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.

x = y + 1;

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

```plaintext
```

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

Assignment in SML

• New mutable locations are allocated with `ref`

```plaintext
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
    ```plaintext
    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, `!

```plaintext
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 `:=`

```plaintext
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:
  \[
  \begin{align*}
  \text{ref} &: \ 'a \to \ 'a \text{ ref} \\
  ! &: \ 'a \text{ ref} \to \ 'a \\
  := &: \ 'a \text{ ref} \times \ 'a \to \text{ unit}
  \end{align*}
  \]
  (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
  \[
  \begin{align*}
  \text{val } x &= \text{ ref 0} \\
  \text{val } y &= \text{ ref 0} \\
  \text{val } z &= x
  \end{align*}
  \]
  x and z alias, but neither is an alias of y.

Exercise

- Consider the functions
  \[
  \begin{align*}
  \text{fun } f : \text{ int} \to \text{ int} \\
  \text{fun } g : \text{ int ref} \to \text{ int ref}
  \end{align*}
  \]
  defined by
  \[
  \begin{align*}
  \text{fun } f(x: \text{ int}) &= \!(\text{ref } x) \\
  \text{fun } g(r: \text{ int ref}) &= \text{ ref}(!r)
  \end{align*}
  \]
  - Are either of these the identity function?
    - If so, which one?
Quick Quiz

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

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

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

Aliasing in Other Languages

• Pointers are sufficient to create aliases
  int x = 3;
  int* y = &x;
  *y = 4;

 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

• 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

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

\texttt{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 \(\{l_1, l_2, \ldots\}\).
• A store is the abstraction of a program’s memory.
  – Associates (arbitrary) values with locations.
  – Represented with the Greek letter \(\sigma\).

\texttt{type store}

\texttt{type loc}

\texttt{\emptyset : store}

\texttt{\sigma(l) : store*loc -> absyn}

\texttt{\sigma[l=V] : store*loc*absyn -> store}

\texttt{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,

\[(\{x=1, y=2, z=3\}, \{l_1=1, l_{12}=2\}, !x+!y) \Downarrow (l_1=1, l_{12}=2), 3]\]

• The interpreter function changes similarly:

\texttt{fun eval(env, store, expr) =}

\texttt{  \{case expr of}

\texttt{    Num n => (store, Num n)}

\texttt{    Bool b => (store, Bool b)}

\texttt{    Nil => (store, Nil)}

\texttt{    Lam _ => (store, Closure(expr, env))}

\texttt{    Var v => (store, lookup(env, v))\}

\texttt{datatype absyn = ... | Loc of loc | Ref of absyn | Deref of absyn | Assign of absyn*absyn}

Revised Interpreter
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 |

\[(\rho, \sigma_1, M_1) \downarrow (\sigma_j, n_3) \quad (\rho, \sigma_1, M_1) \downarrow (\sigma_i, n_j) \quad (\rho, \sigma, M_1 + M_2) \downarrow (\sigma_j, n_1 \oplus n_2)\]

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 |

\[(\rho, \sigma, M_1) \downarrow (\sigma_1, n_1) \quad (\rho, \sigma_1, M_2) \downarrow (\sigma_j, n_2) \quad (\rho, \sigma, M_1 == M_2) \downarrow (\sigma_j, n_1 \equiv n_2)\]

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) |

\[(\rho, \sigma, M) \downarrow (\sigma_j, v) \quad (\rho, \sigma, if M then N_1 else N_2) \downarrow (\sigma_j, v) \quad (\rho, \sigma, if N then N_1 else N_2) \downarrow (\sigma_j, v)\]

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 |

\[(\rho, \sigma, M) \downarrow (\sigma_j, v) \quad (\rho, \sigma, if x=M in N) \downarrow (\sigma_j, v)\]
Application

\[ \text{Apply}(M, N) \Rightarrow \]
\[ \begin{align*}
&\text{let} \\
&\text{val } (\text{store}_1, v_1) = \text{eval}(\text{env}, \text{store}, M) \\
&\text{val } (\text{store}_2, v_2) = \text{eval}(\text{env}, \text{store}_1, N) \\
&\text{in} \\
&\text{case } v_1 \text{ of} \\
&\quad \text{Closure}(\text{Lam}(x, M'), \text{env}_1) \Rightarrow \\
&\quad \text{let} \\
&\quad \text{val } \text{env}' = \text{extend}(\text{env}_1, x, v_2) \\
&\quad \text{in} \\
&\quad \text{eval}(\text{env}', \text{store}_2, M') \\
&\quad \text{end} \\
&\ | \_ \Rightarrow \text{raise Error} \\
\end{align*} \]

\[
\begin{array}{c}
(p, \sigma, M) \Downarrow (\sigma, v) \\
\sigma_1 \Downarrow (\sigma_2, v) \\
\sigma_2 \Downarrow (\sigma_3, \rho)
\end{array}
\]

Refs

\[ \text{Ref}(M) \Rightarrow \]
\[ \begin{align*}
&\text{let} \\
&\text{val } (\text{store}_1, v) = \text{eval}(\text{env}, \text{store}, M) \\
&\text{val } \text{loc} = \text{freshLoc}(\text{store}_1) \\
&\text{val } \text{store}_2 = \text{sextend}(\text{store}_1, \text{loc}, v) \\
&\text{in} \\
&\text{(store}_2, \text{Loc loc}) \\
&\text{end}
\end{align*} \]

\[
\begin{array}{c}
(p, \sigma, M) \Downarrow (\sigma, v) \\
\text{let } \text{dom}(\sigma)_1
\end{array}
\]

\[ (p, \sigma, \text{ref } M) \Downarrow ((\sigma, l=v), l) \]

Dereferencing

\[ \text{Deref}(M) \Rightarrow \]
\[ \begin{align*}
&\text{let} \\
&\text{val } (\text{store}_1, v) = \text{eval}(\text{env}, \text{store}, M) \\
&\text{in} \\
&\text{case } v \text{ of} \\
&\quad \text{Loc } \text{loc} \Rightarrow (\text{store}_1, \text{slookup}(\text{store}_1, \text{loc})) \\
&\quad \_ \Rightarrow \text{raise Error} \\
\end{align*} \]

\[
\begin{array}{c}
(p, \sigma, M) \Downarrow (\sigma, l) \\
(p, \sigma, M) \Downarrow (\sigma_1, l_1)
\end{array}
\]

Assignment

\[ \text{Assign}(M_1, M_2) \Rightarrow \]
\[ \begin{align*}
&\text{let} \\
&\text{val } (\text{store}_1, v_1) = \text{eval}(\text{env}, \text{store}, M_1) \\
&\text{val } (\text{store}_2, v_2) = \text{eval}(\text{env}, \text{store}_1, M_2) \\
&\text{in} \\
&\text{case } v_1 \text{ of} \\
&\quad \text{Loc } \text{loc} \Rightarrow (\text{sextend}(\text{store}_1, \text{loc}, v_2), v_2) \\
&\quad \_ \Rightarrow \text{raise Error} \\
\end{align*} \]

\[
\begin{array}{c}
(p, \sigma, M_1) \Downarrow (\sigma_1, l) \\
(p, \sigma, M_1) \Downarrow (\sigma_2, \rho)
\end{array}
\]

\[
\begin{array}{c}
(p, \sigma, M_1; \! = \! M_2) \Downarrow ((\sigma_2, l=v), v)
\end{array}
\]