FUNCTIONAL PROGRAMMING

Imperative Programming

Prof. Clarkson
Summer 2015

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

Last class:
• Modules

Today: THE DARK SIDE ARRIVES
• Imperative features: refs, arrays, mutable fields, loops, I/O
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;;
# !x;;
- : unit = ()
- : int = 1
```

0
---
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

```plaintext
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 in three ways*/

let next_val =

  let counter = ref 0 in fun () ->

  begin

    incr counter;

    !counter

  end
References

- **Syntax:** `ref e`
- **Evaluation:**
  - `e ==> 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:** e₁ := e₂

• **Evaluation:**
  - e₂ ==> v₂
  - e₁ ==> loc
  - Store v₂ in loc
  - Return ()

• **Type checking:**
  - If e₂ : t
  - and e₁ : t ref
  - then e₁ := e₂ : unit
References

• **Syntax:** !\(e\)
  - note: not negation

• **Evaluation:**
  - Evaluate \(e\) to \(loc\)
  - Return contents of \(loc\)

• **Type checking:**
  - if \(e : t\) then \(!e : t\)
References

• **Syntax:** e₁; e₂

• **Evaluation:**
  - e₁ ==> v₁ then **throw away** that value
    • note: e₁ could have side effects
  - evaluate e₂ to a value v₂
  - return v₂

• **Type checking:**
  - If e₁ : unit
  - and e₂ : t
  - then e₁; e₂ : t
Implementing semicolon

Semicolon is essentially syntactic sugar:

\[
e_1; e_2
\]

(* means the same as *)

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

Except if not \texttt{e1 : unit}...

- let syntax: type error
- semicolon syntax: type warning
Question

What does \( w \) evaluate to?

\[
\begin{align*}
\text{let } x &= \text{ref } 42 \\
\text{let } y &= \text{ref } 42 \\
\text{let } z &= x \\
\text{let } () &= x := 43 \\
\text{let } w &= (\!y) + (\!z)
\end{align*}
\]

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

What does \texttt{w} evaluate to?

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

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

References may have **aliases**:

```plaintext
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

• Double equals is physical equality
  – let r1 = ref 3110
  – let r2 = ref 3110
  – r1 == r1
  – r1 != r2

• Single equals is structural equality
  – (ref 3110) = (ref 3110)
  – [1;2;3] = [1;2;3]
  – 2 <> 3

• You usually want single equals
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 \(e_1 \rightarrow v_1, \ldots, e_n \rightarrow v_n\)

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

• **Syntax:** `e1.(e2)`

• **Evaluation:** If `e1==>v1`, and `e2==>v2`, and `0<=v2<n`, where `n` is the length of array `v1`, then evaluates to element at offset `v2` of `v1`. If `v2<0` or `v2>=n`, raises `Invalid_argument`.

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

• Syntax: e₁.(e₂) <- e₃

• Evaluation: if e₁==v₁, and e₂==v₂, and 0<=v₂<n, where n is the length of array v₁, and e₃==v₃, then mutate element at offset v₂ of v₁ to be v₃. If v₂<0 or v₂>=n, raise Invalid_argument. Evaluates to ()

• Type checking: e₁.(e₂) <- e₃ : unit if e₁ : t array and e₂ : int and e₃ : t

See Array module for more operations, including more ways to create arrays
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
```

- That type is declared in `Pervasives`
- The functions are compiled down to something equivalent
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)
Input and output are **side-effects**: change the state of the program's environment

```ocaml
let read_inc_print () =
    let n = read_int() in
    print_int (n+1);
    print_newline();

let rec forever f =
    f();
    forever f

let _ = forever read_inc_print
```
I/O

• Simple output:
  – \texttt{print\_t},
    for \(t\) in \{\texttt{char}, \texttt{string}, \texttt{bytes}, \texttt{int}, \texttt{float}\}
  – \texttt{print\_newline}

• Simple input:
  – \texttt{read\_t}
    for \(t\) in \{\texttt{line}, \texttt{int}, \texttt{float}\}
    \texttt{line} really means \texttt{string}

• For more sophisticated I/O:
  – \texttt{in\_channel} and \texttt{out\_channel} types and functions on them in \texttt{Pervasives}
  – \texttt{Printf} and \texttt{Scanf} modules
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
```

(vs.

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

(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
  – So don’t worry and focus on your algorithm
• 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[] getNotAllowedUsers() {
        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.currentUser();
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 parallel 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!