

CMSC 330: Organization of Programming Languages

OCaml Imperative Programming

So Far, Only Functional Programming

- We haven't given you **any** way so far to change something in memory
 - All you can do is create new values from old
- This makes programming easier since it supports mathematical (i.e., **functional**) reasoning
 - Don't care whether data is shared in memory
 - Aliasing is irrelevant
 - Calling a function f with the same argument always produces the same result
 - For all x and y , we have $f\ x = f\ y$ when $x = y$

Immutability can sometimes make code **less efficient and more complex** for **inherently stateful problems**

- For example: Graph algorithms such as **BFS** or **Dijkstra's shortest path algorithm**, we need to:
 - Mark nodes as visited
 - Update distances
 - Maintain queues or priority queues

Sometimes Mutability Is Useful

- **Performance and Efficiency**
 - Avoid allocating new arrays or intermediate structures
- **Interfacing with the real world (I/O and Stateful APIs)**
 - Files change, Databases update records
- **Some algorithms are naturally stateful**
 - BFS, DFS
- **Caching and Memoization**
 - The cache persists across function calls
- **Modeling Evolving State (Simulations, Games, Systems)**
 - Game state (player position, score)
 - Servers maintaining session state

Immutability vs. Mutability

- **Immutability provides:** predictability, simpler reasoning, and improved safety in concurrent programs,
- **Mutability provides:** performance and clarity in problems involving frequent or natural state changes.

When to use mutability?

- Functional programming does not say “Never use mutability.”
- It says “Use immutability by default, and use mutability deliberately.”
- Best practice is to:
 - Keep mutation local
 - Hide it behind clean interfaces
 - Avoid shared, uncontrolled mutable state

Imperative OCaml

- OCaml **variables** are *immutable*, but
- OCaml has **references**, **fields**, and **arrays** that are actually *mutable*
 - I.e., they can **change**

References

- `'a ref`: Pointer to a mutable value of type `'a`

- There are three basic operations on references:

`ref 'a -> 'a ref` (* Allocate a reference *)

`! 'a ref -> 'a` (* Read the value stored in reference *)

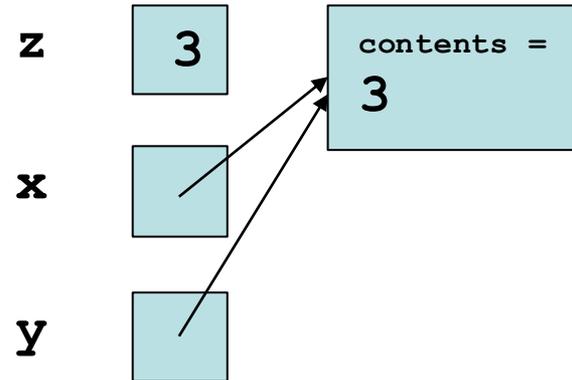
`:= 'a ref -> 'a -> unit` (* Change the value stored in reference *)

- Binding variable `x` to a reference is `immutable`
 - The `contents of the reference` `x` points to may change

References Usage

Example:

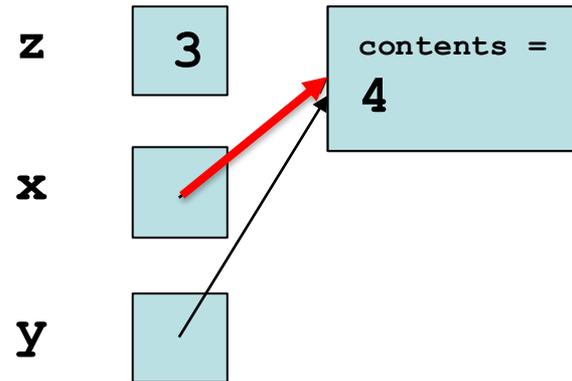
```
# let z = 3;;  
  val z : int = 3  
  
# let x = ref z;;  
  val x : int ref = {contents = 3}  
  
# let y = x;;  
  val y : int ref = {contents = 3}
```



References Usage

Example:

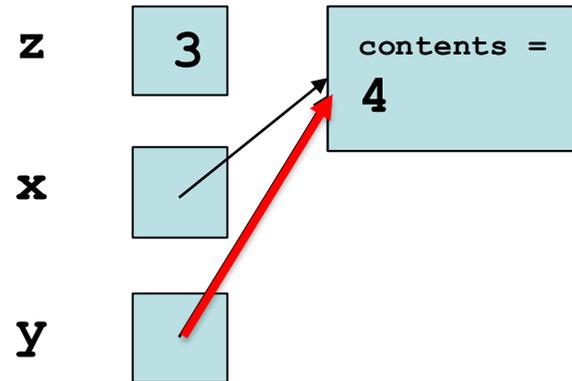
```
# let z = 3;;  
  val z : int = 3  
  
# let x = ref z;;  
  val x : int ref = {contents = 3}  
  
# let y = x;;  
  val y : int ref = {contents = 3}  
  
# x := 4;;  
- : unit = ()
```



References Usage

Example:

```
# let z = 3;;  
# let x = ref z;;  
# let y = x;;  
# x := 4;;  
# !y;;  
- : int = 4
```



Aliasing

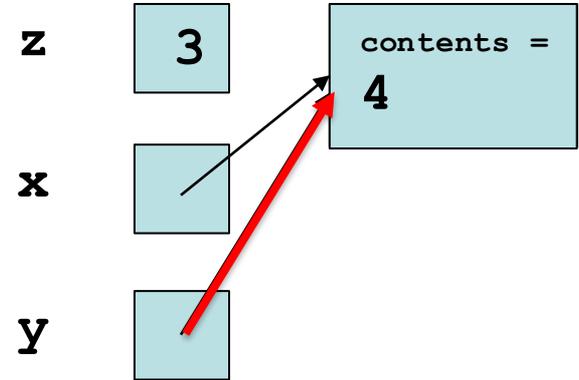
- Reconsider our example

```
let z = 3;;
```

```
let x = ref z;;
```

```
let y = x;;
```

```
x := 4;;
```



Here, variables **y** and **x** are **aliases**:

- In `let y = x`, variable **x** evaluates to a location, and **y** is bound to the **same location**
- So, changing the contents of that location will cause both **!x** and **!y** to change

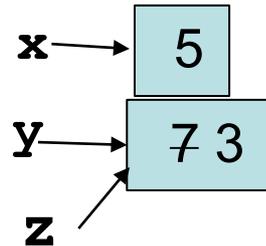
Quiz 1: What is the value **w**?

```
let x = ref 5 in
let y = ref 7 in
let z = y in
let _ = y := 3 in
let w = !y + !z in
w
```

- A. 12
- B. 6
- C. 10
- D. 8

Quiz 1: What is the value w ?

```
let x = ref 5 in
let y = ref 7 in
let z = y in
let _ = y := 3 in
let w = !y + !z in
w
```



$$!y + !z = 3 + 3 = 6$$

- A. 12
- B. 6**
- C. 10
- D. 8

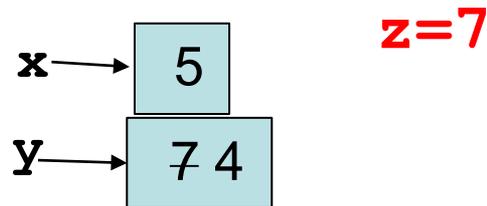
Quiz 1a: What is the value w ?

```
let x = ref 5 in
let y = ref 7 in
let z = !y in
let _ = y := 4 in
let w = !y + z in
w
```

- A. 12
- B. 6
- C. 9
- D. 11

Quiz 1a: What is the value w ?

```
let x = ref 5 in
let y = ref 7 in
let z = !y in
let _ = y := 4 in
let w = !y + z in
w
```



- A. 12
- B. 6
- C. 9
- D. 11

$$!y + z = 4 + 7 = 11$$

::; versus ;

- ::; ends an expression in the top-level of OCaml
 - Use it to say: “Give me the value of this expression”
 - Not used in the body of a function
 - Not needed after each function definition
 - Though for now it won't hurt if used there
- ***e1*; *e2*** evaluates ***e1*** and then ***e2***, and returns ***e2***

```
let print_both (s, t) = print_string s; print_string t;  
                        "Printed s and t"
```

