Mutation

COS 326
David Walker
Princeton University

Mutation?

Breaking News
LIVE
Mutant Alert! New cow-goat
TVN
EXCLUSIVE
ICANHASCHEEZBURGER.COM

Thus far…

- We have considered the (almost) purely functional subset of Ocaml.
  - We've had a few side effects: printing & raising exceptions.
- Two reasons for this emphasis:
  - *Reasoning about functional code is easier*.
    - Both formal reasoning
      - equationally, using the substitution model
      - and informal reasoning
    - Why? *because anything you can prove true stays true*.
      - e.g., 3 is a member of set S.
    - Data structures are *persistent*.
      - They don't change – we build new ones and let the garbage collector reclaim the unused old ones.
  - *To convince you that you don't need side effects for many things where you previously thought you did*.
    - there's no need for a loop to have a mutable counter that we update each time -- we can use recursion and immutable state
    - You can implement *functional* data structures like 2-3 trees or red-black trees or stacks or queues or sets with reasonable space and time.

But alas…

- *Purely functional code is pointless*.
  - The whole reason we write code is to have some effect on the world.
  - For example, the Ocaml top-level loop prints out your result.
    - Without that printing (a side effect), how would you know that your functions computed the right thing?
- *Some algorithms or data structures need mutable state*.
  - Hash-tables have (essentially) constant-time access and update.
    - The best functional dictionaries have either:
      - logarithmic access & update
      - constant access & linear update
      - constant update & linear access
    - Don't forget that we give up something for this:
      - we can't go back and look at previous versions of the dictionary. We *can* do that in a functional setting.
  - Robinson's unification algorithm
    - A critical part of the Ocaml type-inference engine.
    - Also used in other kinds of program analyses.
  - Some persistent functional data structures
    - Queues, functional arrays (see assignment 6)

Reasoning about Mutable State is Hard

mutable set

```
insert i s1;
f x;
member i s1
```

immutable set

```
let s1 = insert i s0 in
f x;
member i s1
```

Is member i s1 == true? …

- When s1 is mutable, one must look at f to determine if it modifies s1.
- Worse, one must often solve the *aliasing problem*.
- Worse, in a concurrent setting, one must look at *every other function* that *any other thread may be executing* to see if it modifies s1.

Moral: *use mutable data structures only where necessary*.

- This will also be true when you use Java or C/C++ or Python or …
- It's harder to be disciplined in non-functional languages.
- Functional languages help you out by setting a good default

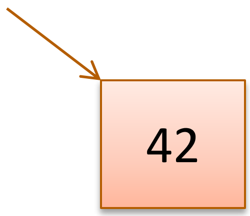
OCAML MUTABLE REFERENCES

References

- New type: `t ref`
  - Think of it as a pointer to a *box* that holds a t value.
  - The contents of the box can be read or written.

References

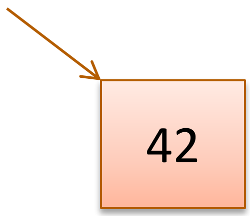
- New type: `t ref`
  - Think of it as a pointer to a *box* that holds a t value.
  - The contents of the box can be read or written.
- To create a fresh box: `ref 42`
  - allocates a new box, initializes its contents to 42, and returns a pointer:



  - `ref 42 : int ref`

References

- New type: `t ref`
  - Think of it as a pointer to a *box* that holds a t value.
  - The contents of the box can be read or written.
- To create a fresh box: `ref 42`
  - allocates a new box, initializes its contents to 42, and returns a pointer:



  - `ref 42 : int ref`
- To read the contents: `!r`
  - if `r` points to a box containing 42, then return 42.
  - if `r : t ref` then `!r : t`
- To write the contents: `r := 42`
  - updates the box that r points to so that it contains 42.
  - if `r : t ref` then `r := 42 : unit`

Example

```
let c = ref 0 ;;

let x = !c ;;     (* x will be 0 *)

c := 42 ;;

let y = !c ;;     (* y will be 42.
                     x will still be 0! *)
```

Another Example

```
let c = ref 0 ;;

let next() =
  let v = !c in
  (c := v+1 ; v)
```

Another Example

```
let c = ref 0 ;;

let next() =
  let v = !c in
  (c := v+1 ; v)
```

Recall: semi-colons conjoin two expressions

If `e1 : unit`
and `e2 : t` then
`(e1 ; e2) : t`

You can also write it like this:

```
let c = ref 0 ;;

let next() : int =
  let (v : int)   = !c in
  let (_ : unit) = c := v+1 in
  v
```

