Mutation

CSI 3120

Amy Felty

University of Ottawa

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.

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
  - Data structures are *persistent*.
    - They don't change – we build new ones and let the garbage collector reclaim the unused old ones.
  - *Hence, any invariant you prove stays true.*
    - e.g., 3 is a member of set S.
- *To convince you that you don't need side effects for many things where you previously thought you did*.
  - Programming with *basic immutable data like ints, pairs, lists is easy*.
    - types do a lot of testing for you!
    - do not fear recursion!
  - You can implement *expressive*, *highly reuseable functional* data structures like polymorphic 2-3 trees or dictionaries or stacks or queues or sets with reasonable space and time.

But alas...

Purely functional code won't do everything.

- Sometimes we really want code 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 & logarithmic 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.
- Depth-first search, more ...

However, ~~purely~~ mostly functional code is amazingly productive

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

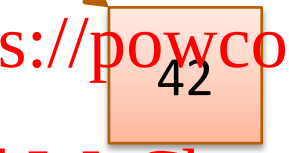
OCAML MUTABLE REFERENCES

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:

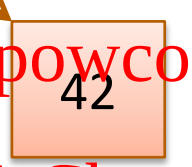


  - `ref 42 : int ref`

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 := 5`
  - updates the box that r points to so that it contains 5.
  - if `r : t ref` then `r := 5 : unit`

Example


```
let c = ref 0 in

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

c := 42;

let y = !c in      (* 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)
```


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

You can also write it like this:


```
let c = ref 0

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

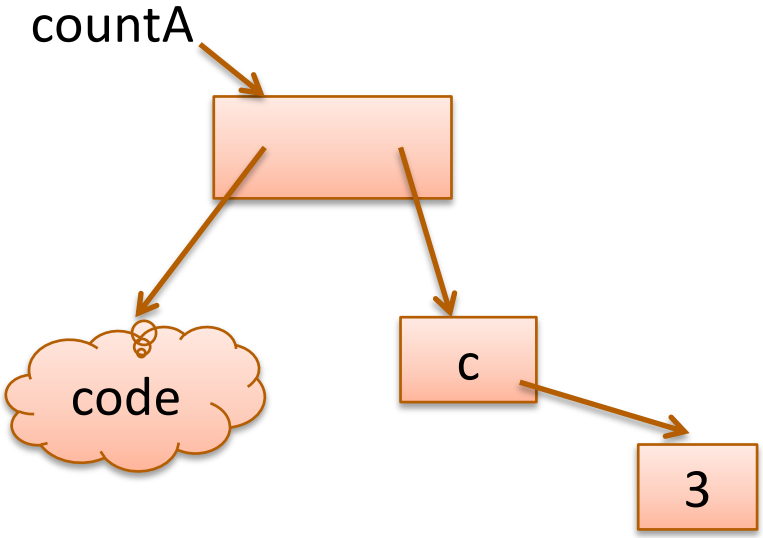
Another Idiom

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() ; (* 0 *)
countA() ; (* 1 *)
countB() ; (* 0 *)
countB() ; (* 1 *)
countA() ; (* 2 *)
```

Imperative loops


```
(* 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
```

```
(* 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
```

Imperative loops?

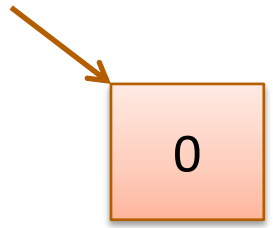

```
(* print n .. 0 *)

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()
  )
```

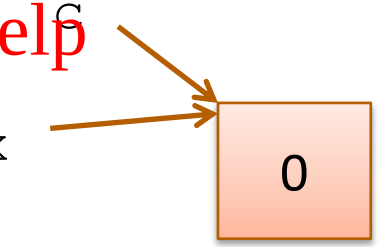
Aliasing


```
let c = ref 0

let x = c

x := 42 ;

!c
```

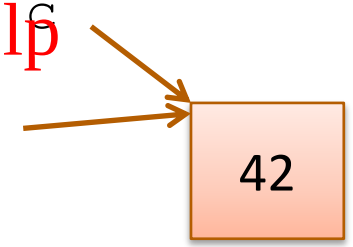
Aliasing


```
let c = ref 0

let x = c

x := 42 ;

!c
```