- notice no ; at end – it's a **separator**, not a **terminator**

Grouping Sequences

- If you're not sure about the scoping rules, use `begin...end`, or *parentheses*, to group together statements with semicolons

```
let x = ref 0
let f () =
  begin
    print_string "hello";
    x := !x + 1
  end
```

```
let x = ref 0
let f () =
  (
    print_string "hello";
    x := !x + 1
  )
```

Implement a Counter

```
# let counter = ref 0 ;;
  val counter : int ref = { contents=0 }

# let next () =
  counter := !counter + 1; !counter ;;

val next : unit -> int = <fun>

# next ();;
- : int = 1

# next ();;
- : int = 2
```

Hide the Reference

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

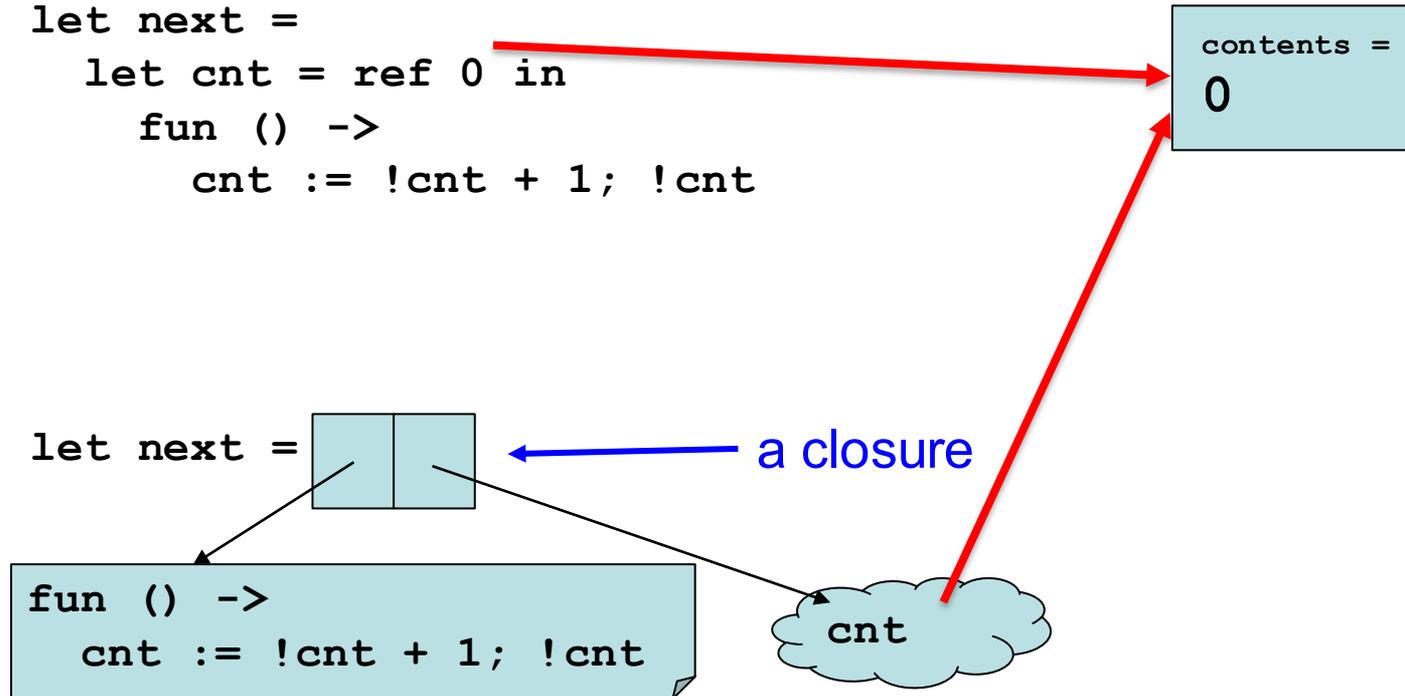
```
val next : unit -> int = <fun>
```

```
# next ();;  
- : int = 1
```

```
# next ();;  
- : int = 2
```

Hide the Reference, Visualized

```
let next =  
  let cnt = ref 0 in  
  fun () ->  
    cnt := !cnt + 1; !cnt
```



Quiz 2: What is wrong with the counter?

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

- A. It returns a boolean, not an integer
- B. It returns the same integer every time
- C. It returns a reference to an integer instead of an integer
- D. Nothing is wrong

Quiz 2: What is wrong with the counter?

