Benjamin Fenster
CS-103 (Skalka)
9 October 2003

Mutation and State

I. Reference Cells
   a. Most basic mutable entity
   b. Essentially a memory location, similar to a pointer but completely transparent.
   c. Example
      i. let x = ref 1;;
      ii. !x (* dereference *)
      iii. x $\Downarrow$ {mutable contents = 1}
      iv. x := 2 (* update, distinct from = *)
      v. !x $\Downarrow$ 2
   d. No pointer arithmetic "or any nasty tricks like that"
   e. Typing Rules
      i. Creation: ref e : τ ref iff e : τ
      ii. Dereferencing: !e : τ iff e : τ ref
      iii. Assignments: e := e' : unit iff e : τ ref and e' : τ
         1. Everything has a type. The choice of unit is arbitrary
         2. The purpose of () is as a placeholder. Anything that has a unit type generally signals the programmer that whatever is happening is side effectual.
   f. Evaluation Rules
      i. Creation: ref e $\Downarrow$ c, where c is a reference cell, with v stored at c, where e $\Downarrow$ v
      ii. Dereferencing: !e $\Downarrow$ v iff e $\Downarrow$ c and v is currently stored at c
      iii. Assignment
         1. e := e' $\Downarrow$ () iff e $\Downarrow$ c
         2. Side effect: Contents of c updated with v iff e' $\Downarrow$ v
         3. In general, side effect is some effect of computation that's distinct from evaluation.
   g. Example
      i. let x = ref 10
      ii. x = 10 NO! Type error (int = int ref)
      iii. Aliasing
         1. let y = ref x;;
         2. let z = ref x;;
         3. (!y) := 0
         4. !(!z) $\Downarrow$ 0
         5. !x $\Downarrow$ 0
         6. "Mutation leads to aliasing, aliasing leads to pain."

II. Sequencing and Evaluation Order
   a. Now the order in which statements are evaluated matters.
   b. Sequencing: e_1; e_2 (first e_1 then e_2)
   c. e_1; e_2 : τ iff e_1 : τ' and e_2 : τ
   d. Evaluation
      i. e_1; e_2 $\Downarrow$ v iff e_1 $\Downarrow$ v' and e_2 $\Downarrow$ v (in that order) with implicit side-effects
      ii. Value of e_1 is essentially thrown away!
      iii. If e_1 does not have type unit, the compiler reports a warning.
   e. Example
      i. let x = ref 0;;
      ii. x := 1; x := 2; x := 3; !x;; $\Downarrow$ 3
      iii. type r = {a : unit; b : unit}
      iv. {a = (x := 1); b = (x := 2) }
      v. !x $\Downarrow$ 1
   f. Example
      i. (x := 1; (fun x -> x))(x := 2; 1) $\Downarrow$ 1
      ii. !x $\Downarrow$ 2

III. Mutable Records
  a. Slightly more complex form
  b. type mutrec = {mutable a : int; b : int};;
  c. let mr = {a = 1; b = 2}
  d. mr.a ⇓ 1
  e. mr.a <- 3
  f. mr.a ⇓ 3
  g. mr.b <- 5 NO!
  h. Reference cells are really just syntactic sugar for mutable records with one field.
    i. type ‘a ref = {mutable contents : ‘a}
    ii. ref e   {contents = e}
    iii. !e ⇓ e.contents
    iv. e := e’ ⇓ e.contents <- e’
    v. # let a = ref 5;;
    vi. – val a : int ref = {contents = 5}

IV. The Values Restriction
  a. Example
    i. let x = ref (fun x -> x)
    ii. x : (‘a -> ‘a) ref
    iii. let f y = (!x)y : ‘a -> ‘a
    iv. (Remember static typing!)
    v. x := (fun x -> x + 1)
    vi. f(“uh-oh”) still *looks* valid but no longer makes sense!
  b. This has been a major topic for years
  c. Andrew Wright in 1995 proposed a solution known as “The Values Restriction”
    i. Only values can have polymorphic types.
    ii. let x = (fun x -> x : ‘a -> ‘a
    iii. let x = ref (fun x -> x)
      1. In standard ML this won’t be allowed.
      2. In OCaml it gets type (_’a -> _’a)
      3. This is a placeholder that will be filled in once it gets used for the first time.
    iv. let f y = (!x) y : _’a -> _’a)
    v. f 1 ⇓ 1
      1. Now the type of f is (int -> int)
      2. The type of x is (int -> int)
    vi. f(“uh-oh”) NO! Not well typed!
  d. This is a simple solution.
    i. Many more complex ideas were suggested.
    ii. Ultimately a compiler was written to implement this solution and millions of lines of existing code were compiled.
    iii. Only a few places had problems and those problems were easily fixed.
    iv. The “restriction” didn’t seem to restrict normal / correct use of the language, so it was adopted.
  e. Eta-Conversion
    i. Example
      1. let duplicate = map (fun x -> (x, x))
      2. map : (‘a -> ‘b) -> ‘a list -> ‘b list
      3. duplicate : _’a list -> _’a * _’a list
      4. It’s no longer polymorphic!
    ii. The way around that problem is to wrap it in a function.
      1. let dupliate = (fun x -> map (fun x -> (x, x)))
      2. This is equivalent, but it’s wrapped in a lambda abstraction so now it’s a value (not an application) and the values restriction doesn’t apply.

V. Why Have Mutation?

a. Purists say “no reason!”
b. let rec fibnum x = match x with 0 -> 1 | 1 -> 1 | x -> fibnum(x – 1) + fibnum(x – 2)
c. This is a naïve implementation – it’s extremely exponential.
d. Memoization: Uses mutation / mutable reference cells to store and lookup values previously computed. Now the function becomes linear.

ERROR: undefinedfilename
OFFENDING COMMAND: </FONT>

STACK: