Programming Languages and Techniques (CIS120e)

Lecture 13
Oct 4, 2010

References

When to Use Mutable State

- action at a distance
  - allow remote parts of a program to communicate / share information without threading the information through all the points in between
- direct manipulation of hardware (device drivers, etc.)
- random-access data (arrays)
- data structures with explicit sharing
  - e.g. graphs
  - (without mutation, it is only possible to build trees – no cycles)
- efficiency
  - a few data structures have imperative versions with better asymptotic efficiency than the best declarative version

Mutable References in OCaml

OCaml offers essentially the same imperative features as other languages, but it packages them in a bit more explicit way: the type “reference cell holding an int” is different from the type int, and operations like reference creation and dereferencing are visible in the program’s syntax.

```ocaml
let r : int ref = ref 0 in
r := !r + 40;
r := !r + 2;
!r
```
The unit Type

- The unique value of type `unit` is written `()`
- `unit` is the result type of assignments

```
let r = ref 0
let u : unit = (r := 5)
```

- The `;` in `e1;e2` is a `binary operator`: it takes two expressions, `e1` and `e2`, and yields an expression that: evaluates `e1`, throws away the result, and then evaluates `e2`
- The type of `e1` must be `unit`

```
let x : int = (r := 5; !r)
```

Uses of unit

- As the result of functions that are called only for their side-effects
- As an argument of functions that don’t actually need to be passed any real data values

```
let r = ref 0
let s = ref 0

let f (x:int) : unit = r := !r + x

let tick () : int = s := !s + 1; !s
```

Action at a Distance

```
let r = ref 0
let f (x:int) : unit = 
  r := !r + x
;; f(5)
;; f(10)
;; assert_eq "" (!r) 15
```

Aliasing

What does this function return?

```
let f (r1:int ref) (r2:int ref) : int = 
  r1 := 5;
  r2 := 42;
  !r1
```

r is not mentioned in the signature of `f` (which is just `int->unit`), but it changes every time `f` is called
**Aliasing**

Are you sure?

```ocaml
define f (r1:int ref) (r2:int ref) : int =
    r1 := 5;
    r2 := 42;
    !r1
let r = ref 0
let x = f r r
```

**Hidden State**

```ocaml
define newcounter (n:int) : int->int =
    let r = ref n in
    fun (x:int) -> (r := !r + x; !r)
```

result of newcounter is a function that, when called, adds its argument into the reference cell \( r \)

**Workspace**  **Stack**  **Heap**

```ocaml
define newcounter (n:int) : int->int =
    let r = ref n in
    fun (x:int) ->
        (r := !r + x; !r)
    in
    newcounter 5
```
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in
append a b

newcounter

let r = ref n in
fun (x:int) ->
(r := !r + x; !r)

fun (x:int) ->
(r := !r + x; !r)
A Similar Example

```ocaml
let c1 = newcounter 0
let c2 = newcounter 0
;; c1 30
;; c2 10
;; c1 2
;; assert_eq "" (c1 0) ;;
;; assert_eq "" (c2 0) ;;
```

since the variable r is needed in the body of this function, we save its value (which is about to get popped off the stack) here

```ocaml
let newcounter (n:int) : int->int =
  let r = ref n in
  Fun (x:int) ->
    (r := !r + x; !r)
```
let x1 = c1 30 in
let x2 = c2 10 in
c1 2

let c = ref 0
let up () : int =
c := !c + 1;
!c
let down () : int =
c := !c - 1;
!c
**Shared State**

Or even...

```ocaml
let new_updown () : (unit->int) * (unit->int) =
  let c = ref 0 in
  let up () : int =
    c := !c + 1;
    !c in
  let down () : int =
    c := !c - 1;
    !c in
  (up,down)
```

**Digression: Records**

The last example is a little clumsy: it returns a pair of functions, but the caller has to remember which one goes up and which goes down — a bug waiting to happen!

```ocaml
let new_updown () : (unit->int) * (unit->int) =
  let c = ref 0 in
  let up () : int =
    c := !c + 1;
    !c in
  let down () : int =
    c := !c - 1;
    !c in
  (up,down)

let (mydown,myup) = new_updown()

let r1 = myup()
let r2 = myup()

;; assert_eq "uptwice" r2 2
```

---

**Digression: Records**

A better way is to return a record of functions, so that the names are carried along with the functions:

```ocaml
type updown = {up: unit->int; down: unit->int}

let new_updown () : updown =
  let c = ref 0 in
  let u () : int =
    c := !c + 1;
    !c in
  let d () : int =
    c := !c - 1;
    !c in
  {up=u; down=d}

;; let ud = new_updown() in
  assert_eq "ud1" (ud.up()) 1;
  assert_eq "ud2" (ud.up()) 2;
  assert_eq "ud3" (ud.down()) 1;
  assert_eq "ud4" (ud.down())
```