Aliasing


```
let c = ref 0

let x = c

x := 42 ;

!c
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

REFS AND MODULES

Types and References

Concrete, first-order type tells you a lot about a data structure:

- int ==> immutable
- int ref ==> mutable
- int * int ==> immutable
- int * (int ref) ==> 1st component immutable, 2nd mutable
- ... etc

What about higher-order types?

- int -> int ==> the function can't be changed
  ==> what happens when we run it?

What about abstract types?

- stack, queue? stack * queue?

Functional Stacks


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

Functional Stacks


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


A functional interface takes in arguments, analyzes them, and produces new results

Imperative Stacks


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

Imperative Stacks


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


This is a terrific way to use references in ML. Look for these opportunities

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
```

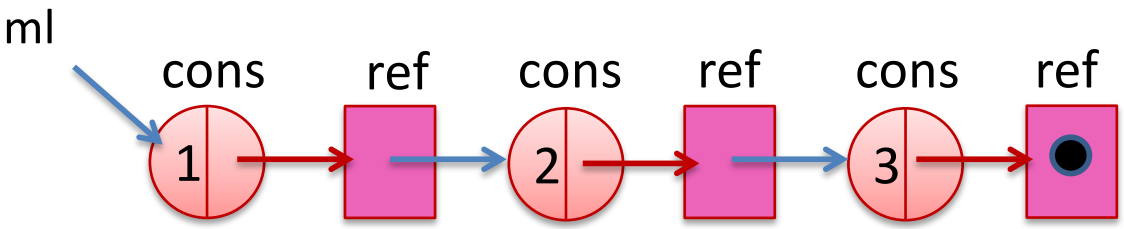
Imperative Stacks


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

    let empty() : 'a stack = ref []

    let push (x:'a) (s:'a s
      s := x::(!s)

    let pop (s:'a stack) :
      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.

