Imperative Programming

Prof. Clarkson
Fall 2015

Today’s music: *The Imperial March*
from the soundtrack to *Star Wars, Episode V: The Empire Strikes Back*
Review

Previously in 3110:
• Functional programming
• Modular programming
• Interpreters

Today: THE DARK SIDE ARRIVES
• Imperative data structures: refs, mutable fields
• Due in about 9 days
• Implement an interpreter for OCalf, a subset of OCaml
• After this lecture, you'll have seen everything you need
Mutable features of OCaml

• Time to finally admit that OCaml has mutable features
  — It is not a *pure language*
  — *Pure* = no side effects

• I like to joke about the evils of mutability, BUT... sometimes it really is best to allow values to change, e.g.,
  — call a function that returns an incremented counter every time
  — efficient hash tables

• OCaml variables really are immutable
• But OCaml has mutable *references, fields, and arrays...*
References

- aka “ref” or “ref cell”
- Pointer to a typed location in memory

```ocaml
# let x = ref 0;;
val x : int ref = {contents = 0}
# !x;;
- : int = 0
# x:=1;;
- : unit = ()
# !x;;
- : int = 1
```
References

- The binding of \( x \) to the pointer is immutable, as always
  - \( x \) will always point to the same location in memory
  - unless its binding is shadowed
- But the contents of the memory may change
Implementing a counter

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

- `next_val()` returns 1
- then `next_val()` returns 2
- then `next_val()` returns 3
- etc.
Implementing a counter

(* better *)

let next_val =
  let counter = ref 0 in fun () ->
    incr counter;
!counter
Question

What's wrong with this implementation?

```ocaml
let next_val = fun () ->
  let counter = ref 0
  in incr counter;
     !counter
```

A. It won't compile, 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 wrong integer
D. Nothing is wrong
E. I don't know
Question

What's wrong with this implementation?

```ocaml
let next_val = fun () ->
  let counter = ref 0
  in incr counter;
  !counter
```

A. It won't compile, 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 wrong integer**
D. Nothing is wrong
E. I don't know
Follow-up

Q: Why does this implementation work?

```plaintext
let next_val =
    let counter = ref 0 in fun () ->
        incr counter;
    !counter
```

A: the closure captures `counter` in its environment
References

• **Syntax:** `ref e`

• **Evaluation:**
  – Evaluate `e` to a value `v`
  – Allocate a new `location loc` in memory to hold `v`
  – Store `v` in `loc`
  – Return `loc`
  – Note: locations are first-class values; can pass and return from functions

• **Type checking:**
  – New type constructor: `t ref` where `t` is a type
    • Note: `ref` is used as keyword in type and as keyword in value
  – `ref e : t ref` if `e : t`
• **Syntax:** \( e_1 := e_2 \)

• **Evaluation:**
  - Evaluate \( e_2 \) to a value \( v_2 \)
  - Evaluate \( e_1 \) to a location \( \text{loc} \)
  - Store \( v_2 \) in \( \text{loc} \)
  - Return \( () \)

• **Type checking:**
  - If \( e_2 : t \)
  - and \( e_1 : t \ \text{ref} \)
  - then \( e_1 := e_2 : \text{unit} \)
• **Syntax:** !e
  — note: not negation

• **Evaluation:**
  — Evaluate e to loc
  — Return contents of loc

• **Type checking:**
  — If e : t ref
  — then !e : t
References

• **Syntax:** \( e_1; e_2 \)

• **Evaluation:**
  – Evaluate \( e_1 \) to a value \( v_1 \)
  – Then **throw away** that value (note: \( e_1 \) could have side effects)
  – evaluate \( e_2 \) to a value \( v_2 \)
  – return \( v_2 \)

• **Type checking:**
  – If \( e_1 : \text{unit} \)
  – and \( e_2 : t \)
  – then \( e_1; e_2 : t \)
Implementing semicolon

Semicolon is essentially syntactic sugar:

\[ e_1; e_2 \]

(* means the same as *)

\[ \text{let } () = e_1 \text{ in } e_2 \]

Except: suppose it's not the case that \( e_1 : \text{unit} \)...  
- let syntax: type error  
- semicolon syntax: type warning
What does \textbf{w} evaluate to?

