Computer Science 131
Programming Languages

September 28, 2000

Side-Effects: Assignment

Assignment

- A mutable (assignable) variable has two attributes
  - a location
  - its current contents
- In most languages you are familiar with, context determines which is meant.

```
x = x + 1;
```

Assignment

- A mutable (assignable) variable has two attributes
  - a location
  - its current contents
- In most languages you are familiar with, context determines which is meant.

```
x = x + 1;
```

the address of **x**

the contents of **x**

Terminology

- l-value: an assignable location
- r-value: a value which can be assigned

x = x + 1;

the l-value

the r-value

- l-values can be more general than just variables

```
x->foo[3] = x->foo[2] + 1          (C or C++)
(if i>4 then x else y) := 7        (SML)
```

"Mutable" Variables in SML

- New mutable locations are allocated with **`ref`**

```
val x = ref 0       (* new mutable location, initially 0 *)
val y = ref 0       (* different location, initially 0 *)
val z = ref "hello" (* third location *)
```

- Appearances of **`x`** or **`y`** *always* denote the l-value
  - Enforced by the type system

```
x : int ref              (* x is not an integer *)
y : int ref
z : string ref
```

Dereferencing

- To coerce l-values to r-values, use the contents-of operator, **`!`**

```
val x = ref 0        (* mutable location w/ initial value 0 *)
val y = ref 0        (* different location w/ initial value 0 *)
val z = ref "hello"  (* third location, w/ this string *)

!x                   (* evaluates to 0 *)
!x + !y              (* evaluates to 0+0 = 0 *)
!x + size(!z)        (* evaluates to 0+5 = 5 *)
```

Assignment

- In SML the assignment operator is **`:=`**

```
val x = ref 0         (* mutable location w/ initial value 0 *)
val y = ref 0         (* different location w/ initial value 0 *)
val z = ref "hello"   (* third location, w/ this string *)

x := 3;               (* sets the location given by x to 3 *)
x := !x + 1;          (*  sets the location given by x to 4 *)
z := "bye";           (* changes string in loc. given by z *)

!x + size(!z)         (* evaluates to 4+3 = 7 *)
```

Variables Still Don't Vary!

- After the assignment

```
x := 3
```

  the variable **x** has not changed!

Variables Still Don't Vary!

- After the assignment

  `x := 3`

  the variable **x** has not changed!

- The variable **x** still represents the same location.
- What may have changed is the *value* at the location stored in **x**
  - that is, **`!x`** is now 3.

SML Typing

- The types of the reference operations are:

```
ref    : 'a            -> 'a ref
!      : 'a ref        -> 'a
:=     : 'a ref * 'a -> unit
```

(assignment is written infix)

Equality

- In SML, two references of the same type can also be compared for equality
  - Do these two references refer to the *same* piece of mutable storage?
  - Under the hood, pointer equality.

Aliasing

- Two expressions denoting the same l-value are said *to alias* or *to be aliases*

```
val x = ref 0
val y = ref 0
val z = x
```

- What do the following evaluate to?

```
!x
!y
!x = !y
x = y
x = z
```

Aliasing

- Two expressions denoting the same l-value are said *to alias* or *to be aliases*

```
val x = ref 0
val y = ref 0
val z = x
```

- What do the following evaluate to?

```
!x         (* evaluates to 0 *)
!y         (* evaluates to 0 *)
!x = !y    (* evaluates to true *)
x = y      (* evaluates to false *)
x = z      (* evaluates to true *)
```

Aliasing

- Two expressions denoting the same l-value are said *to alias* or *to be aliases*

```
val x = ref 0
val y = ref 0
val z = x
x := 1;
```

- What do the following evaluate to?

```
!x
x = y
x = z
!z
!y
```

Aliasing

- Two expressions denoting the same l-value are said *to alias* or *to be aliases*

```
val x = ref 0
val y = ref 0
val z = x
x := 1;
```

- What do the following evaluate to?

```
!x          (* evaluates to 1 *)
x = y       (* evaluates to false *)
x = z       (* evaluates to true *)
!z          (* evaluates to 1 *)
!y          (* evaluates to 0 *)
```

A Counter

```
local
   val count = ref 0
in
   fun reset() = (count := 0)
   fun inc()   = (count := !count + 1;
                       !count)
end
```

A Counter Generator

```
fun make_counter() =
   let
      val count = ref 0
      fun reset() = (count := 0)
      fun inc()   = (count := !count + 1;
                               !count)
   in (reset,inc) end

val (reset1, inc1) = make_counter()
val (reset2, inc2) = make_counter()
```

Loops Without Recursion

```
val fref : (int->int) ref =
  ref (fn x => x)

val fact : int->int =
  (fn n => if (n=0) then 1
           else n * (!fref)(n-1))
```

Loops Without Recursion

```
val fref : (int->int) ref =
  ref (fn x => x)

val fact : int->int =
  (fn n => if (n=0) then 1
           else n * (!fref)(n-1))
```

What is **`fact(0)`**? How about **`fact(2)`**?

Loops Without Recursion

```
val fref : (int->int) ref =
  ref (fn x => x)

val fact : int->int =
  (fn n => if (n=0) then 1
           else n * (!fref)(n-1))

fref := fact
```

Loops Without Recursion

```
val fref : (int->int) ref =
  ref (fn x => x)

val fact : int->int =
  (fn n => if (n=0) then 1
           else n * (!fref)(n-1))

fref := fact
```

Now what is **`fact(0)`**? How about **`fact(2)`**?

Adding References to NQSML

- So far we have used purely *language-based* models of program execution
  - Every intermediate state of a program can be represented as another program
  - Advantages:
    - Relatively direct
    - Don't have to worry about irrelevant machine details (memory layout, data representations)

Adding References to NQSML

- Now we need to maintain a notion of memory, memory locations, aliasing, etc.
  - Do we need to start thinking about bits, bytes, and data representation?
  - Or is there a middle ground?

An Abstract Notion of Memory

- Postulate an (infinite) set of *locations*
  - Denoted l_1, l_2, l_3, …
  - Each location can hold a single value
    - An integer, or a function, or a pair of values, …
- Memory is a kind of "environment"
  - Associates locations with the values they contain
  - For example,
    - l_1**=<u>3</u>**, l_6 **=(<u>3</u>,<u>4</u>)**, l_{17} **=(<u>3</u>,(<u>4</u>,(<u>5</u>,<u>tt</u>)))**

Programs and Expressions

- We extend the notion of expressions by adding unit and locations as new values

```
v ::= ...
    | ()
    | l
```

and by adding three new expression forms

```
e ::= ...
    | mkref e
    | get e
    | set(e1,e2)
```

Example Expressions

```
let r be mkref(0) in
    let y = get(r) in
         set(x, y+1)
```

set(l_3 **, get(**l_{12}**))**

Programs

- A program then consists of
  - a memory M (often called a "store")
  - an expression **e**

```
p ::= (M, e)
```

- Recall that M associates values with locations
- Evaluation is defined on *programs*

$p \rightarrow p'$ $\qquad\qquad$ $(M,e) \rightarrow (M',e')$