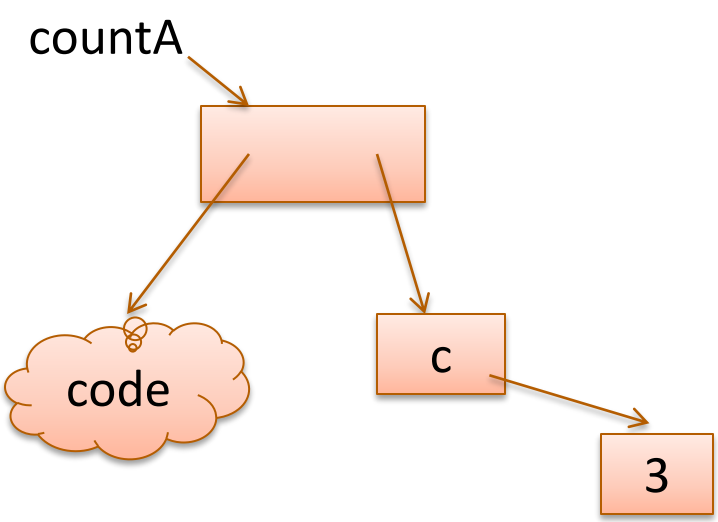
```
(e1 ; e2) == (let _ = e1 in e2)
```

(syntactic sugar)

Another Idiom

Global Mutable Reference

```
let c = ref 0 ;;

let next () : int =
  let v = !c in
  (c := v+1 ; v)
;;
```



Mutable Reference Captured in Closure

```
let counter () =
    let c = ref 0 in
   fun () ->
     let v = !c in
     (c := v+1 ; v)
;;


let countA = counter() in
let countB = counter() in
countA() ;; (* 1 *)
countA() ;; (* 2 *)
countB() ;; (* 1 *)
countB() ;; (* 2 *)
countA() ;; (* 3 *)
```

Imperative loops

```
(* sum of 0 .. n *)

let sum (n:int) =
  let         s = ref 0 in
  let current = ref n in
  while !current > 0 do
    s := !s + !current;
    current := !current - 1
  done;
  !s
;;
```

```
(* print n .. 0 *)
let count_down (n:int) =
  for i = n downto 0 do
    print_int i;
    print_newline()
  done;
;;

(* print 0 .. n *)
let count_up (n:int) =
  for i = 0 to n do
    print_int i;
    print_newline()
  done;
;;
```

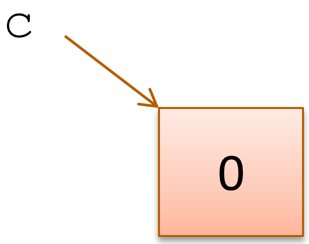
Imperative loops?

```
(* print n .. 0 *)

let count_down (n:int) =
  for i = n downto 0 do
    print_int i;
    print_newline()
  done
;;
```

```
(* for i=n downto 0 do f i *)

let rec for_down
         (n : int)
         (f : int -> unit)
            : unit =
  if n >= 0 then
    (f n; for_down (n-1) f)
  else
    ()
;;

let count_down (n:int) =
  for_down n (fun i ->
    print_int i;
    print_newline()
  )
;;
```

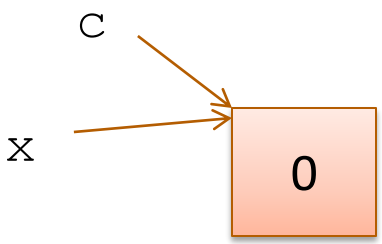
Aliasing

```
let c = ref 0 ;;

let x = c ;;

x := 42 ;;

!c ;;
```



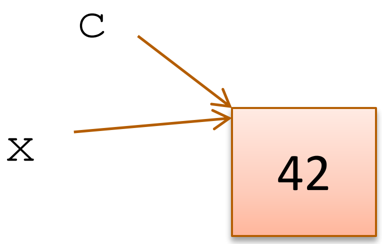
Aliasing

```
let c = ref 0 ;;

let x = c ;;

x := 42 ;;

!c ;;
```



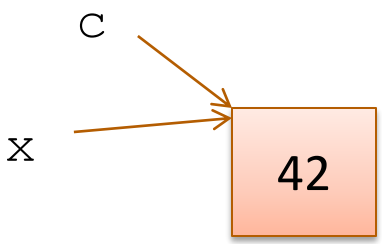
Aliasing

```
let c = ref 0 ;;

let x = c ;;

x := 42 ;;

!c ;;
```



Aliasing

```
let c = ref 0 ;;

let x = c ;;

x := 42 ;;

!c ;;
```



result: 42

MANAGING IMPERATIVE TYPES AND INTERFACES

Imperative Stacks

```
module type IMP_STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push : 'a -> 'a stack -> unit
    val pop : 'a stack -> 'a option
  end
```

Imperative Stacks

```
module type IMP_STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push : 'a -> 'a stack -> unit
    val pop : 'a stack -> 'a option
  end
```

When you see "unit" as the return type, you know the function is being executed for its side effects. (Like void in C/C++/Java.)

