Mutation

COS 326

David Walker

Princeton University

Reasoning about Immutable data is Easy

imagine an immutable data structure s1

```
let s1 = insert i s0 in     (* s1 contains i *)

f x;

member i s1                 (* s1 still contains i *)
```

Reasoning about Immutable data is Easy

imagine a mutable data structure s1

```
let s1 = insert i s0 in     (* s1 contains i *)

f x;

member i s1                 (* s1 still contains i? *)
```

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
```

When s1 is mutable, *one must look at f* to see if s1 changes

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*

There is no modularity

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.

```
let c = ref 0 in    (* create an int ref *)

let x = !c in       (* read *)

c := 42;            (* assign *)

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

Another Example

```
let c = ref 0

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

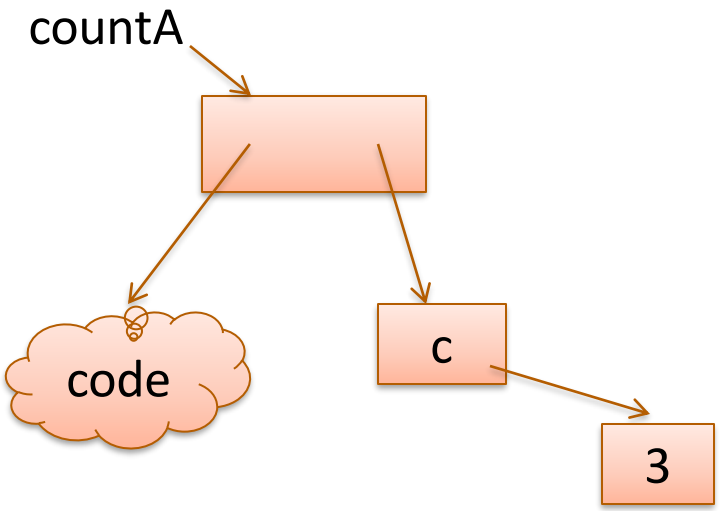
```
let x0 = next ().   (* 0 *)
let x1 = next ()    (* 1 *)
…
```

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

```
(* 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()
  )
```

Summary

Refs are OCaml's basic mutable data structure. But also:

- arrays, mutable records, objects
- discover these other features yourself!

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*.
- *cycles in data (other than functions) can't happen until* without refs
- *concurrency makes things worse*

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