CS 3110

Mutable Data Types

A New Despair
Mutability Strikes Back
Return of Imperative Programming

Prof. Clarkson
Fall 2018

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

Attendance question

Best Star Wars trilogy?

A. Episodes I, II, III

B. Episodes IV, V, VI

C. Episodes VII, VIII, (IX)

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)

REFS

Demo

References

- Aka “refs” or “ref cell”
- Pointer to a typed location in memory
- The binding of a variable to a pointer is immutable but the contents of the memory may change

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`

References

- Syntax: `!e`
  - note: not negation
- Evaluation:
  - Evaluate `e` to `loc`
  - Return contents of `loc`
- Type checking:
  - If `e : t ref`
  - then `!e : t`

Question

What does **w** evaluate to?

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

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

Aliases

References may create **aliases**:

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

`z` and `x` are aliases

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

EXAMPLE: COUNTER

Demo

Semicolon

- Syntax: `e1; e2`
- Evaluation:
  - Evaluate **`e1`** to a value **`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`**

Question

What's wrong with this implementation?

```
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

Scope matters

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


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

MUTABLE FIELDS

Demo

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
```

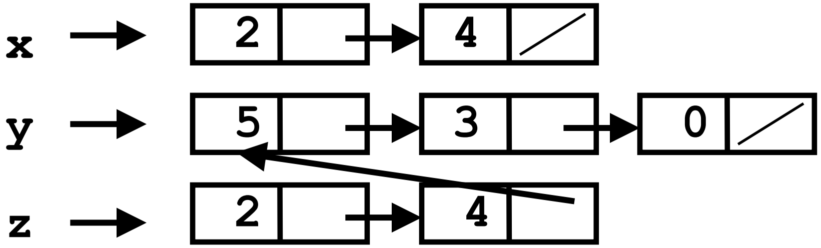
- That type is declared in **`Pervasives`**
- The functions are compiled down to something equivalent

BEWARE

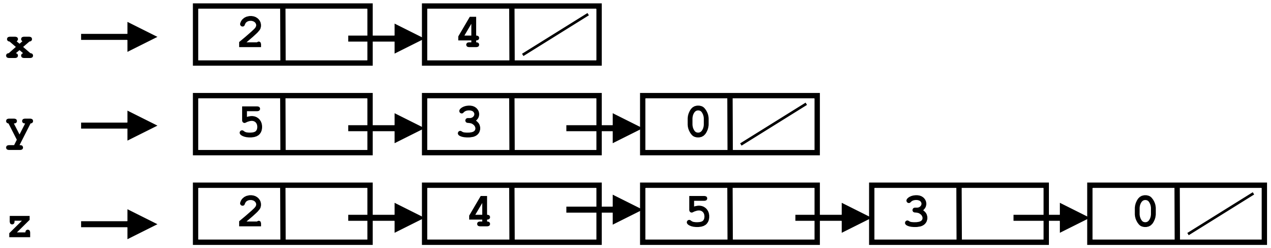
Immutable lists

We have never needed to worry about aliasing with lists!

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



vs.



(no code you write could ever tell, but OCaml implementation uses the first one)

OCaml:

blissfully unaware of aliasing

Java:

obsession with aliasing

Faulty 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();
   }
}
```

Discussion: Can you find the security fault?

Have to make copies

The problem:

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

The fix:

```
    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, …) are more efficient if imperative

Try not to abuse your new-found power!

Upcoming events

- N/A

This is (reluctantly) imperative.

THIS IS 3110