CS 3110

Lecture 12: Imperative features

Prof. Clarkson

Fall 2014

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

Review

Recently:

- Programming in the large
  - Modules, signatures, functors
  - Modularity, abstraction, specification
  - Many data abstractions (stacks, queues, dictionaries, …)

Today: THE DARK SIDE ARRIVES

- Imperative features: refs, arrays, mutable fields

Question #1

How much of PS3 have you finished?

A. None
B. About 25%
C. About 50%
D. About 75%
E. I'm done!!!

Question #2

What's your opinion of *Episode V*?

A. Great movie
B. The greatest movie
C. I've never watched it
D. I'm not a sci-fi fan

Question #2

What's your opinion of *Episode V*?

A. Great movie

B. The greatest movie

C. I've never watched it

D. I'm not a sci-fi fan

Prelim 1

- One week from today
- Covers everything from Aug 26 through Oct 1 (inclusive)
  - People with Thursday recitations, note that today's recitation is included
- Sample prelim posted on Piazza
- Review session in recitation day before prelim
- Cancel lecture on day of prelim
- You can take prelim at your choice of 5:30-7:00 pm or 7:30-9:00 pm; no need to reserve in advance
- Three rooms, will be assigned by netid next wek
- Closed book
  - But you may have one page of notes
  - 8.5x11" two-sided ☺

Question #3

Which prelim do you think you will attend?

A. 5:30 pm

B. 7:30 pm

I'm just curious—you are not committing to anything.

News about PS1

- Relax, your grade is not going to change because of this news
- Some groups got a very slightly higher grade on PS1 than they should have
  - About 60 groups out of 150
  - But not that much higher…an average of just 1 point out of 160 across all groups
- We will upload your correct autograder feedback as a new comment in CMS for PS1 later today
  - You need to know as you study for Prelim 1!
- But we will not lower your PS1 grade
- I apologize for the accidental bonus…won't happen again ☺

News about PS1

Object lesson in having tests suites and running regression tests

- Autograder bug discovered while we graded PS2
  - (That's why it took so long... we did a lot of manual validation before releasing PS2 grades. And we will continue to manually grade a random sample.)
- Bug caused autograder to think you passed a test case when you actually failed
  - Was introduced over the summer and was isolated to a single regular expression match
  - Nobody got a lower grade than they should have because of this bug
- Course staff should be practicing what I preach!

IMPERATIVE FEATURES

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, 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*...

References

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

```
let x = ref 0
let y = !x (* y bound to 0 *)
x := 1
(* could write let _ = x := 1 for uniformity *)
let z = !x (* z bound to 1 *)
(* x + 1 does not type-check *)
```

- 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

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

- **`next_val ()`** returns **1**
- then **`next_val ()`** returns **2**
- then **`next_val ()`** returns **3**
- etc.

Implementing a counter

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

somewhat better style:

```
let counter = ref 0
let next_val : unit -> int = fun () ->
  begin
     counter := (!counter) + 1;
     !counter
  end
```

References

- **Syntax: `ref e`**
- **Evaluation:**
  - Evaluate `e` to a value `v`
  - Allocate a new *location* in memory to hold `v`
  - Store `v` there
  - Return that location
  - Note: 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`

Evaluation semantics with refs

- Reverting back to *substitution model*
  - There is a global memory called the *heap* mapping locations to values
  - Evaluation order matters!
- Could give *environment model* semantics, too
  - Would need to write something like

```
env, heap :: e --> v :: heap'
```

  - The final **`heap'`** reflects any side effects

References

- **Syntax: `e1 := e2`**
- **Evaluation:**
  - Evaluate `e2` to a value `v2`
  - Evaluate `e1` to a location `v1`
  - Store `v2` in location `v1`
  - Return `()`
- **Type checking:**
  - If `e2 : t`
  - and `e1 : t ref`
  - then `e1:=e2 : unit`

References

- **Syntax: `!e`**
  - note: not negation
- **Evaluation:**
  - Evaluate `e` to a location `v`
  - Return the contents of location `v`
- **Type checking:**
  - If `e : t ref`
  - then `!e : t`

References

- **Syntax: `e1; e2`**
- **Evaluation:**
  - evaluate `e1` to a value `v1`, then forget about 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`
- Useful function from `Pervasives`:
  `ignore : 'a -> unit`
  Evaluates its argument then returns `()`

Implementing semicolon

Essentially syntactic sugar:

```
e1; e2
(* means the same as *)
let _ = e1 in e2
```

Except that type checker gives a warning if type of **`e1`** is not **`unit`** in the semicolon syntax

Aliases

- Mini-review:
  - A variable bound to a reference is immutable: it will always be bound to the same reference
  - But the contents of the reference may be changed by **:=**
- And there may be **aliases** to the reference

Question #4

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. zardoz

Question #4

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. zardoz

Equality

- Single equals is *structural equality*
  - `(ref 3110) = (ref 3110)`
  - `[1;2;3] = [1;2;3]`
  - `2 <> 3`
- Double equals is *physical equality*
  - `let r1 = ref 3110`
  - `let r2 = ref 3110`
  - `r1 == r1`
  - `r1 != r2`

Beware

"You don't know the power of the dark side"

Immutability is a valuable non-feature

might seem weird that lack of feature is valuable...

