A Lazy Fix!

COS 326
David Walker

Streams in OCaml

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

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

Surprisingly, this *does* work in OCaml.

```
let head (Cons (hd,tl)) = hd
let tail (Cons (hd,tl)) = tl

head ones --> 1
head (tail ones) --> 1
head (tail (tail ones)) --> 1
...
```

It's an infinite list of ones!

Two ways to think about this:

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

We can think in terms of the substitution model:

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

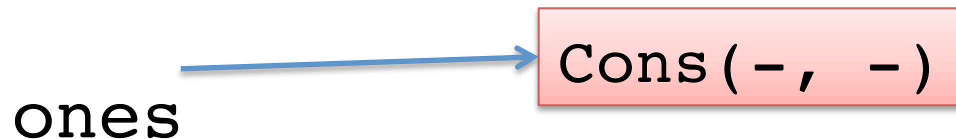
```
let ones =
 Cons(1,Cons(1,(let rec ones = Cons(1,ones) in
                ones))) ;;
```

But the substitution model tells us that we can unwind this forever. Somehow, OCaml cleverly constructs the limit of this unwinding process and represents an infinite list of ones with a finite memory…

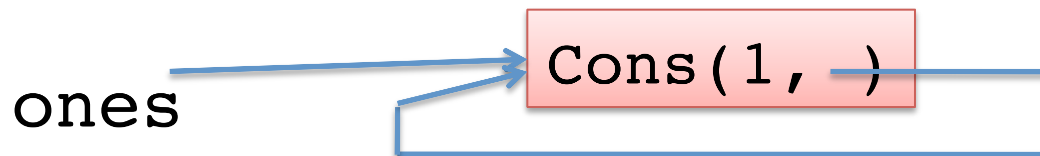
What really happens:

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

OCaml allocates space for the Cons (without initializing it yet) and makes `ones` point to the Cons-cell.



Then it initializes the contents of the Cons-cell with the values of the arguments:



This doesn't always work…

```
let rec x = 1 + x ;;
```

The example above gives us an error – we're trying to use the value of x before we've finished defining it.

In general, it seems to work only when we build a cyclic data structure where we don't peek inside the recursive parts of the data structure.

Processing Circular Lists

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

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

What happens if we write map?

```
let rec map (f:’a->’b) (s:’a stream) =
  match s with
  | Cons(h,t) -> Cons(f h, map f t)
```

Processing Circular Lists

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

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

Or equivalently:

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

Processing Circular Lists

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

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

Or equivalently:

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

map ((+) 1) ones --> ?
```

Processing Circular Lists

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

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

Alas, map will run forever on a stream (or more properly, until we run out of stack space since it's not tail-recursive.)

```
map ((+) 1) ones --> ?
```

Processing Circular Lists

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

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

We still need to convince ML to be a little less eager to unwinding recursive definitions.

`lazy_t` – the type of lazy computations

`lazy(exp)` – create a lazy expression that computes exp later

`Lazy.force e` – do the computation now (if not already done) and extract the result from the lazy computation

Back to Lazy Lists

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

Back to Lazy Lists

```
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,_) = Lazy.force s in hd

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

Back to Lazy Lists

```
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,_) = Lazy.force s in hd

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

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

Back to Lazy Lists

```
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,_) = Lazy.force s in hd

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

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

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