Imperative Stacks

```
module type IMP_STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push : 'a -> 'a stack -> unit
    val pop : 'a stack -> 'a option
  end
```

Unfortunately, we can't always tell from the type that there are side-effects going on. It's a good idea to document them explicitly. If the user can perceive them

Imperative Stacks

```
module type IMP_STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push : 'a -> 'a stack -> unit
    val pop : 'a stack -> 'a option
  end
```

Unfortunately, we can't always tell from the type that there are side-effects going on. It's a good idea to document them explicitly. If the user can perceive them

Sometimes, one uses references inside a module but the data structures have functional (persistent) semantics

Imperative Stacks

```
module ImpStack : IMP_STACK =
  struct
    type ‘a stack = (‘a list) ref

    let empty() : ‘a stack = ref []

    let push(x:’a)(s:’a stack) : unit =
       s := x::(!s)

    let pop(s:’a stack) : ‘a option =
      match !s with
      | [] -> None
      | h::t -> (s := t ; Some h)
  end
```

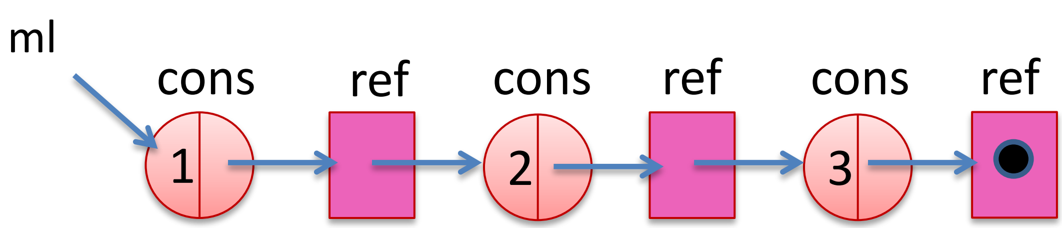
Imperative Stacks

```
module ImpStack : IMP_STACK =
  struct
    type ‘a stack = (‘a list) ref

    let empty() : ‘a stack =    f []

    let push(x:’a)(s:’a sta
       s := x::(!s)

    let pop(s:’a stack) : ‘a
      match !s with
      | [] -> None
      | h::t -> (s := t ; Some h)
  end
```

Note: We don't have to make *everything* mutable. The list is an immutable data structure stored in a single mutable cell.

