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 argument x 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
  - ! : 'a ref -> 'a
  - := : 'a ref -> 'a -> unit

• 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}
```

Diagram:
- `z`: integer value
- `x`: reference to `z`
- `y`: reference to `x`

Contents of `x` and `y`: 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 = ()
```

```
3
contents = 4
```
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 = ()

# !y;;
- : int = 4
```
Aliasing

• Reconsider our example
  
  ```ml
  let z = 3;;
  let x = ref z;;
  let y = x;;
  x := 4;;
  !y;;
  ```

• 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 $w$?

```ocaml
let x = ref 12 in
let y = ref 13 in
let z = y in
let _ = y := 4 in
let w = !y + !z in
w
```

A. 25
B. 8
C. 17
D. 16
Quiz 1: What is the value \( w \)?

```ocaml
let x = ref 12 in
let y = ref 13 in
let z = y in
let _ = y := 4 in
let w = !y + !z in
w
```

A. 25  
B. 8  
C. 17  
D. 16
Quiz 1a: What is the value $w$?

```ocaml
let x = ref 12 in
let y = ref 13 in
let z = !y in
let _ = y := 4 in
let w = !y + z in
w
```

A. 25
B. 8
C. 17
D. 16
Quiz 1a: What is the value $w$?

```ocaml
let x = ref 12 in
let y = ref 13 in
let z = !y in
let _ = y := 4 in
let w = !y + z in
w
```

A. 25  
B. 8  
C. 17  
D. 16
References: Syntax and Semantics

- **Syntax**: \texttt{ref e}

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

- **Type checking**
  - \texttt{(ref e) : t ref}
  - \texttt{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 $e_1 : t$ 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 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 \( e_2 : t \)
;; 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

print_both ("Colorless green ", "ideas sleep")

Prints "Colorless green ideas sleep", and returns
               "Printed s and t"
Grouping Sequences

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

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

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

# let next =
    fun () ->
        counter := !counter + 1; !counter ;;
val next : unit -> int = <fun>

# next ();;
- : int = 1

# next ();;
- : int = 2
```
Hide the Reference

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

# next ();;
- : int = 1

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

→ let next =
    let ctr = loc in
    fun () ->
        ctr := !ctr + 1; !ctr

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

ctr = loc
contents = 0

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

```ocaml
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?

```ocaml
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 absolutely 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

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

• The first argument to the call to \texttt{f} is the result of the assignment expression \texttt{y:=2}, which is unit ()
• The second argument is the current contents of reference \texttt{y}

• What is \texttt{w} if \texttt{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
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?

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

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

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 $y = \text{ref}\ z$ in
   $y := !y + z$;
   !$y$

B. let $f =$
   let $y = \text{ref}\ 0$ 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 $z$ ->
   $y := !y + z$; $!y$

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

D. let $f \ z = z+1$

This is basically the **counter** function
Structural vs. Physical Equality

- The `=` operator compares objects structurally
  - The `<>` operator is the negation of structural equality
- The `==` operator compares objects physically
  - The `!=` operator is the negation of physical equality

Examples
- `([1;2;3] = [1;2;3]) = true`   `([1;2;3] <> [1;2;3]) = false`
- `([1;2;3] == [1;2;3]) = false`   `([1;2;3] != [1;2;3]) = true`

- Mostly you want to use `=` and `<>`
  - E.g., the `=` operator is used for pattern matching
- But `=` is a problem with cyclic data structures
Cyclic Data Structures Possible With Ref

```ml
type 'a rlist =
    Nil | Cons of 'a * ('a rlist ref);;

let newcell x y = Cons(x,ref y);;
let updnext (Cons (_,r)) y = r := y;;

# let x = newcell 1 Nil;;
val x : int reflist = Cons (1, {contents = Nil})
```

```
x
Cons (1, __)
```

contents = Nil
Cyclic Data Structures Possible With Ref

```
type 'a rlist =
  Nil | Cons of 'a * ('a rlist ref);;
let newcell x y = Cons(x,ref y);;
let updnext (Cons (_,r)) y = r := y;;

# let x = newcell 1 Nil;;
val x : int reflist = Cons (1, {contents = Nil})

# updnext x x;;
- : unit = ()

# x == x;;
- : bool = true

# x = x;; (* hangs *)
```
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};;
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;;
val 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:

```ocaml
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

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

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

# v;;
# v;;
- : float array = [|5.; 1.|]
```
Arrays

• Syntax: \([ |e_1; \ldots; e_n| ]\)

• Evaluation
  • Evaluates to an \(n\)-element array, whose elements are initialized to \(v_1 \ldots v_n\), where \(e_1\) evaluates to \(v_1\), ..., \(e_n\) evaluates to \(v_n\)
    ➢ Evaluates them \(right\ to \(left\)

• Type checking
  • \([ |e_1; \ldots; e_n| ] : t\ array\)
    ➢ If for all \(i\), each \(e_i : t\)
Arrays

• Syntax: \texttt{e1. (e2)}

• Evaluation
  • Evaluate \texttt{e2} to integer value \texttt{v2}
  • Evaluate \texttt{e1} to array value \texttt{v1}
  • If \(0 \leq v2 < n\), where \(n\) is the length of array \texttt{v1}, then return element at offset \texttt{v2} of \texttt{v1}
  • Else raise \texttt{Invalid_argument} exception

• Type checking: \texttt{e1. (e2) : t}
  • if \texttt{e1 : t array} and \texttt{e2 : int}
Arrays

- Syntax: $e_1.(e_2) \leftarrow e_3$
- Evaluation
  - Evaluate $e_3$ to $v_3$
  - Evaluate $e_2$ to integer value $v_2$
  - Evaluate $e_1$ to array value $v_1$
  - If $0 \leq v_2 < n$, where $n$ is the length of array $v_1$, then update element at offset $v_2$ of $v_1$ to $v_3$
    - Else raise $Invalid\_argument$ exception
  - Return ()
- Type checking: $e_1.(e_2) \leftarrow e_3 : unit$
  - if $e_1 : t\ array$ and $e_2 : int$ and $e_3 : t$
Quiz 6: What does this evaluate to?

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

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

```ocaml
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:

  while e1 do e2 done
  for x=e1 to e2 do e3 done
  for x=e1 downto e2 do e3 done
Comparison To L- and R-values

• Recall that in C/C++/Java, there’s a strong distinction between l- and r-values
  • An r-value refers to just a value, like an integer
  • An l-value refers to a location that can be written

• A variable's meaning depends on where it appears
  • On the right-hand side, it’s an r-value, and it refers to the contents of the variable
  • On the left-hand side of an assignment, it’s an l-value, and it refers to the location the variable is stored in
L-Values and R-Values In C

- Notice that $x$, $y$, and $3$ all have type `int`

```
int x, y;
x = 3;
y = x;
3 = x;
```
Comparison To OCaml

<table>
<thead>
<tr>
<th>C</th>
<th>OCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>let x = ref 0;;</td>
</tr>
<tr>
<td>int y;</td>
<td>let y = ref 0;;</td>
</tr>
<tr>
<td>x = 3;</td>
<td>x := 3;; (* x : int ref *)</td>
</tr>
<tr>
<td>y = x;</td>
<td>y := (!x);;</td>
</tr>
<tr>
<td>3 = x;</td>
<td>3 := x;; (* 3 : int; error *)</td>
</tr>
</tbody>
</table>

- In OCaml, an updatable location and the contents of the location have different types
  - The location has a ref type