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
    - $f \ x = f \ x$ for all $x$
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
  Ø Allocate a reference

! : 'a ref -> 'a
  Ø Read the value stored in reference

:= : 'a ref -> 'a -> unit
  Ø 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:

```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 = ()

# !y;;
- : int = 4
```
Reconsider our example

```plaintext
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 $w$?

\begin{verbatim}
let x = ref 42 in
let y = ref 42 in
let z = x in
let () = x := 43 in
let w = !y + !z in
w
\end{verbatim}

A. 42  
B. 84  
C. 85  
D. 86  

Quiz 1: What is the value \( w \)?

```ocaml
let x = ref 42 in
let y = ref 42 in
let z = x in
let () = x := 43 in
let w = !y + !z in
w
```

A. 42
B. 84
C. 85
D. 86
Quiz 1a: What is the value \( w \)?

```plaintext
let x = ref 42 in
let y = ref 42 in
let z = !x in
let () = x := 43 in
let w = !y + z in
w
```

A. 42
B. 84
C. 85
D. Error
Quiz 1a: What is the value $w$?

```
let x = ref 42 in
let y = ref 42 in
let z = !x in
let () = x := 43 in
let w = !y + z in
w
```

A. 42  
B. 84  
C. 85  
D. Error
# 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
```
# let next =
    let counter = ref 0 in
    fun () ->
        counter := !counter + 1; !counter ;;
val next : unit -> int = <fun>

# next ();;
- : int = 1

# next ();;
- : int = 2
Quiz 2: What is wrong with the counter?

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

A. Error, because `counter` isn't in scope in the final line
B. It returns a reference to an integer instead of an integer
C. It returns the same integer every time
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. Error, because `counter` isn't in scope in the final line
B. It returns a reference to an integer instead of an integer
C. It returns the same integer every time
D. Nothing is wrong
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

contents = 0
ctr = loc
References: Syntax and Semantics

- **Syntax**: `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`
    - Note: locations are first-class values

- **Type checking**
  - `(ref e) : t 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

• 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 `let () = e1 in e2`

- **Evaluation**
  - Evaluate $e_1$ to a value $v_1$
  - Evaluate $e_2$ to a value $v_2$
  - Return $v_2$
    - We throw away $v_1$ – so $e_1$ is useful only if it has *effects*, e.g., if it changes 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

```ocaml
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

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

```plaintext
let x = ref 0
let f () =
  (  
    print_string "hello";
    x := !x + 1
  )
```
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 $r_1$ and $r_2$, it might do strange things if $r_1$ and $r_2$ are aliased
Quiz 3: What is the value $w$?

```
let f _ z = z+1 in
let y = ref 1 in
let w = f (y:=2) !y in
w
```

A. 3
B. 2
C. Type Error
D. ()
Quiz 3: What is the value \( w \)?

```
let f _ z = z+1 in
let y = ref 1 in
let w = f (y:=2) !y in

w
```

A. 3
B. 2
C. Type Error
D. ()
Quiz 4: What is the value \( w \)?

```
let f z _ = z+1 in
let y = ref 1 in
let w = f !y (y:=2) in
w
```

A. 3
B. 2
C. Type Error
D. ()
Quiz 4: What is the value \( w \)?

\[
\begin{align*}
\text{let } & f z \ _ = z+1 \ \text{in} \\
\text{let } & y = \text{ref } 1 \ \text{in} \\
\text{let } & w = f \ !y \ (y:=2) \ \text{in} \\
\text{w}
\end{align*}
\]

A. 3
B. 2
C. Type Error
D. ()
Structural vs. Physical Equality

In OCaml, the = operator compares objects structurally
- [1;2;3] = [1;2;3] (* true *)
- (1,2) = (1,2) (* true *)
- The = operator is used for pattern matching

The == operator compares objects physically
- [1;2;3] == [1;2;3] (* false *)

Mostly you want to use the first one
- But it’s a problem with cyclic data structures
Cyclic Data Structures Possible With Ref

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

```
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})
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 ref list = Cons (1, {contents = Nil})

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

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

# x = x;; (* hangs *彪
```
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;;
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;;
- : float array = [|5. ; 1.|]
```
Arrays

- **Syntax:** \[ | e_1; \ldots; e_n | \]

- **Evaluation**
  - Evaluates to an \textit{n-element} array, whose elements are initialized to \textit{v_1} \ldots \textit{v_n}, where \textit{e_1} evaluates to \textit{v_1}, \ldots, \textit{e_n} evaluates to \textit{v_n}
  - Evaluates them \textit{right to left}

- **Type checking**
  - \[ | e_1; \ldots; e_n | : t \text{ array} \]
  - If for all \textit{i}, each \textit{e_i} : \textit{t}
Arrays

- Syntax: \texttt{e1 \cdot 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: \( e_1. (e_2) \leftarrow e_3 : \text{unit} \)
  • if \( e_1 : \text{t array} \) and \( e_2 : \text{int} \) and \( e_3 : \text{t} \)
Quiz 5: What is the value \( w \)?

```latex
let x = [\mid 0; 1 \mid] in
let w = x in
x.(0) <- 1;

w
```

A. 1  
B. [\mid 0; 1 \mid]  
C. Type Error  
D. [\mid 1; 1 \mid]
Quiz 5: What is the value $w$?

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

w
```

A. 1
B. [| 0; 1 |]
C. Type Error
D. [| 1; 1 |]
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
In OCaml, an updatable location and the contents of the location have different types

- The location has a `ref` type

```ocaml
let x = ref 0;;
let y = ref 0;;
x := 3;; (* x : int ref *)
y := (!x);;
3 := x;; (* 3 : int; error *)
```
OCaml Language Choices

-x Implicit or explicit declarations?
  • Explicit – variables must be introduced with `let` before use
  • But you don’t need to specify types

-x Static or dynamic types?
  • Static – but you don’t need to state types
  • OCaml does type inference to figure out types for you
  • Good: less work to write programs
  • Bad: easier to make mistakes, harder to find errors
OCaml Programming Tips

- Compile your program often, after small changes
  - The OCaml parser often produces inscrutable error messages
  - It’s easier to figure out what’s wrong if you’ve only changed a few things since the last compile

- If you’re getting strange type error messages, add in type declarations
  - Try writing down types of arguments
  - For any expression e, can write (e:t) to assert e has type t
OCaml Programming Tips (cont.)

- Watch out for precedence and function application

```ocaml
let mult x y = x*y

mult 2 2+3   (* returns 7 *)
(* parsed as (mult 2 2)+3 *)

mult 2 (2+3) (* returns 10 *)
```
OCaml Programming Tips (cont.)

- All branches of a pattern match must return the same type

```ocaml
match x with
... -> -1 (* branch returns int *)
| ... -> () (* uh-oh, branch returns unit *)
| ... -> print_string "foo"
    (* also returns unit *)
```
OCaml Programming Tips (cont.)

× You cannot assign to ordinary variables!

```ocaml
# let x = 42;;
val x : int = 42
# x = x + 1;;       (* this is a comparison *)
-: bool = false
# x := 3;;
Error: This expression has type int but is here used with type 'a ref
```
Again: You cannot assign to ordinary variables!

```
# let x = 42;;
val x : int = 42
# let f y = y + x;;    (* captures x = 42*)
val f : int -> int = <fun>
# let x = 0;;        (* shadows binding of x *)
val x : int = 0
# f 10;;             (* but f still refers to x=42 *)
- : int = 52
```