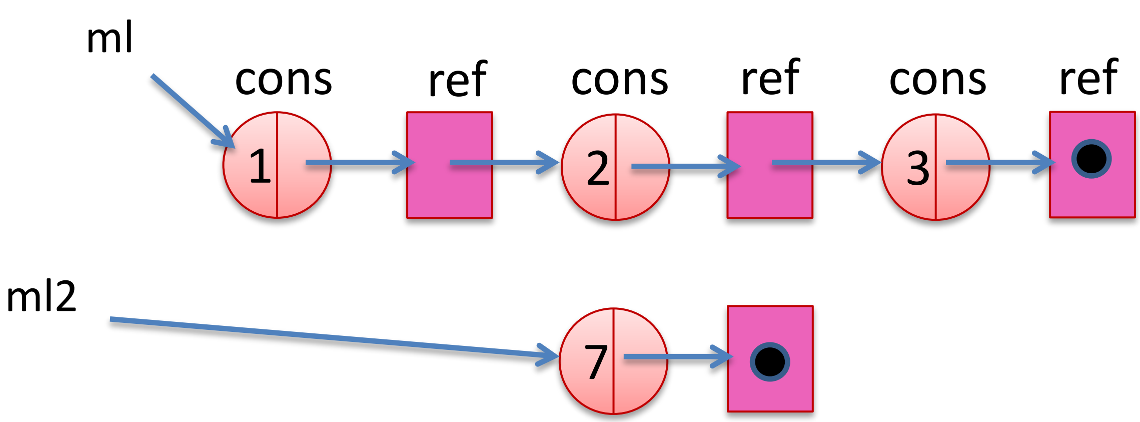
Fully Mutable Lists

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec length(m:'a mlist) : int =
  match m with
   | Nil -> 0
   | Cons(h,t) -> 1 + length(!t)
```



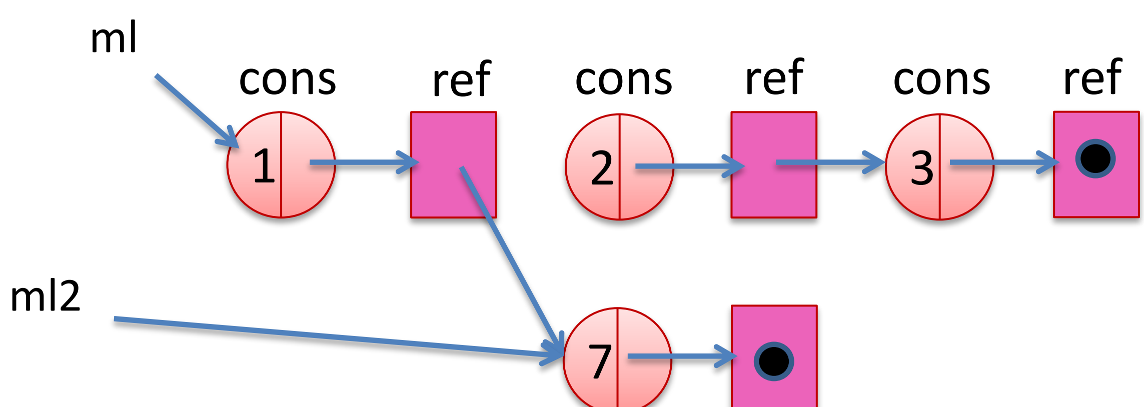
Fully Mutable Lists

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec length(m:'a mlist) : int =
  match m with
   | Nil -> 0
   | Cons(h,t) -> 1 + length(!t)
```



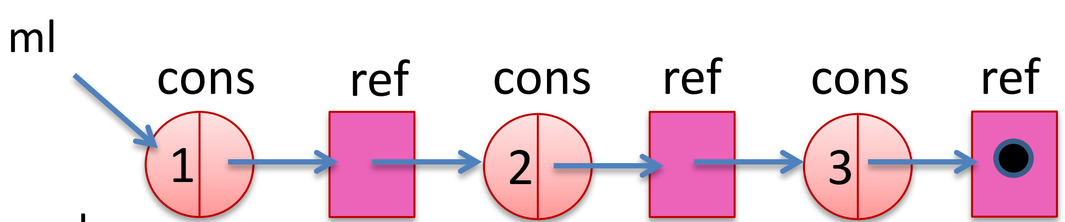
Fully Mutable Lists

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec length(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + length(!t)
```

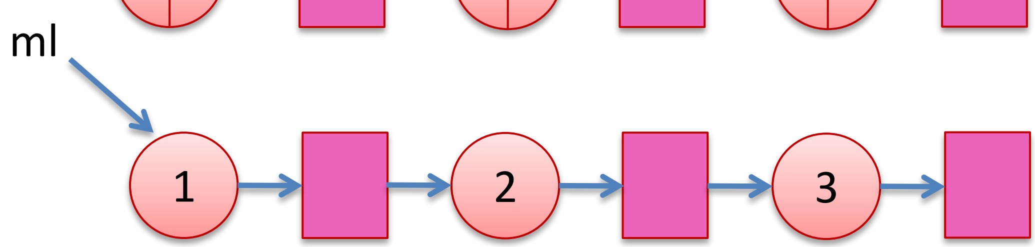


Fully Mutable Lists

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec length(m:'a mlist) : int =
  match m with
   | Nil -> 0
   | Cons(h,t) -> 1 + length(!t)
```



pictorial
convention:

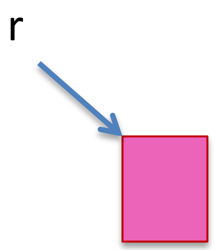


Fraught with Peril

```
type 'a mlist =
  Nil | Cons of 'a * (('a mlist) ref)

let rec mlength(m:'a mlist) : int =
  match m with
   | Nil -> 0
   | Cons(h,t) -> 1 + length(!t)

let r = ref Nil ;;
let m = Cons(3,r) ;;
r := m ;;
mlength m ;;
```

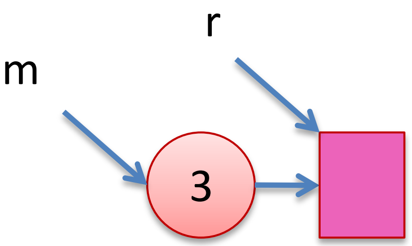
Fraught with Peril

```
type 'a mlist =
  Nil | Cons of 'a * (('a mlist) ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + length(!t)

let r = ref Nil ;;
let m = Cons(3,r) ;;
r := m ;;
mlength m ;;
```



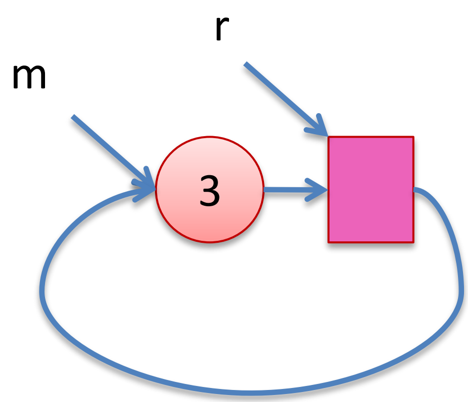
Fraught with Peril

```
type 'a mlist =
  Nil | Cons of 'a * (('a mlist) ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + length(!t)

let r = ref Nil ;;
let m = Cons(3,r) ;;
r := m ;;
mlength m ;;
```



