Aliasing and Assignment

March 5, 2001
CS 131: Programming Languages

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.

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

Terminology

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

```plaintext
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
```
Assignable Variables in SML

- New mutable locations are allocated with ref:
  
  ```sml
  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
  
  ```sml
  x : int ref (* x is not an integer! *)
  y : int ref
  z : string ref
  ```

Assignment

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

Dereferencing

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

ML Variables Still Don't Vary!

- After this assignment the variable `x` has not changed:
  
  ```sml
  x := 3
  ```

- The variable `x` still represents the same location:
  - that is, the same l-value

- What may have changed is the value at the location stored in `x`:
  - that is, `!x` is now 3.
Typechecking

• The types of these new SML operations are:

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

(where assignment is infix)

Equality

• In SML, any two references of the same type can be compared for equality
  – Do these two references refer to the same piece of mutable storage?

• Under the hood, implements pointer equality.

Aliasing

• 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
```

what do the following evaluate to?

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

• Which would change if we had first done x := 1 ?

Problem

• 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(!x)
```

• Are either of these the identity function?
**Quick Quiz**

```ml
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 = [ref1, ref2, ref3]
val _ = f3(x3)
  length(x3) = ?  !hd(x3) = ?
```

**Remarks on Formal Semantics**

- Recall the Simple Imperative Language:
  - Program states of the form \((M,e)\) where the memory \(M\) mapped variables to values.
  - Fine, as long as the language has no aliasing.
- To describe aliasing, need layer of indirection
  - Memory as mapping from locations to values
  - We further associate a location with each variable
    - E.g., maintain an environment mapping variables to locations in addition to the memory.

**Aliasing in Other Languages**

- Any language with pointers permits aliasing.

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

- But pointers are not required to have aliasing
  - Objects in Java
  - Call-by-reference (FORTRAN, C++, ...)

**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 -> 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 : ?

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()

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())

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

```ml
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)?

Assign-Once Variables

• Specification:

```ml
type 'a oneshot
exception Oneshot
val new : unit -> 'a oneshot
val get : 'a oneshot -> 'a
val set : 'a oneshot * 'a -> unit
```

Assign-Once Variables

```ml
type 'a oneshot = 'a option ref
exception Oneshot
```