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

- A. It returns a boolean, not an integer
- B. It returns the same integer every time**
- C. It returns a reference to an integer instead of an integer
- D. Nothing is wrong

The Trade-Off Of Side Effects

- Side effects are necessary
 - That's usually why we run software! We want something to happen that we can observe
- They also make reasoning harder
 - **Order of evaluation** now matters
 - **No referential transparency**
 - Calling the same function with the same arguments may produce different results
 - **Aliasing** may result in hard-to-understand bugs
 - If we call a function with refs **r1** and **r2**, it might do strange things if **r1** and **r2** are aliases

Order of Evaluation

- Consider this C example

```
int x = 10;
```

```
foo(x++, ++x);
```

Eval order: Left to right

```
foo(10, 12)
```

Eval order: right to left

```
foo(11, 11)
```

Order of Evaluation

- Consider this example

```
let y = ref 1;;  
let f _ z = z+1;; (* ignores first arg *)  
let w = f (y:=2) !y;;  
w;;
```

- What is **w** if **f**'s arguments are evaluated **left to right**?
 - 3
- What if they are evaluated **right to left**?
 - 2

OCaml Order of Evaluation

- In OCaml, the **order of evaluation** is **unspecified**
 - This means that the language doesn't take a stand, and different implementations may do different things
- On my Mac, OCaml evaluates **right to left**
 - True for the bytecode interpreter and x86 native code
 - Run the previous example and see for yourself!
- Strive to make your programs **produce the same answer regardless of evaluation order**

Order of Evaluation

List items are evaluated in right to left order

```
let f () = Printf.printf "F\t";;
```

```
let g () = Printf.printf "G\t";;
```

```
[f (); g ()]
```

```
G F - : unit list = [(); ()]
```

g () is called before **f ()**

Structural vs. Physical Equality

- Structural comparison: = and <>
- Physical comparison: == and !=
- let x = [1;2;3];; let y = [1;2;3];;
 - (x = y) (* true *) (x <> y) (* false *)
 - (x == y) (* false *) (x != y) (* true *)
- Mostly you want to use = and <>
 - E.g., the = operator is used for pattern matching
- But = is a problem with **cyclic data structures**

Equality of refs themselves

- Refs are compared **structurally** by their **contents**,
physically by their **addresses**
 - `ref 1 = ref 1` (* true *)
 - `ref 1 <> ref 2` (* true *)
 - `ref 1 != ref 1` (* true *)
 - `let x = ref 1 in x == x` (* true *)

Mutable fields

- Fields of a record type can be declared as mutable:

```
type point = {x:int; y:int; mutable c:string};;
```

```
let p = {x=0; y=0; c="red"};;
```

```
p.c <- "white";;
```

```
p.x <- 3;;
```

Error: The record field x is not mutable

Implementing Refs

- Ref cells are essentially syntactic sugar:

```
type 'a ref = { mutable contents: 'a }  
let ref x = { contents = x }  
let (!) r = r.contents  
let (:=) r newval = r.contents <- newval
```

- ref type is declared in `Stdlib`
- ref functions are compiled to equivalents of above

Hash Table

- Hashtbl Module

```
let h = Hashtbl.create 1331;  
Hashtbl.add h "alice" 100;;  
Hashtbl.add h "bob" 200;;  
Hashtbl.iter (Printf.printf "%s,%d\n") h;;
```

```
(alice,100)
```

```
(bob,200)
```

List.assoc as Map

- An *association list* is an easy implementation of a map (aka dictionary)

```
let d = [("alice", 100); ("bob", 200);  
         ("cathy", 300)]. (* (string * int) list *)  
# List.assoc "alice" d;;  
- : int = 100
```

```
List.assoc "frank" d;;  
Exception: Not_found.
```

Build a Map Using Functions

```
let empty v = fun _-> 0;;  
let update m k v = fun s->if k=s then v else m s
```

```
let m = empty 0;;  
let m = update m "foo" 100;;  
let m = update m "bar" 200;;  
let m = update m "baz" 300;;
```

```
m "foo" ;; (* 100 *)  
m "bar" ;; (* 200 *)  
let m = update m "foo" 101;;  
m "foo" ;; (* 101 *)
```

Challenge: change the code to return all the values for a key