Fraught with Peril

```
type 'a mlist =
  Nil | Cons of 'a * (('a mlist) ref)

let rec mlength(m:'a mlist) : int =
  match m with
  | Nil -> 0
  | Cons(h,t) -> 1 + length(!t)

let r = ref Nil ;;
let m = Cons(3,r) ;;
r := m ;;
mlength m ;;
```



Another Example:

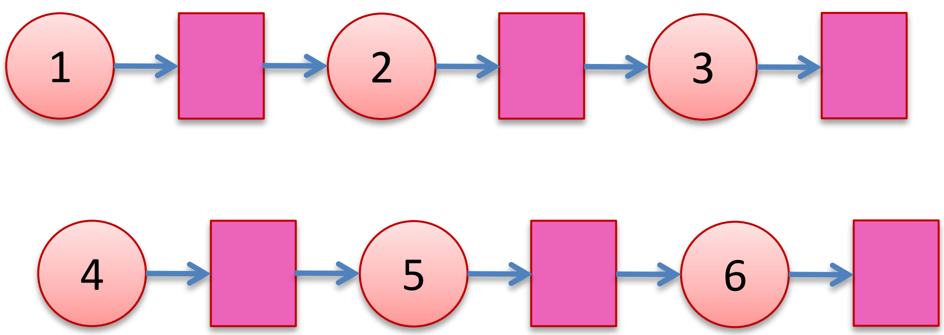
```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec mappend xs ys =
  match xs with
   | Nil -> ()
   | Cons(h,t) ->
      (match !t with
       | Nil -> t := ys
       | Cons(_,_) as m -> mappend m ys)
```

Another Example:

```
type 'a mlist =
  Nil | Cons of 'a * (('a mlist) ref)

let rec mappend xs ys =
  match xs with
   | Nil -> ()
   | Cons(h,t) ->
      (match !t with
       | Nil -> t := ys
       | Cons(_,_) as m -> mappend m ys)
```

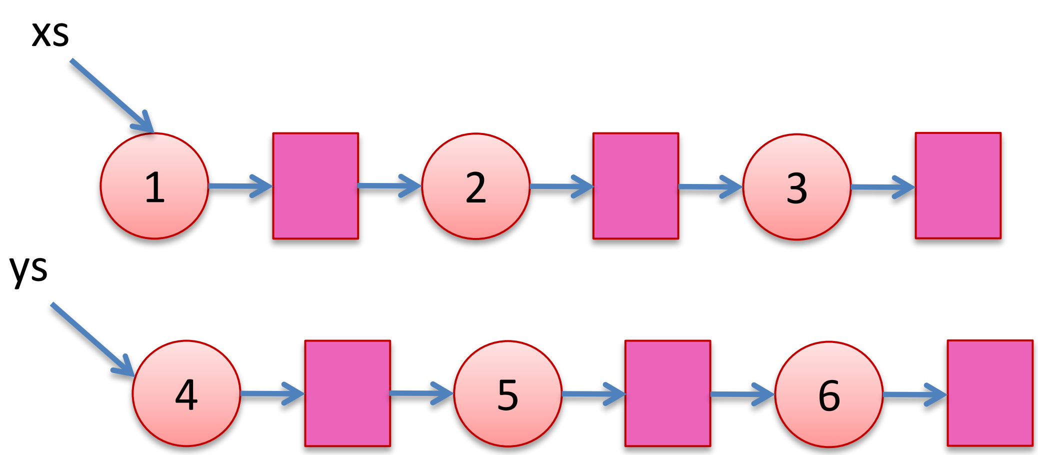
Mutable Append Example:

