CMSC 330: Organization of Programming Languages

OCaml Imperative Programming

CMSC330 Spring 2024
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
Imperative OCaml

• Sometimes it is useful for values to change
  • Call a function that returns an *incremented* counter
  • Store aggregations in *efficient* hash tables

• 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:

```ocaml
# 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:

```ocaml
# 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:

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

- $z = 3$
- $x$ points to $z$ with contents $3$
- $y$ points to $x$ with contents $4$
Aliasing

• Reconsider our example

```ml
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 \( w \)?

\[
\begin{align*}
\text{let } & \ x = \text{ref } 5 \ \text{in} \\
\text{let } & \ y = \text{ref } 7 \ \text{in} \\
\text{let } & \ z = !y \ \text{in} \\
\text{let } & \ _ = y := 4 \ \text{in} \\
\text{let } & \ w = !y + z \ \text{in} \\
w
\end{align*}
\]

A. 12  
B. 6  
C. 9  
D. 11
Quiz 1a: What is the value \( w \)?

\[
\begin{align*}
\text{let } x &= \text{ref } 5 \text{ in} \\
\text{let } y &= \text{ref } 7 \text{ in} \\
\text{let } z &= !y \text{ in} \\
\text{let } _ &= y := 4 \text{ in} \\
\text{let } w &= !y + z \text{ in} \\
\text{w}
\end{align*}
\]

\[
\begin{align*}
x &\rightarrow 5 \\
y &\rightarrow 7 \ 4 \\
z &= 7 \\
!y + z &= 4 + 7 = 11
\end{align*}
\]

A. 12  
B. 6  
C. 9  
D. 11
References: Syntax and Semantics

• **Syntax**: $\text{ref } e$

• **Evaluation**
  - Evaluate $e$ to a value $v$
  - Allocate a new location $loc$ in memory to hold $v$
  - Store $v$ in contents of memory at $loc$
  - Return $loc$ (which is itself a value)

• **Type checking**
  - $(\text{ref } e) : t \text{ ref}$
    - if $e : t$
References: Syntax and Semantics

• **Syntax**: $e_1 := e_2$

• **Evaluation**
  - Evaluate $e_2$ to a value $v_2$
  - Evaluate $e_1$ to a location $loc$
  - Store $v_2$ in contents of memory at $loc$
  - Return ()

• **Type checking**
  - $(e_1 := e_2) : \text{unit}$
    - if $e_1 : t \text{ ref}$ and $e_2 : t$
References: Syntax and Semantics

• Syntax: \( !e \)
  • *This is not negation. Operator \( ! \) is like operator \( * \) in C*

• Evaluation
  • Evaluate \( e \) to a location \( loc \)
  • Return contents \( v \) of memory at \( loc \)

• Type checking
  • \( !e : t \)
    • if \( e : t \) ref
Sequences: Syntax and Semantics

• Syntax: \( e_1 ; e_2 \)
  • \( e_1 ; e_2 \) is the same as \( \text{let } () = e_1 \text{ in } e_2 \)

• Evaluation
  • Evaluate \( e_1 \) to a value \( v_1 \)
  • Evaluate \( e_2 \) to a value \( v_2 \)
  • Return \( v_2 \)
    • Throws away \( v_1 \) – so \( e_1 \) is useful only if it has \textit{side effects}, e.g., if it modifies a reference’s contents or accesses a file

• Type checking
  • \( e_1 ; e_2 : t \)
    • if \( e_1 : \text{unit} \) and \( e_2 : t \)
• ;; 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.

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

```ocaml
let x = ref 0
let f () =
  (  
    print_string "hello";
    x := !x + 1
  )
```
## Implement a Counter

```ocaml
define counter
  let counter = ref 0
  val counter : int ref = { contents=0 }
define next
  let next () =
    counter := !counter + 1;
    !counter
  next : unit -> int = <fun>
define next
  # next ();
  - : int = 1

  # next ();
  - : int = 2
```
# let next =
    let counter = ref 0 in
    fun () ->
        counter := !counter + 1; !counter ;;

val next : unit -> int = <fun>

# next ();;
    - : int = 1

# next ();;
    - : int = 2
let next =
let cnt = ref 0 in
fun () ->
cnt := !cnt + 1; !cnt

let next =
fun () ->
cnt := !cnt + 1; !cnt

contents =
0

a closure
Quiz 2: What is wrong with the counter?

