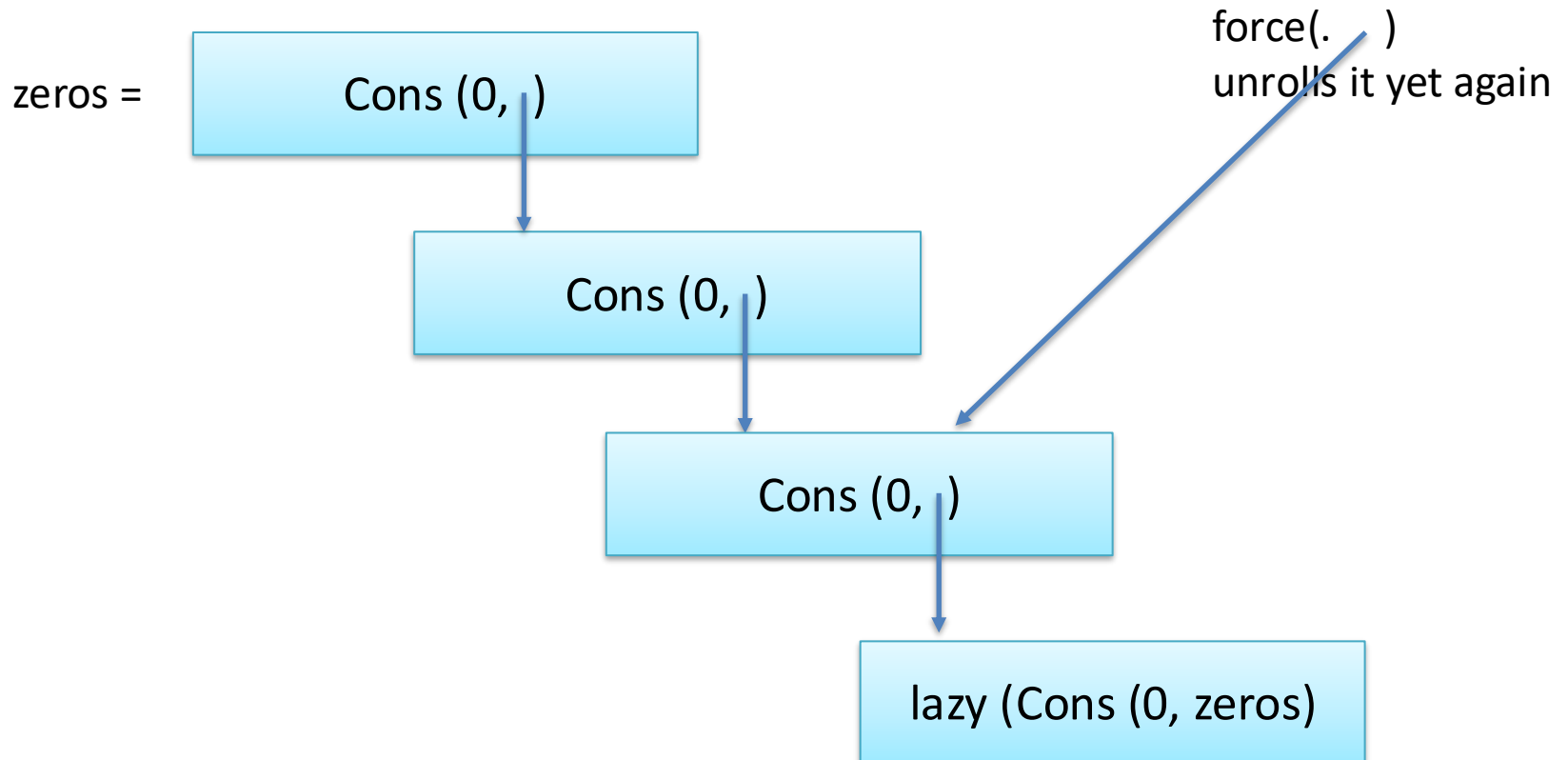
```
type 'a s = Cons of 'a * ('a stream)
and 'a stream = 'a s lazy_t
```

```
let rec zeros = lazy (Cons (0, zeros))
```



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

```
let rec zeros = lazy (Cons (0, zeros))
```



open Lazy

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

```
let rec zeros = lazy (Cons (0, zeros))
```

```
let head s =
  let Cons (hd, _) = force s in hd
```

```
let tail s =
  let Cons (_, tl) = force s in tl
```

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

let rec zeros = lazy (Cons (0, zeros))

let head s =
  let Cons (hd,_) = force s in hd

let tail s =
  let Cons (_,tl) = force s in tl

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

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

let rec zeros = lazy (Cons (0, zeros))

let rec map f s =
  lazy (Cons (f (head s), map f (tail s)))
```

Calling `map f s` for any `f`, no matter how expensive, returns immediately.

`force(map f s)` will give you back `Cons(something, lazy ( ... unexecuted ... ))`

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

let rec zeros = lazy (Cons (0, zeros))

let rec map f s =
  lazy (Cons (f (head s), map f (tail s)))
```

Calling `map f s` for any `f`, no matter how expensive, returns immediately.

`force(map f s)` will give you back `Cons(something, lazy ( ... unexecuted ... ))`

OCaml vs Haskell

- OCaml is call-by-value by default
 - optional laziness
- Haskell is lazy by default
 - has some call-by-value support
- Most people seem to think the OCaml approach is preferred. Laziness is very useful occasionally, but most of the time, we want call by value