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h,t) ->
     (match !t with
      | Nil -> t := ys
      | Cons(_,_) as m -> mappend m ys) ;;
let xs = Cons(1,ref (Cons 2, ref (Cons 3, ref Nil))) ;;
let ys = Cons(4,ref (Cons 5, ref (Cons 6, ref Nil))) ;;
mappend xs ys ;;
```



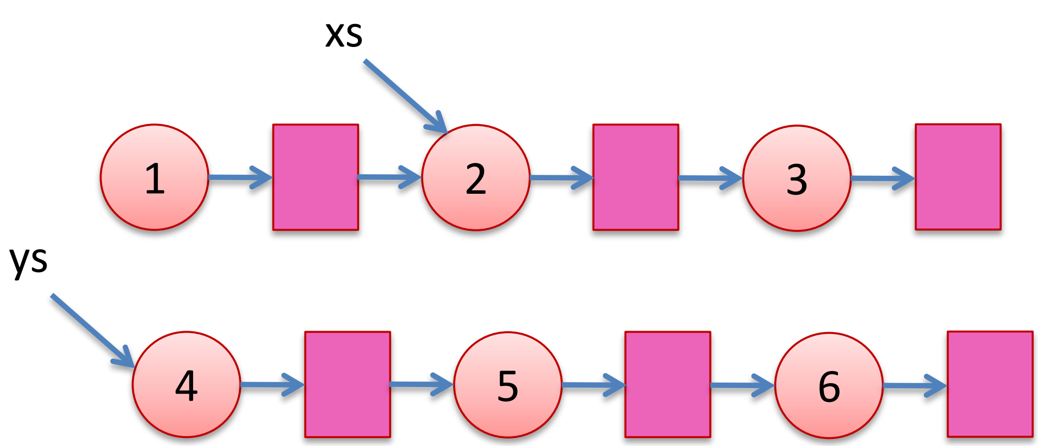
Mutable Append Example:

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h,t) ->
     (match !t with
      | Nil -> t := ys
      | Cons(_,_) as m -> mappend m ys) ;;
let xs = Cons(1,ref (Cons 2, ref (Cons 3, ref Nil))) ;;
let ys = Cons(4,ref (Cons 5, ref (Cons 6, ref Nil))) ;;
mappend xs ys ;;
```



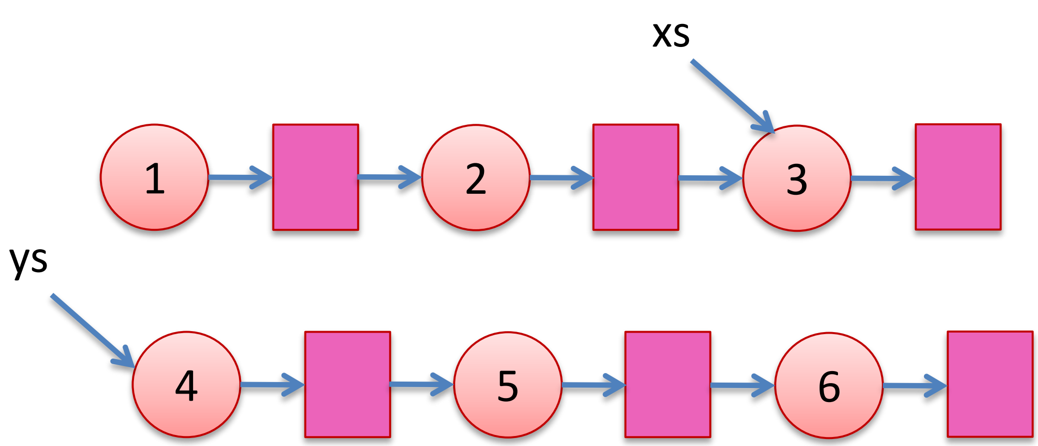
Mutable Append Example:

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h,t) ->
     (match !t with
      | Nil -> t := ys
      | Cons(_,_) as m -> mappend m ys) ;;
let xs = Cons(1,ref (Cons 2, ref (Cons 3, ref Nil))) ;;
let ys = Cons(4,ref (Cons 5, ref (Cons 6, ref Nil))) ;;
mappend xs ys ;;
```



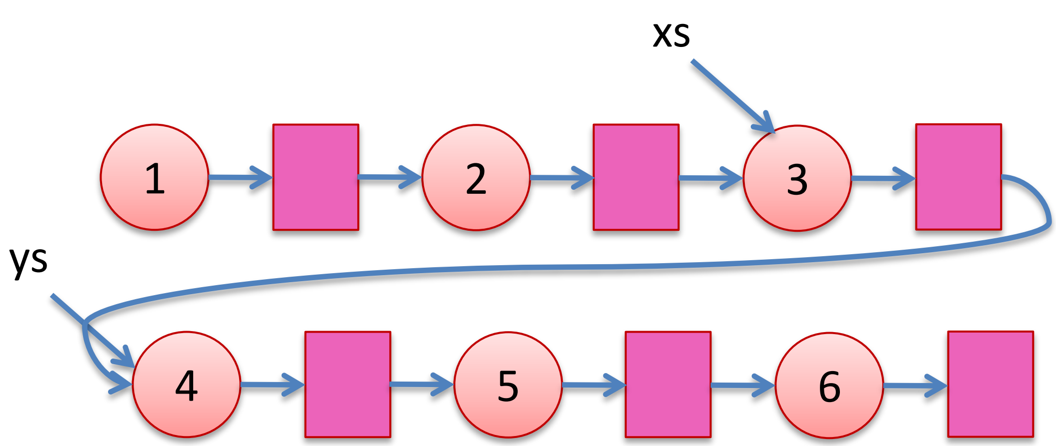
Mutable Append Example:

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h,t) ->
     (match !t with
      | Nil -> t := ys
      | Cons(_,_) as m -> mappend m ys) ;;
let xs = Cons(1,ref (Cons 2, ref (Cons 3, ref Nil))) ;;
let ys = Cons(4,ref (Cons 5, ref (Cons 6, ref Nil))) ;;
mappend xs ys ;;
```



