Assignment and Aliasing

October 11, 2001

CS 131: Programming Languages

Enter Side-Effects

- When introducing SML, I noted that expressions in ML may return a value, and may have a *side-effect*.
- Today's topics:
  - Assignment in SML
  - Extending the interpreter (without *using* assignment)

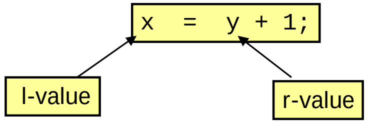
Assignment

- An *assignable* (or *mutable*) variable has two run-time attributes of importance:
  - its location (address)
  - the value it currently contains.
- In most imperative languages, context determines which the code is referring to at any point.

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

Terminology

- An *l-value* is an assignable location.
- An *r-value* is a value which can be stored.



- L-values appear on the left of an assignment, and r-values appear on the right.

Complex L-values

- In many languages, l-values can be more general than just variables

```
x->foo[3] = x->foo[2] + 1;
```

```
(if i>4 then x else y) := 7
```

Assignment 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 such variables always denotes 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

- The SML 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 *)
```

ML Variables Still Don't Vary!

- After this assignment the variable `x` has not changed:

```
x := 3
```

- The variable `x` still represents the same *location*.
- The *value* at the location stored in `x` (that is, `!x`) may have changed, however

Typechecking

- The types of these new SML operations are:

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

(where assignment is infix)

Aliasing

- In SML, any two references of the same type can be compared for equality with =
  - Asks whether the two references refer to the *same* piece of mutable storage?
- Two expressions denoting the same l-value are said *to alias* or *to be aliases.*
  - After

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

  `x` and `z` alias, but neither is an alias of `y`.

Exercise

- Consider the functions

```
f : int     -> int
g : int ref -> int ref
```

defined by

```
fun f(x:int) = !(ref x)
fun g(r:int ref) = ref(!r)
```

- Are either of these the identity function?
  - If so, which one?

Quick Quiz

```
val x1 : int list = [1,2,3]
val _ = f1(x1)
```

`length(x1) =` `hd(x1) =`

```
val x2 : int list ref = ref [1,2,3]
val _ = f2(x2)
```

`length(!x2) =` `hd(!x2) =`

```
val x3 : int ref list = [ref 1, ref 2, ref 3]
val _ = f3(x3)
```

`length(x3) =` `!hd(x3) =`

Aliasing in Other Languages

- Pointers are sufficient to create aliases

```
int  x = 3;
int* y = &x;
*y = 4;
```

but not necessary
- Implicit pointers (objects in Java)
- Call-by-reference (FORTRAN, C++, ...)

```
PROGRAM SAMPLE
INTEGER M
M = 2
Q(M,M)
PRINT *,M
END

SUBROUTINE Q(I,J)
INTEGER SIZE
I = I+1
J = J+1
RETURN
END
```

Pure vs. Imperative Interfaces

- Persistent environments

```
type 'a env
val empty : 'a env
val insert: 'a env * string * 'a -> 'a env
val lookup: 'a env * string -> 'a option
```

- Ephemeral environments

```
type 'a env
val empty : unit -> 'a env
val insert: 'a env * string * 'a -> unit
val lookup: 'a env * string -> 'a
val copy  : 'a env -> 'a env
```

why a function?

- NB: interface *suggests*, but does not *specify* the implementation.

A Counter

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

types?
```
reset :
check :
inc   :
```

Using This Counter

```
fun fib(n) =
  (inc();
   if (n=0) then 1
     else if (n=1) then 1
            else fib(n-1)+fib(n-2))

val x = (reset(); fib 5; check())
```

A Counter Generator

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

val (reset1, check1, inc1) = make_counter()
val (reset2, check2, 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))
```

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

fref := fact
```

Now what is `fact(0)`? How about `fact(1)`?

Interpreter Review

- When we last saw our interpreter, it used
  - *environments* for efficiency
  - *closures* to obtain static scope

Big-Step Semantics

$(\rho, M) \Downarrow V$

Corresponding Interpreter

```
seval : env * absyn -> absyn
```

Stores

- We will need to refer to memory locations
  - But, don't care about exact memory locations
  - Only need to *distinguish* different memory locations $\{l_1, l_2, l_3, \ldots\}$.
- A *store* is the abstraction of a program's memory.
  - Associates (arbitrary) values with locations.
  - Represented with the Greek letter σ

	`type store`
	`type loc`
$\varnothing$	`val sempty : store`
$\sigma(l)$	`val slookup : store*loc -> absyn`
$\sigma, l{=}V$	`val sextend : store*loc*absyn -> store`
	`val freshLoc : store -> loc`

Input and Output Stores

- The language specification now has rules of the form

$$(\rho, \sigma, M) \Downarrow (\sigma', V)$$

For example,

$$(\{x{=}l_4, y{=}l_{12}, z{=}7\}, \{l_4{=}1, l_{12}{=}2\}, !x+!y) \Downarrow (\{l_4{=}1, l_{12}{=}2\}, 3)$$

- The interpreter function changes similarly:

```
eval : env*store*absyn -> store*absyn
```

```
datatype absyn = ... | Loc of loc
               | Ref of absyn | Deref of absyn
               | Assign of absyn*absyn
```

Revised Interpreter