Suppose OCaml had mutable pairs…

```
let x = (4,3)
let y = sort_pair x

(* somehow mutate fst x to be 5 *)

let z = fst y
```

What is **z**?

- Would depend on how we implemented **sort_pair**
  - Would have to decide carefully and document **sort_pair**
- But without mutability…
  - No code can ever distinguish aliasing vs. copying
  - Programmer has no need to think about aliasing
  - Run-time can use aliasing, which saves space, without danger

No need to think about aliasing…

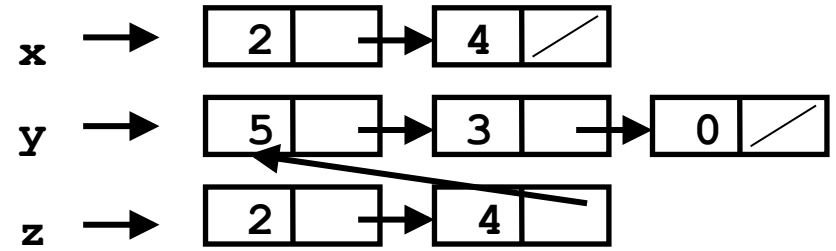
```
let sort_pair (pr : int * int) =
  if fst pr > snd pr
  then pr
  else (snd pr, fst pr)

let sort_pair (pr : int * int) =
  if fst pr > snd pr
  then (fst pr, snd pr)
  else (snd pr, fst pr)
```

In OCaml, these two implementations of **`sort_pair`** are indistinguishable

- But only because tuples are immutable
- The first is better style: simpler and avoids making a new pair in the then-branch
- In Java, you make copies like the second one all the time to avoid aliasing

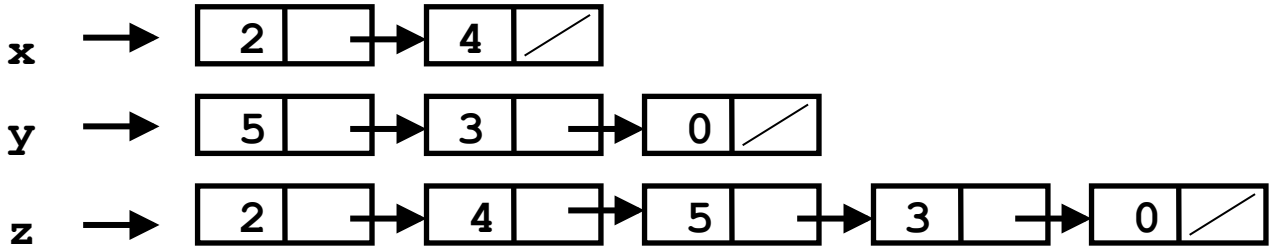
No need to think about aliasing…

```
let rec append lst1 lst2 =
  match lst1 with
     [] -> lst2
   | h::t -> h :: (append t lst2)
let x = [2;4]
let y = [5;3;0]
let z = append x y
```



(no code can tell, but run-time uses the first one)

or



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[] 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:

```
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 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!

Additional imperative features

- Arrays
- Mutable fields
- Control structures (**`while`** and **`for`** loops)
  - Not themselves imperative but mostly used in conjunction with imperative features

Arrays

Arrays generalize ref cells from a single mutable value to a sequence of mutable values

```
[|e1; ...; en|]
```

- evaluates to an `n`-element array, whose elements are initialized to `v1...vn`, where `e1-->v1, ..., en-->vn`
- `[|e1; ...; en|] : t array` if each `ei : t`

Arrays

`e1.(e2)`

- 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`.
- `e1.(e2) : t` if `e1 : t array` and `e2 : int`

Arrays

`e1.(e2) <- e3`

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

```
# 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

Control structures

Traditional loop structures are useful with imperative features:

- `while e1 do e2 done`
- `for id=e1 to e2 do e3 done`
- `for id=e1 downto e2 do e3 done`

Read the manual for (the obvious) semantics...

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Upcoming events

- **PS3 due tonight**
- **PS4 released next week**
- **Prelim 1 is in one week**

This is imperative.

THIS IS 3110