Mutation

COS 326
David Walker
Princeton University
Midterm Notes

- See Piazza for the distribution
- Use the midterm to figure out what you don't know
- Use office hours to talk to your preceptor if there is a topic you need to work on.
- Some similar problems will likely reappear on the final exam
Mutation?
Reasoning about Mutable State is Hard

Is \texttt{member i s1} == true? ...

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

\begin{align*}
\text{mutable set} & \quad \text{immutable set} \\
\text{insert} \ i \ \texttt{s1}; & \quad \text{let} \ \texttt{s1} = \text{insert} \ i \ \texttt{s0} \ \text{in} \\
\text{f} \ \texttt{x}; & \quad \text{f} \ \texttt{x}; \\
\text{member} \ i \ \texttt{s1} & \quad \text{member} \ i \ \texttt{s1}
\end{align*}
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 true 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 or expressions or programming languages 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 & 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*
John Alan Robinson

John Alan Robinson
1928 –
PhD Princeton 1956 (philosophy)
Professor (emeritus), Syracuse U.
The value of a classics degree

Inventor (1960s) of algorithms now fundamental to computational logical reasoning (about software, hardware, and other things...)

John Alan Robinson
1928 –
PhD Princeton 1956 (philosophy)

"Robinson was born in Yorkshire, England in 1930 and left for the United States in 1952 with a classics degree from Cambridge University. He studied philosophy at the University of Oregon before moving to Princeton University where he received his PhD in philosophy in 1956. He then worked at Du Pont as an operations research analyst, where he learned programming and taught himself mathematics. He moved to Rice University in 1961, spending his summers as a visiting researcher at the Argonne National Laboratory's Applied Mathematics Division. He moved to Syracuse University as Distinguished Professor of Logic and Computer Science in 1967 and became professor emeritus in 1993."

--Wikipedia
OCAML MUTABLE REFERENCES
New type: \texttt{t ref}

- Think of it as a pointer to a \textit{box} that holds a \texttt{t} value.
- The contents of the box can be read or written.
References

• New type: \texttt{t ref}
  – Think of it as a pointer to a \texttt{box} that holds a \texttt{t} value.
  – The contents of the box can be read or written.

• To create a fresh box: \texttt{ref 42}
  – allocates a new box, initializes its contents to 42, and returns a pointer:

\begin{verbatim}
  \texttt{ref 42 : int ref}
\end{verbatim}
References

• New type: \texttt{t ref}
  – Think of it as a pointer to a \textit{box} that holds a \texttt{t} value.
  – The contents of the box can be read or written.

• To create a fresh box: \texttt{ref 42}
  – allocates a new box, initializes its contents to 42, and returns a pointer:

  \begin{verbatim}
  ref 42 : int ref
  \end{verbatim}

• To read the contents: \texttt{!r}
  – if \texttt{r} points to a box containing 42, then return 42.
  – if \texttt{r : t ref} then \texttt{!r : t}

• To write the contents: \texttt{r := 5}
  – updates the box that \texttt{r} points to so that it contains 5.
  – if \texttt{r : t ref} then \texttt{r := 5 : unit}
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! *)
let c = ref 0 ;

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

```ocaml
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:

```ocaml
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**

```ml
let c = ref 0 ;;

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

**Mutable Reference Captured in Closure**