Mutable Append Example:

```
let rec mappend xs ys =
  match xs with
   | Nil -> ()
   | Cons(h,t) ->
      (match !t with
       | Nil -> t := ys
       | Cons(_,_) as m -> mappend m ys) ;;
let xs = Cons(1,ref (Cons 2, ref (Cons 3, ref Nil))) ;;
let ys = Cons(4,ref (Cons 5, ref (Cons 6, ref Nil))) ;;
mappend xs ys ;;
```

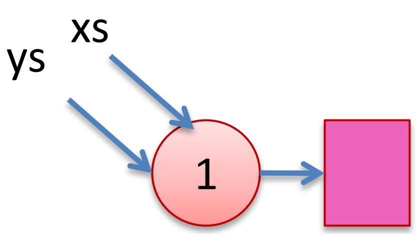


Another Example:

```
let rec mappend xs ys =
  match xs with
   | Nil -> ()
   | Cons(h,t) ->
       (match !t with
        | Nil -> t := y
        | Cons(_,_) as m -> mappend m ys)

let m = Cons(1,ref Nil);;
mappend m m ;;
mlength m ;;
```

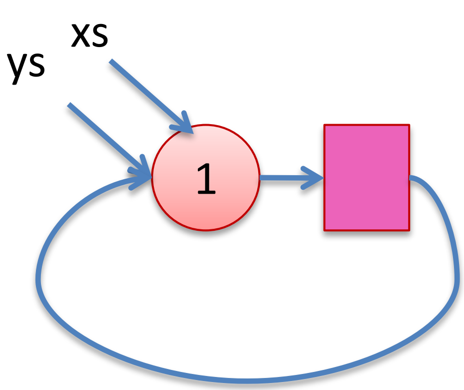
Mutable Append Example:

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h,t) ->
     (match !t with
      | Nil -> t := ys
      | Cons(_,_) as m -> mappend m ys) ;;
let m = Cons(1,ref Nil);;
mappend m m ;;
```



Mutable Append Example:

```
let rec mappend xs ys =
  match xs with
  | Nil -> ()
  | Cons(h,t) ->
     (match !t with
      | Nil -> t := ys
      | Cons(_,_) as m -> mappend m ys) ;;
let m = Cons(1,ref Nil);;
mappend m m ;;
```



Add mutability judiciously

Two types:

```
type 'a very_mutable_list =
  Nil
| Cons of 'a * (('a very_mutable_list) ref)
```

```
type 'a less_mutable_list = 'a list ref
```

The first makes cyclic lists possible, the second doesn't

- the second preemptively avoids certain kinds of errors.
- often called a *correct-by-construction design*

Is it possible to avoid all state?

- Yes! (in single-threaded programs)
  - Pass in old values to functions; return new values from functions
- Consider the difference between our functional stacks and our imperative ones:
  - `fnl_push : 'a -> 'a stack -> 'a stack`
  - `imp_push : 'a -> 'a stack -> unit`
- In general, we a dictionary that records the current values of references in to and out of every function.
  - But then accessing or updating a reference takes O(lg n) time.
  - Hash tables may be more efficient:

www.caml.inria.fr/pub/docs/manual-ocaml/libref/Hashtbl.html

MUTABLE RECORDS AND ARRAYS

Records with Mutable Fields

OCaml records with mutable fields:

```
type 'a queue1 =
  {front : 'a list ref;
   back  : 'a list ref } ;;

type 'a queue2 =
  {mutable front : 'a list;
   mutable back : 'a list} ;;

let q1 = {front = [1]; back = [2]};;
let q2 = {front = [1]; back = [2]};;

let x = q2.front @ q2.back;;

q2.front <- [3];;
```

In fact: `type 'a ref = {mutable contents : 'a}`

Mutable Arrays

For arrays, we have:

`A.(i)`

- to read the ith element of the array `A`

`A.(i) <- 42`

- to write the ith element of the array `A`

`Array.make : int -> 'a -> 'a array`

- `Array.make 42 'x'` creates an array of length 42 with all elements initialized to the character `'x'`.

See the reference manual for more operations.

www.caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html

OCaml Objects

```
class point =
  object
    val mutable x = 0
    method get_x = x
    method move d = x <- x + d
  end;;
```