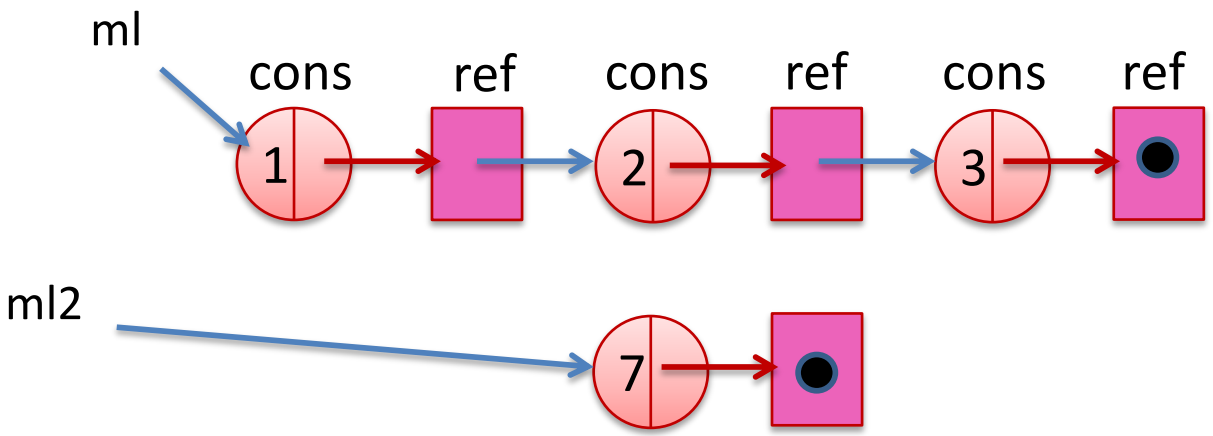
Fully Mutable Lists


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

let ml = Cons(1, ref (Cons(2, ref
   (Cons(3, ref Nil)))))
```

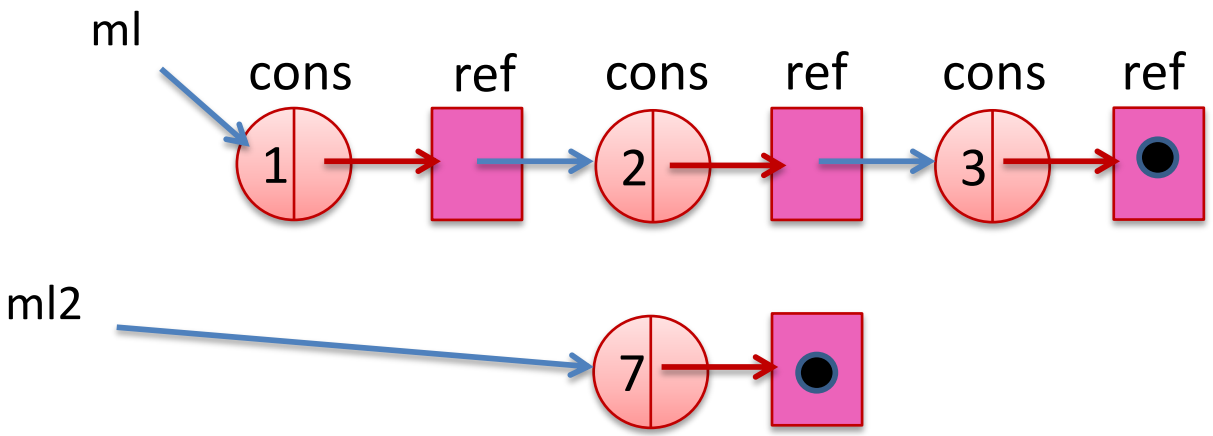
Fully Mutable Lists


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

let ml = Cons(1, ref (Cons(2, ref
   (Cons(3, ref Nil)))))

let ml2 = Cons(7, ref Nil)
```

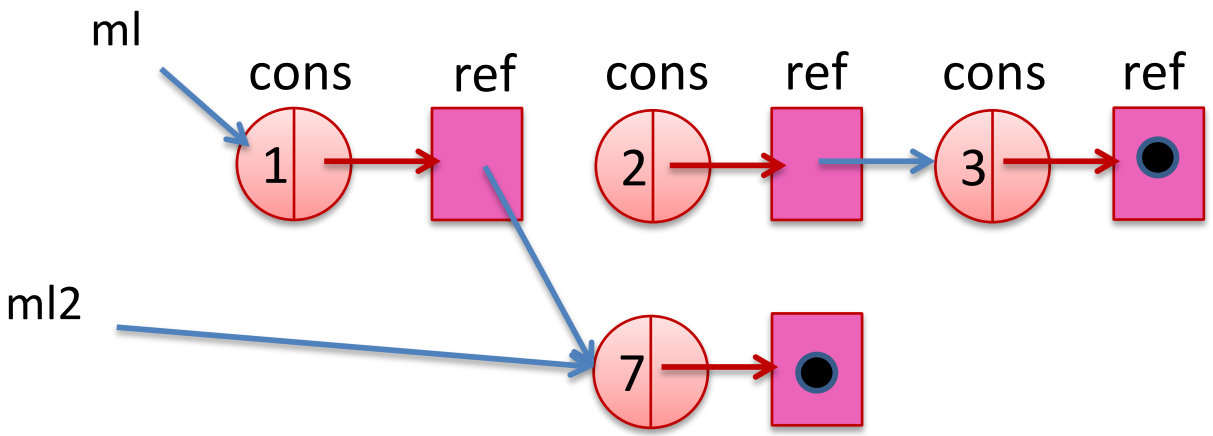
Fully Mutable Lists


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

let rec fudge(l:'a mlist)
                (m:'a mlist) : unit =
  match l with
  | Nil -> ()
  | Cons(h,t) -> t := m ; ()

