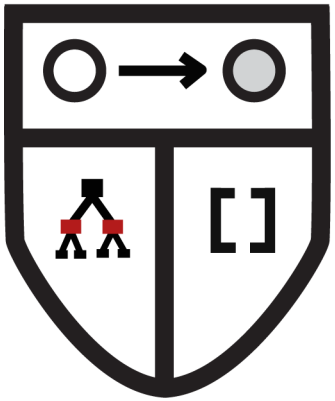


CS 3110

Mutable Data Types

A New Despair
Mutability Strikes Back
Return of Imperative Programming

Prof. Clarkson

Fall 2019

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

CLICKER QUESTIONS 1 AND 2

Review

Previously in 3110:

- Efficiency: Big Oh

Lectures 13-15:

- Efficiency of data structures
- Running example: maps

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`

CLICKER QUESTION 3

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

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`

MUTABLE FIELDS

Demo

Implementing refs

Ref cells are essentially implemented as records with a mutable field:

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

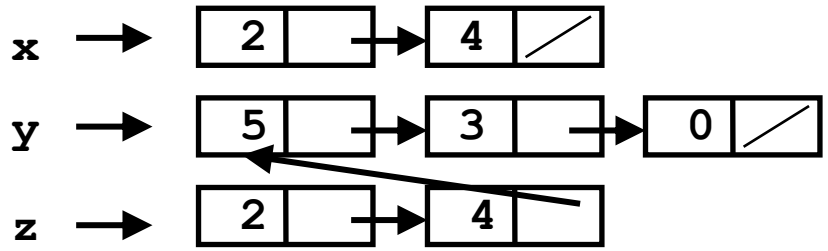
- That type is declared in **`Stdlib`**
- The functions are compiled down to something equivalent

BEWARE

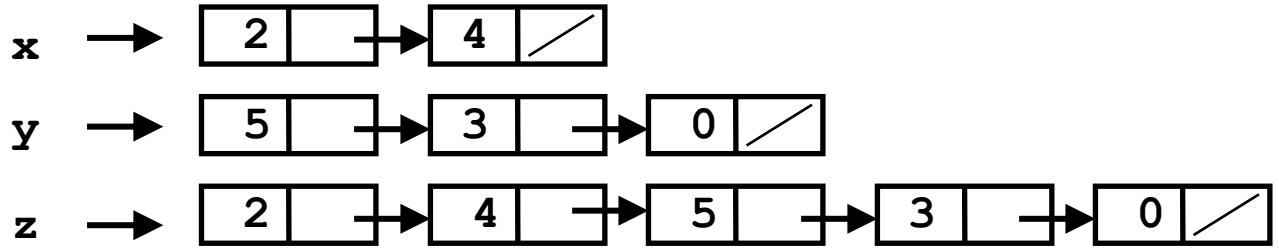
Immutable lists

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();
   }
}
```

Can you find the security fault?

Have to make copies

The exploit:

```
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

Pros and cons of immutability

Pros:

- 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

Cons:

- I/O is fundamentally about mutation
- Some data abstractions (dictionaries, arrays, …) are more efficient if imperative

Try not to abuse your new-found power!

Upcoming events

- [last night] R5 due
- [Wed] A4 due

This is (reluctantly) imperative.

THIS IS 3110