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

A. 42  
B. 84  
C. 85  
D. 86  
E. None of the above
What does \texttt{w} evaluate to?

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

A. 42
B. 84
C. 85
D. 86
E. None of the above
Aliases

References may have aliases:

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

*z* and *x* are aliases:
- in "let z = x", *x* evaluates to a location, and *z* is bound to the same location
- changing the contents of that location will cause both !x and !z to change
Equality

• Suppose we have two refs...
  – `let r1 = ref 3110`
  – `let r2 = ref 3110`

• Double equals is physical equality
  – `r1 == r1`
  – `r1 != r2`

• Single equals is structural equality
  – `r1 = r1`
  – `r1 = r2`
  – `ref 3110 <> ref 2110`

• You usually want single equals
MUTABLE FIELDS
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:

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

- That type is declared in `{Pervasives}`
- The functions are compiled down to something equivalent
Beware

Immutability is a valuable non-feature

*might seem weird that lack of feature is valuable*...
Immutable lists

We have never needed to worry about aliasing with lists!

```ml
let x = [2; 4]
let y = [5; 3; 0]
let z = x @ y
```

```ml
  \[ \begin{array}{c}
  \text{x} \quad \rightarrow \quad \begin{array}{c}
  2 \quad \rightarrow \quad 4
  \end{array} \\
  \text{y} \quad \rightarrow \quad \begin{array}{c}
  5 \quad \rightarrow \quad 3 \quad \rightarrow \quad 0
  \end{array} \\
  \text{z} \quad \rightarrow \quad \begin{array}{c}
  2 \quad \rightarrow \quad 4
  \end{array}
  \end{array} \]
```

vs.

```ml
  \[ \begin{array}{c}
  \text{x} \quad \rightarrow \quad \begin{array}{c}
  2 \quad \rightarrow \quad 4
  \end{array} \\
  \text{y} \quad \rightarrow \quad \begin{array}{c}
  5 \quad \rightarrow \quad 3 \quad \rightarrow \quad 0
  \end{array} \\
  \text{z} \quad \rightarrow \quad \begin{array}{c}
  2 \quad \rightarrow \quad 4 \quad \rightarrow \quad 5 \quad \rightarrow \quad 3 \quad \rightarrow \quad 0
  \end{array}
  \end{array} \]
```

(no code you write could ever tell, but OCaml implementation uses the first one)
OCaml vs. Java on mutable data

• In OCaml, we blissfully create aliases all the time without thinking about it because it is impossible to tell where there is aliasing
  – Example: `tl` is constant time; does not copy rest of the list
• In Java, programmers are obsessed with aliasing and object identity
  – They have to be, so that subsequent assignments affect the right parts of the program
  – Often crucial to make copies in just the right places…
Java security nightmare (bad code)

class ProtectedResource {
    private Resource theResource = ...;
    private String[] allowedUsers = ...;
    public String[] getAllowedUsers() {
        return allowedUsers;
    }
    public String currentUser() { ... }
    public void useTheResource() {
        for(int i=0; i < allowedUsers.length; i++) {
            if(currentUser().equals(allowedUsers[i])) {
                ... // access allowed: use it
                return;
            }
        }
    }
    throw new IllegalAccessExcpetion();
}
}
Have to make copies

The problem:

```java
p.getAllowedUsers()[0] = p.currentUsers();
p.useTheResource();
```

The fix:

```java
public String[] getAllowedUsers() {
    ... return a copy of allowedUsers ... 
}
```

Similar errors as recent as Java 1.7beta
Benefits of immutability

• Programmer doesn’t have to think about aliasing; can concentrate on other aspects of code
• Language implementation is free to use aliasing, which is cheap
• Often easier to reason about whether code is correct
• Perfect fit for concurrent programming

But there are downsides:
• I/O is fundamentally about mutation
• Some data structures (hash tables, arrays, …) hard(er) to implement in pure style

Try not to abuse your new-found power!
Upcoming events

• [next Thursday] A4 due

This is (reluctantly) imperative.

THIS IS 3110
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 \(n\)-element array, whose elements are initialized to \(v_1\ldots v_n\), where \(e_1\) evaluates to \(v_1\), \(\ldots\), \(e_n\) evaluates to \(v_n\)

- **Type checking:** \([|e_1; \ldots; e_n|]\) : \(t\) array if each \(e_i : t\)
Arrays

• **Syntax:** e1.(e2)

• **Evaluation:** if e1 evaluates to v1, and e2 evaluates to v2, and \(0 \leq v2 < n\), where \(n\) is the length of array v1, then evaluates to element at offset v2 of v1. If \(v2 < 0\) or \(v2 \geq n\), raises **Invalid_argument**.

• **Type checking:** e1.(e2) : t if e1 : t array and e2 : int
Arrays

- Syntax: \( e_1.(e_2) \leftarrow e_3 \)

  - Evaluation: if \( e_1 \) evaluates to \( v_1 \), and \( e_2 \) evaluates to \( v_2 \), and \( 0 \leq v_2 < n \), where \( n \) is the length of array \( v_1 \), and \( e_3 \) evaluates to \( v_3 \), then mutate element at offset \( v_2 \) of \( v_1 \) to be \( v_3 \). If \( v_2 < 0 \) or \( v_2 \geq n \), raise \texttt{Invalid_argument}. Evaluates to \( () \).
  
  - Type checking: \( e_1.(e_2) \leftarrow e_3 : \text{unit} \) if \( e_1 : t \ array \) and \( e_2 : \text{int} \) and \( e_3 : t \)

See \texttt{Array} module for more operations, including more ways to create arrays.
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**

(they work like you expect)