```
exception Error

fun eval(env,store,expr) =
   (case expr of
       Num n =>  (store,Num n)
     | Bool b => (store,Bool b)
     | Nil =>    (store,Nil)
     | Lam _ =>  (store,Closure(expr,env))

     | Var v =>  (store,lookup(env,v))
```

$$\frac{}{(\rho, \sigma, V) \Downarrow (\sigma, V)}$$

$$\frac{}{(\rho, \sigma, x) \Downarrow (\sigma, \rho(x))}$$

Revised Interpreter

```
| Plus(e1,e2) =>
   let
     val (store1,v1) = eval(env,store,e1)
     val (store2,v2) = eval(env,store1,e2)
   in
     (case (v1,v2) of
        (Num m1, Num m2)
           => (store2, Num (m1+m2))
      | _  => raise Error)
   end
```

$$\frac{(\rho,\sigma_1,M_2) \Downarrow (\sigma_2,n_2) \quad (\rho,\sigma,M_1) \Downarrow (\sigma_1,n_1)}{(\rho,\sigma,M_1+M_2) \Downarrow (\sigma_2,n_1 \oplus n_2)}$$

Revised Interpreter

```
| Equal(e1,e2) =>
   let
     val (store1,v1) = eval(env,store,e1)
     val (store2,v2) = eval(env,store1,e2)
   in
     (case (v1,v2) of
        (Num m1, Num m2)
           => (store2, Bool(m1=m2))
      | _  => raise Error)
   end
```

$$\frac{(\rho,\sigma,M_1) \Downarrow (\sigma_1,n_1) \quad (\rho,\sigma_1,M_2) \Downarrow (\sigma_2,n_2)}{(\rho,\sigma,M_1{=}{=}M_2) \Downarrow (\sigma_2,n_1 \equiv n_2)}$$

Conditional

```
| If(M,N1,N2) =>
   (case eval(env,store,M) of
      (store1,Bool true)  => eval(env,store1,N1)
    | (store1,Bool false) => eval(env,store1,N2)
    | _                   => raise Error)
```

$$\frac{(\rho,\sigma,M) \Downarrow (\sigma_1,\mathtt{tt}) \quad (\rho,\sigma_1,N_1) \Downarrow (\sigma_2,v)}{(\rho,\sigma,\mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2) \Downarrow (\sigma_2,v)}$$

$$\frac{(\rho,\sigma,M) \Downarrow (\sigma_1,\mathtt{ff}) \quad (\rho,\sigma_1,N_2) \Downarrow (\sigma_2,v)}{(\rho,\sigma,\mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2) \Downarrow (\sigma_2,v)}$$

Local Definitions

```
| Let(x,M,N) =>
   let
     val (store1,v1) = eval(env,store,M)
     val env' = extend(env,x,v1)
     val (store2,v2) = eval(env',store1,N)
   in
     (store2,v2)
   end
```

$$\frac{(\rho,\sigma,M) \Downarrow (\sigma_1,v_1) \quad ((\rho,x{=}v_1),\sigma_1,N) \Downarrow (\sigma_2,v_2)}{(\rho,\sigma,\mathtt{let}\ x{=}M\ \mathtt{in}\ N) \Downarrow (\sigma_2,v_2)}$$

Application

```
| Apply(M,N) =>
  let
    val (store1,v1) = eval(env,store,M)
    val (store2,v2) = eval(env,store1,N)
  in
   (case v1 of
      Closure(Lam(x,M'),env1) =>
       let
        val env' = extend(env1,x,v2)
       in
        eval(env',store2,M')
       end
    | _ => raise Error)
 end
```

$$\frac{(\rho,\sigma,M)\Downarrow(\sigma_1,\llbracket\lambda x.M',\rho_1\rrbracket) \quad (\rho,\sigma_1,N)\Downarrow(\sigma_2,v_2) \quad ((\rho,x{=}v_2),\sigma_2,M')\Downarrow(\sigma_3,v)}{(\rho,\sigma,M(N))\Downarrow(\sigma_3,v)}$$

Refs

```
| Ref(M) =>
  let
    val (store1,v) = eval(env,store,M)
    val loc = freshLoc(store1)
    val store2 = sextend(store1,loc,v)
  in
    (store2, Loc loc)
  end
```

$$\frac{(\rho,\sigma,M)\Downarrow(\sigma_1,v) \quad l\notin dom(\sigma_1)}{(\rho,\sigma,\mathtt{ref}\ M)\Downarrow((\sigma_1,l{=}v),l)}$$

Dereferencing

```
| Deref(M) =>
  let
    val (store1, v) = eval(env,store,M)
  in
    (case v of
       Loc loc => (store1, slookup(store1,loc))
     | _ => raise Error)
  end
```

$$\frac{(\rho,\sigma,M)\Downarrow(\sigma_1,l)}{(\rho,\sigma,!M)\Downarrow(\sigma_1,\sigma_1(l))}$$

Assignment

```
| Assign(M1,M2) =>
  let
    val (store1, v1) = eval(env,store,M1)
    val (store2, v2) = eval(env,store1,M2)
  in
    (case v1 of
       Loc loc => (sextend(store2,loc,v2), v2)
     | _ => raise Error)
  end
```

$$\frac{(\rho,\sigma,M_1)\Downarrow(\sigma_1,l) \quad (\rho,\sigma_1,M_2)\Downarrow(\sigma_2,v)}{(\rho,\sigma,M_1{:=}M_2)\Downarrow((\sigma_2,l{=}v),v)}$$