```
let p = new point in
let x = p#get in

p#move 4;

x + p#get   (* 0 + 4 *)
```

http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual005.html

Xavier Leroy (OCaml inventor):

- No one ever uses objects in OCaml
- Adding objects to OCaml was one of the best decisions I ever made

SUMMARY

Summary: How/when to use state?

- In general, I try to write the functional version first.
  - e.g., prototype
  - don't have to worry about sharing and updates
  - don't have to worry about race conditions
  - reasoning is easy (the substitution model is valid!)
- Sometimes you find you can't afford it for efficiency reasons.
  - example: routing tables need to be fast in a switch
  - constant time lookup, update (hash-table)
- When I do use state, I try to *encapsulate* it behind an interface.
  - try to reduce the number of error conditions a client can see
    - correct-by-construction design
  - module implementer must think explicitly about sharing and invariants
  - write these down, write assertions to test them
  - if encapsulated in a module, these tests can be localized
  - *most of your code should still be functional*

Summary

Mutable data structures can lead to *efficiency improvements*.

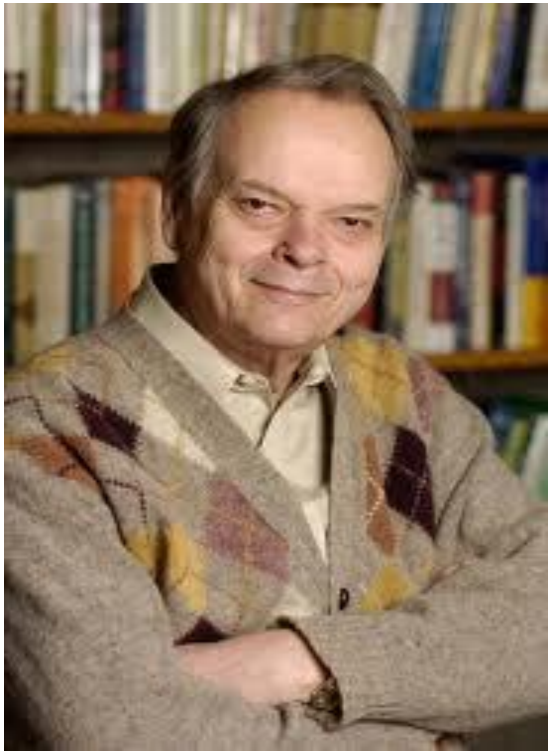
- e.g., Hash tables, memoization, depth-first search

But they are *much* harder to get right, so don't jump the gun

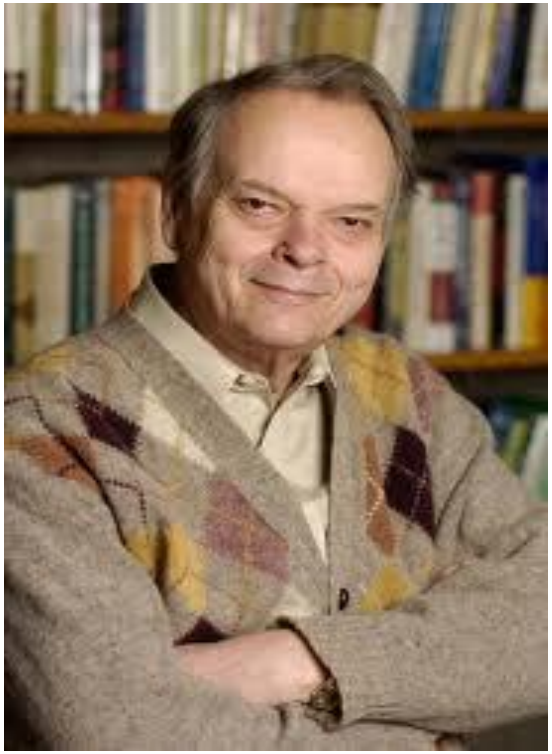
- mostly because we must think about *aliasing*.
- updating in one place may have an effect on other places.
- *writing and enforcing invariants becomes more important*.
  - e.g., assertions we used in the queue example
- *cycles in data can't happen until* we introduce refs.
  - must write operations much more carefully to avoid looping
- we haven’t even gotten to the multi-threaded part.

- *So use refs when you must, but try hard to avoid it*.

Serial Killer or PL Researcher?



Serial Killer or PL Researcher?



John Reynolds: super nice guy. Discovered the polymorphic lambda calculus. (OCaml with just functions)

Developed Relational Parametricity: A technique for proving the equivalence of modules.

Luis Alfredo Garavito: super evil guy. In the 1990s killed between 139-400+ children in Columbia. According to wikipedia, killed more individuals than any other serial killer. Due to Columbian law, only imprisoned for 30 years; decreased to 22.

END