

# Assignment and Aliasing

February 20, 2002

CS 131: Programming Languages

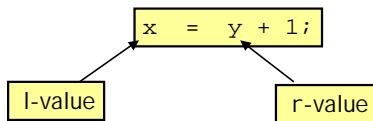
## Assignment

- 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.

```
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 *)
```
- In SML, appearances of such variables always denote the *l-value*
  - Enforced by the type system

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

## 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 *)
(x := 7; !x)      (* gives 7 since ; acts like , in C/C++ *)
```

## 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 *)
```

## 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)

## 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?

## 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.

## 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
```

## A Counter

```
val count = ref 0

fun reset() = (count := 0)
fun check() = !count
fun inc() = (count := !count + 1;
           !count)
```

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()
```

## Another Implementation of 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)`?

## Another Implementation of 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)`?

## 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 behavior of the implementation.