```ml
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 *)
```

**Code**

- `let c = ref 0 ;;`
- `let next () : int =
  let v = !c in
  (c := v+1 ; v)
;;`
- `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

(* 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()
  )
;;
let c = ref 0 ;;
let x = c ;;
x := 42 ;;
!c ;;
let $c = \text{ref} \ 0$ ;

let $x = c$ ;

$x := 42$ ;

$c$ ;

!c ;
let c = ref 0 ;;

let x = c ;;

x := 42 ;;

!c ;;
let \( c = \text{ref} \ 0 \);;

let \( x = c \);;

\( x := 42 \);;

\( !c \);;

warning! we can’t say \(!c == 0\)
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, 2\textsuperscript{nd} 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?
module type STACK =

sig

  type 'a stack

  val empty : unit -> 'a stack

  val push : 'a -> 'a stack -> 'a stack

  val peek : 'a stack -> 'a option

  ...

end
module type STACK =

sig

  type 'a stack

  val empty : unit -> 'a stack

  val push : 'a -> 'a stack -> 'a stack

  val peek : 'a stack -> 'a option

  ...

end

A functional interface takes in arguments, analyzes them, and produces new results
module type IMP_STACK =

  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push : 'a -> 'a stack -> unit
    val peek : 'a stack -> 'a option

  end
module type IMP_STACK =

  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push : 'a -> 'a stack -> unit
    val peek : '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.)
module type IMP_STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push : 'a -> 'a stack -> unit
    val peek : 'a stack -> 'a option
    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.
module type IMP_STACK =

sig

  type 'a stack

  val empty : unit -> 'a stack

  val push : 'a -> 'a stack -> unit

  val peek : 'a stack -> 'a option

  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.
module type IMP_STACK =

sig

  type 'a stack
  val empty : unit -> 'a stack
  val push : 'a -> 'a stack -> '
  val peek : 'a stack -> 'a option
  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.

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

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

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

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

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

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

Just like our hand-constructed example from “Fraught with Peril” slide. Good luck calling mlength on this.
Two types:

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

```haskell
type 'a less_mutable_list_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

• In general, we could pass a dictionary into and out of every function.
  – That dictionary would map “addresses” to “values”
    • it would record the value of every reference
  – But then accessing or updating a reference takes O(lg n) time.
  – ... (wonder how bad the constant factors would be, too) ...
MUTABLE RECORDS AND ARRAYS
OCaml records with mutable fields:

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

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.

Fully encapsulated state
Memoized functions

Example: factoring integers

```ocaml
let factor n =  
    let s = int_of_float (sqrt (float_of_int n))  
in let rec f i =  
    if i<=s  
    then if n mod i = 0 then Some i else f (i+1)  
    else None  
in f 2
```

Caveat 1:
Many applications of prime numbers are for many-bit (500-bit, 2000-bit) numbers; OCaml ints are 31-bit or 63-bit, so you’d want a version of this for the bignums

Caveat 2:
This primitive factoring algorithm, already obsolete 2000 years ago, is not what you’d really use. Modern algorithms based on fancy number theory are much faster.

Caveat 3:
Even the fancy number-theory algs take superpolynomial time (as function of the number of bits in n)
Example: factoring integers

```ocaml
let factor n =
    let s = int_of_float (sqrt (float_of_int n))
  in let rec f i =
      if i<=s
          then if n mod i = 0 then Some i else f (i+1)
          else None
    in f 2

# factor 77;;
- : int option = Some 7
# factor 97;;
- : int option = None
```
let table : (int, int option) Hashtbl.t = Hashtbl.create 1000

let memofactor n =
  try Hashtbl.find table n
  with Not_found ->
    let p = factor n
    in Hashtbl.add table n p; p

# let table : (int, int option) Hashtbl.t = Hashtbl.create 1000;;
val table : (int, int option) Hashtbl.t = <abstr>

# memofactor 77;;
- : int option = Some 7

# memofactor 97;;
- : int option = None

# Hashtbl.stats table;;
- : Hashtbl.statistics =
  {Hashtbl.num_bindings = 2;
   num_buckets = 1024;
   max_bucket_length = 1;
   bucket_histogram = [|1022; 2|]}
Encapsulating the side effects

```ocaml
let memofactor =
  let table : (int, int option) Hashtbl.t = Hashtbl.create 1000
  in fun n ->
    try Hashtbl.find table n
    with Not_found ->
      let p = factor n
      in Hashtbl.add table n p; p
```

The table is hidden inside the function closure; no way for the client to access it, or even know that it’s there.

From outside, we can pretend memofactor is a pure function
• NOT just based on its type (int -> int option)
• Based on its actual behavior (but how do you prove that?)

You could also use the module system to do this in a more general way, which permits several interface functions to share the same imperative data structure. In fact, you will do this in assignment 6.
Xavier Leroy (OCaml inventor):

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

```ocaml
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
SUMMARY
Summary: How/when to use state?

- A complicated question!
- 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*
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  
  • 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.*