```ocaml
goto 0 in

let counter = ref 0 in

begin
  !counter := !counter + 1;
  !counter

end
```

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?

```ocaml
class 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 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
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

```plaintext
let f () = Printf.printf "F\t";;
let g () = Printf.printf "G\t";;
[f (); g ()]
G   F - : unit list = [(()); ()]

g () is called before f ()
```
Quiz 3: Will $w$’s value differ

If evaluation order is left to right, rather than right to left?

```ocaml
let y = ref 1 in
let f z = z := !z+1; !z in
let w = (f y) + (f y) in
w
```

A. True
B. False
Quiz 3: Will $w$’s value differ if evaluation order is left to right, rather than right to left?

```ocaml
let y = ref 1 in
let f z = z := !z+1; !z in
let w = (f y) + (f y) in
w
```

A. True
B. False
Quiz 4: Will \( w \)'s value differ

If evaluation order is left to right, rather than right to left?

\[
\begin{align*}
\text{let } y & = \text{ref 1 in} \\
\text{let } f z & = z := !z+1; !z \text{ in} \\
\text{let } w & = (f y) + !y \text{ in} \\
w &
\end{align*}
\]

A. True  
B. False
Quiz 4: Will \( w \)'s value differ

If evaluation order is left to right, rather than right to left?

```ml
let y = ref 1 in
let f z = z := !z+1; !z in
let w = (f y) + !y in
w
```

left to right: 4

right to left: 3

A. True
B. False
Quiz 5: Which f is not referentially transparent?

I.e., not the case that f x = f y for all x = y

A. let f z =
   let y = ref z in
   y := !y + z;
   !y

B. let f =
   let y = ref 0 in
   fun z ->
   y := !y + z; !y

C. let f z =
   let y = z in
   y+z

D. let f z = z+1
Quiz 5: Which \( f \) is not referentially transparent?

I.e., not the case that \( f \ x = f \ y \) for all \( x = y \)

A. let \( f \) \( z = \)
   let \( y = \text{ref} \ z \) in
   \( y := !y + z; \)
   \( !y \)

B. let \( f \) =
   let \( y = \text{ref} \ 0 \) in
   fun \( z \to \)
   \( y := !y + z; \)
   \( !y \)

C. let \( f \) \( z = \)
   let \( y = z \) in
   \( y+z \)

D. let \( f \) \( z = z+1 \)

This is basically the \texttt{counter} function
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 \texttt{refs} themselves

• Refs are compared \textit{structurally} by their contents, \textit{physically} by their addresses

  • \texttt{ref 1 = ref 1} \hfill (* true *)
  • \texttt{ref 1 <> ref 2} \hfill (* true *)
  • \texttt{ref 1 != ref 1} \hfill (* true *)
  • \texttt{let x = ref 1 in x == x} \hfill (* true *)
Mutable fields

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

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

# let p = {x=0; y=0; c="red"};;
val p : point = {x = 0; y = 0; c = "red"}

# p.c <- "white";;
- : unit = ()

# p;;
  p : point = {x = 0; y = 0; c = "white"}

# p.x <- 3;;
Error: The record field x is not mutable
```
Implementing Refs

- Ref cells are essentially syntactic sugar:

  ```ml
  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 `Pervasives`
- ref functions are compiled to equivalents of above
Arrays

- **Arrays** generalize ref cells from a single mutable value to a sequence of mutable values

```
# let v = [|0.; 1.|];;
  val v : float array = [|0.; 1.|]

# v.(0) <- 5.;;
  - : unit = ()

# v;;
  - : float array = [|5.; 1.|]
```
Quiz 6: What does this evaluate to?

\[
\begin{align*}
\text{let } x &= [\mid 0; 1 \mid] \text{ in} \\
\text{let } w &= x \text{ in} \\
x.(0) &\leftarrow 1; \\
x &= w
\end{align*}
\]

A. ()
B. true
C. false
D. Type error
Quiz 6: What does this evaluate to?

let x = [| 0; 1 |] in
let w = x in
x.(0) <- 1;
x == w

A. ()
B. true – they point to the same array
C. false
D. Type error
Control structures

• Traditional loop structures are useful with imperative features:

```plaintext
while e1 do e2 done
for x = e1 to e2 do e3 done
for x = e1 downto e2 do e3 done

for i = 1 to 5 do
    Printf.printf "%d " i
done;;
1 2 3 4 5,
```
Hash Table

- **Hashtbl Module**

```ocaml
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)

```ml
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

```ocaml
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