Mutable Data Types

Nate Foster
Spring 2018
Review

Previously in 3110:
• Advanced data structures
  – Streams and laziness
  – Balanced binary trees

Today: THE DARK SIDE ARRIVES
• Mutable data types: refs, mutable fields, arrays
Mutable features of OCaml

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

• Sometimes it really is best to allow values to change:
  – 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

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

• The binding of $\mathbf{x}$ to the pointer is immutable, as always

• 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: hides [counter] *)

let next_val =
    let counter = ref 0
    in fun () ->
        incr counter;
    !counter
Compare these implementations

(* works *)

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

(* broken *)

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

Q: Why does the first implementation work?
A: the anonymous function captures `counter` in its scope
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 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`
References

• Syntax: `e1 := e2`

• Evaluation:
  – Evaluate `e2` to a value `v2`
  – Evaluate `e1` to a location `loc`
  – Store `v2` in `loc`
  – Return `()`

• Type checking:
  – If `e2 : t` and `e1 : t ref`
  – then `e1 := e2 : 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 \( e_1; e_2 : t \)
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

• 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:

# 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
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
  – elements are initialized to \(v_1 \ldots v_n\)
    • if \(e_1\) 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<=v2<n, where n is the length of array v1,
  – then e1.(e2) 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 evaluates to v1, and e2 evaluates to v2,
  – and 0<=v2<n, where n is the length of array v1,
  – and e3 evaluates to 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
Control structures

Traditional loop structures are useful with arrays:

- `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)
BEWARE
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!

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

```
x → [2; 4]
y → [5; 3; 0]
z → [2; 4]
```

```
x → [2; 4]
y → [5; 3; 0]
z → [2; 4; 5; 3; 0]
```

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

• OCaml: blissfully unaware of aliasing
  – Impossible to tell where there is aliasing (except when using imperative features)
  – Example: `List.tl` is constant time; does not copy rest of the list

• Java: obsession with aliasing and object identity
  – Must 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[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 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

• [Today] Charter due!
• Next week, a breather (sort of)