Assignment and Aliasing

February 20, 2002
CS 131: Programming Languages

Assignment

- An assignable (or mutable) variable has two runtime 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


\[ (if \ i > 4 \ then \ x \ else \ y) := ? \]
Assignment 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 *)
  ```

- In SML, appearances of such variables 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

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

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.
  - The value at the location stored in `x` (that is, `!x`) may have changed, however
Typechecking

- The types of these new SML operations are:

\[
\begin{align*}
\text{ref} & : \ 'a \rightarrow \ 'a \ \text{ref} \\
! & : \ 'a \ \text{ref} \rightarrow \ 'a \\
:= & : \ 'a \ \text{ref} \times \ 'a \rightarrow \ \text{unit} \\
\end{align*}
\]

(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

\[
\begin{align*}
f & : \text{int} \rightarrow \text{int} \\
g & : \text{int} \ \text{ref} \rightarrow \text{int} \ \text{ref}
\end{align*}
\]

defined by

\[
\begin{align*}
\text{fun } f(x: \text{int}) &= !(\text{ref } x) \\
\text{fun } g(r: \text{int} \ \text{ref}) &= \text{ref}(!r)
\end{align*}
\]

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

Quick Quiz

\[
\begin{align*}
\text{val } x1 : \text{int} \ \text{list} &= [1,2,3] \\
\text{val } _1 &= f1(x1) \\
&
\text{val } x2 : (\text{int} \ \text{list}) \ \text{ref} &= \text{ref } [1,2,3] \\
&\text{val } _2 &= f2(x2) \\
&
\text{val } x3 : (\text{int} \ \text{ref}) \ \text{list} &= [\text{ref } 1, \text{ref } 2, \text{ref } 3] \\
&\text{val } _3 &= f3(x3)
\end{align*}
\]

\[
\begin{align*}
\text{length}(x1) = \quad \text{hd}(x1) = \\
\text{length}(x2) = \quad \text{hd}(!x2) = \\
\text{length}(x3) = \quad \text{!hd}(x3) =
\end{align*}
\]
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++, ...)

Using This Counter

```java
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

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

A Counter Generator

```java
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

```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))
```

What is \(\text{fact}(0)\)? How about \(\text{fact}(1)\)?

Another Implementation of 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))
```

Now what is \(\text{fact}(0)\)? How about \(\text{fact}(1)\)?

Pure vs. Imperative Interfaces

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

- **Ephemeral environments**
  ```ml
  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 behavior of the implementation.