fudge ml ml2
```

Fully Mutable Lists

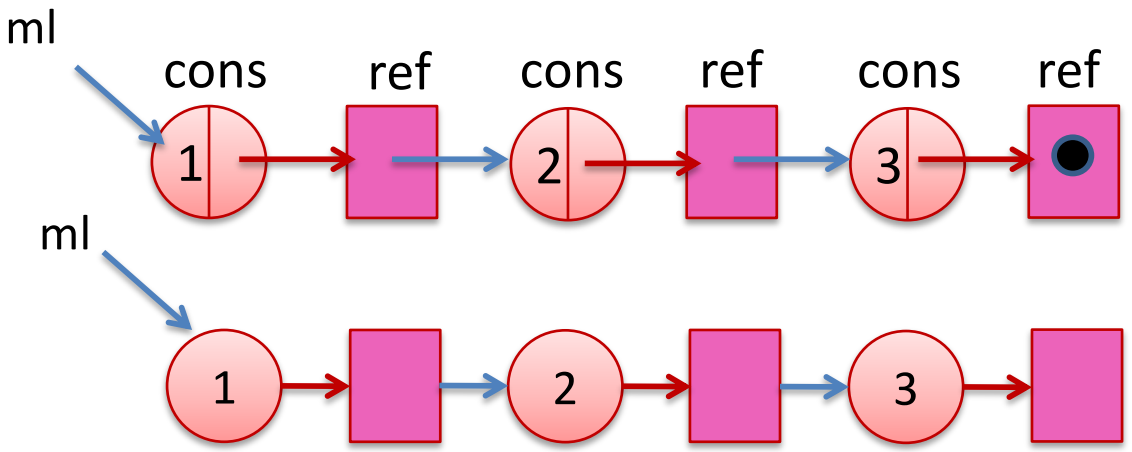

```
type 'a mlist =
  Nil | Cons of 'a * ('a mlist ref)

let rec fudge(l:'a mlist)
                (m:'a mlist) : unit =
  match l with
  | Nil -> ()
  | Cons(h,t) -> t := m ; ()

fudge ml ml2
```

Fully Mutable Lists


```
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 + mlength(!t)
```


pictorial convention:

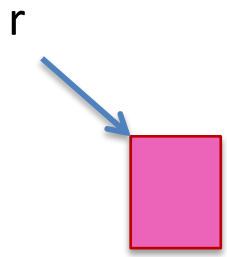


Fraught with Peril


```
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 ;;
```

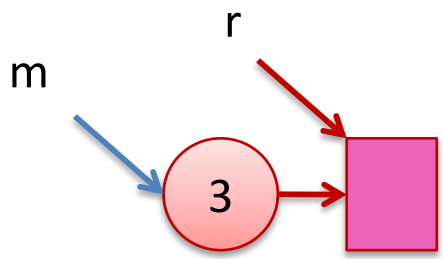
Fraught with Peril


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

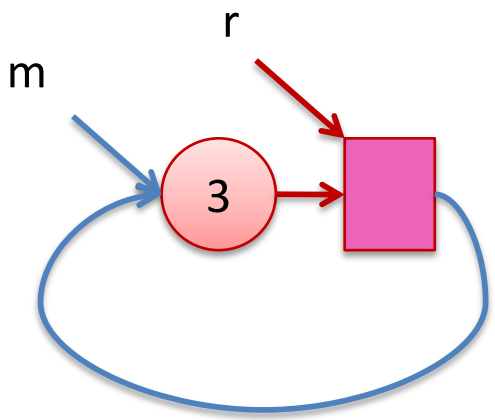
Fraught with Peril


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

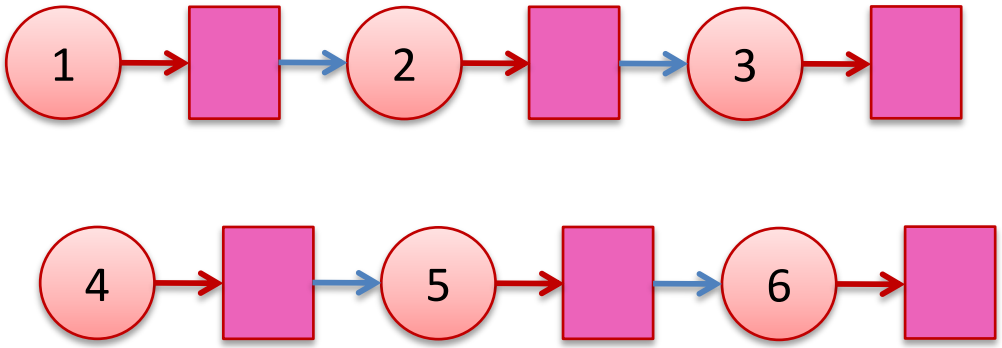



Another Example:


```
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)
```

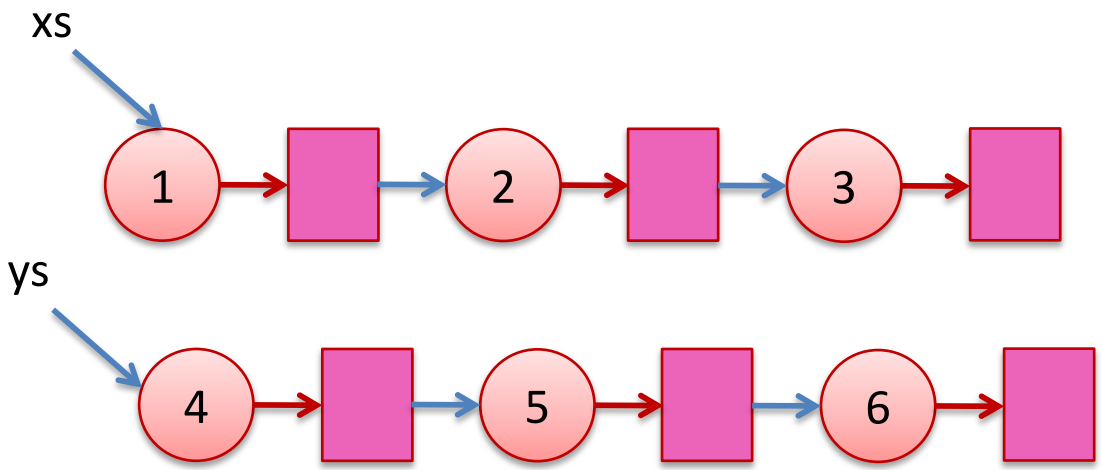
Mutable Append Example:


```
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 x = Cons(1,ref (Cons (2, ref (Cons (3, ref Nil))))) ;;
let y = Cons(4,ref (Cons (5, ref (Cons (6, ref Nil))))) ;;
mappend x y ;;
```

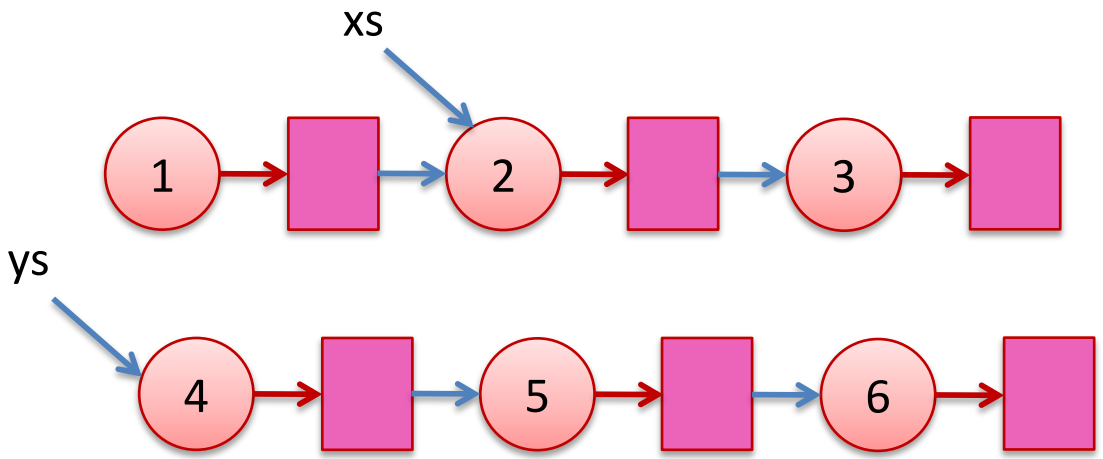
Mutable Append Example:


```
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 x = Cons(1,ref (Cons (2, ref (Cons (3, ref Nil))))) ;;
let y = Cons(4,ref (Cons (5, ref (Cons (6, ref Nil))))) ;;
mappend x y ;;
```

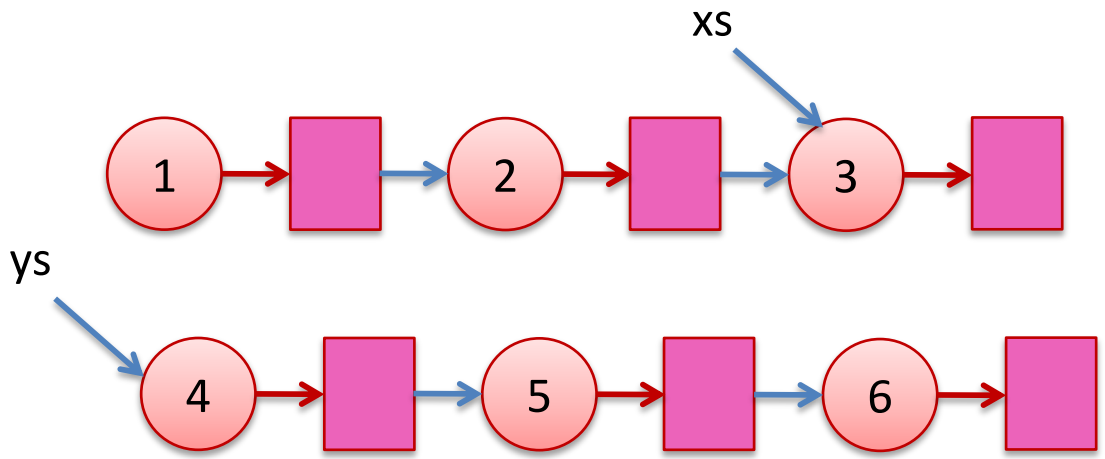
Mutable Append Example:


```
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 x = Cons(1,ref (Cons (2, ref (Cons (3, ref Nil))))) ;;
let y = Cons(4,ref (Cons (5, ref (Cons (6, ref Nil))))) ;;
mappend x y ;;
```

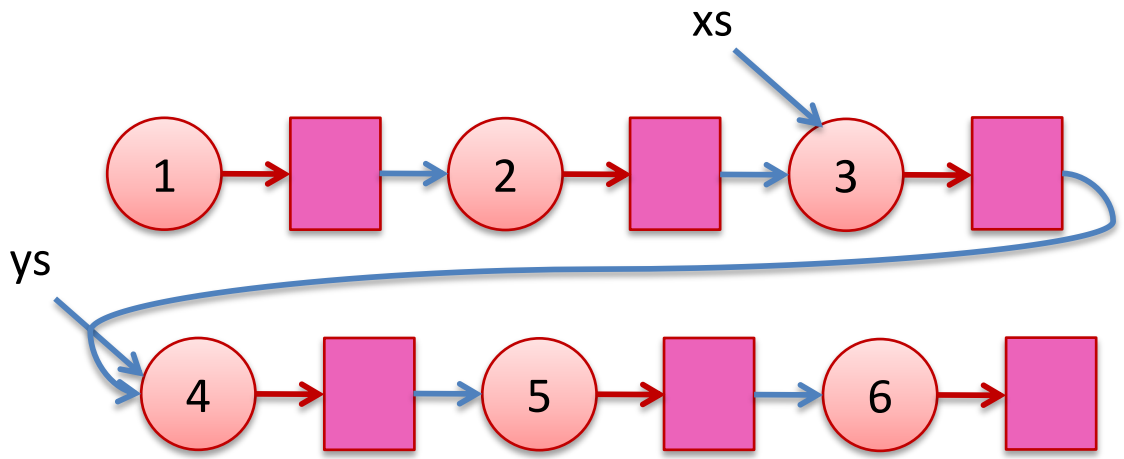
Mutable Append Example:


```
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 x = Cons(1,ref (Cons (2, ref (Cons (3, ref Nil))))) ;;
let y = Cons(4,ref (Cons (5, ref (Cons (6, ref Nil))))) ;;
mappend x y ;;
```

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 x = Cons(1,ref (Cons (2, ref (Cons (3, ref Nil))))) ;;
let y = Cons(4,ref (Cons (5, ref (Cons (6, ref Nil))))) ;;
mappend x y ;;
```

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 ... but this isn't necessarily the most efficient thing to do

Consider the difference between our functional stacks and our imperative ones:

- `fnl_push : 'a -> 'a stack -> 'a stack`
- `imp_push : 'a -> 'a stack -> unit`

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

SUMMARY

Summary: How/when to use state?

- A complicated question!
- In general, try to write the functional version first.
  - e.g., prototype
  - don't have to worry about sharing and updates
  - 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 using state, 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

- *updating in one place may have an effect on other places.*
- *writing and enforcing invariants becomes more important*.
  - why more important? because the types do less ...
- *cycles in data (other than functions) can't happen until* we introduce refs.
  - must write operations much more carefully to avoid looping
  - more cases to deal with and the compiler doesn't help you!
- we haven't even gotten to the multi-threaded part.

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