CMSC 336: Type Systems for Programming Languages

Lecture 8: References

Acar & Ahmed **12 February 2008**

Contents

1 References

In many languages, you can perform an assignment operation that changes the contents of the memory or storage.

In some languages like ML, the mechanism for binding and assignment are different. Let's consider some examples.

```
x = 5          (* x is  a variable of value 5 *)
y = ref 5      (* y is a reference whose contents is 5 *)
w = x + 5      (* z has value 8 *)
x := 6         (* this is not OK.  x is a variable not a refernce *)
y := 6         (* this is OK.  changes the contents of y to 6 *)
z = y + 8      (* this is not OK.  y is a reference *)
z = (!y) + 8   (* this is OK.  z has value 14 *)
```

We will now see some example usage of references. We will use the following syntax.

- To create a reference, we will use the operation `ref v`. This operation will allocate a reference and change its contents to `v` and return the allocated reference.
- To deference we will write `!r`. The operation will return the contents of `r`.

- For assignment, we will write `r := v`. This operation will return unit.
- For the type of a reference whose contents if τ, we will write τ `ref`

2 Aliasing

Programming with references can be extremely difficult because it is difficult to know where a reference is pointing to. In particular, you can have aliasing, i.e., multiple references pointing to the same memory cell.

```
let val r = ref 10       (* allocate a reference *)
    val s = r            (* s becomes an alias for r *)
    val () = s := 18     (* assign s the value 18 *)
in
 print !r                (* print 18.  the value of r changes *)
end
```

Aliasing can make arguing about properties of program difficult. As an example, consider the program fragment

`(r := 1; r := !s)`

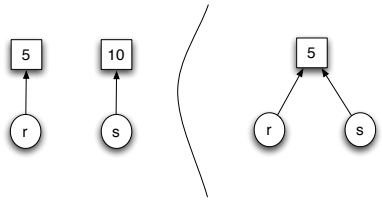
Is this equivalent to

`r:=!s`?

This depends on whether `r` or `s` are pointing to the same cell or not.

If `r` and `s` point to two difference cells, then this equivalence holds.

But if they are aliases (point to the same cell), then it does not hold anymore.



Aliasing enables communication. For example you can define two functions `ic` and `dc` that increment and decrement a shared counter.

```
val c = ref 0
fun ic () = (c := c+1; !c)
fun dc () = (c := c-1; !c)
```

Each invocation of these functions will change the contents of the counter `c`.

```
ic ()       (* c becomes 1 *)
ic ()       (* c becomes 2 *)
dc ()       (* c becomes 1 *)
```

In a sense, we have implemented an *object* that supports an increment and decrement function. More precisely, we can think of the pair `(ic,dc)` as such as object.

3 References to compound types

Since functions are first class values, reference can contain functions as their contents. This makes them quite powerful. We will see two example of this.

Recursion via backpatching. Using references, we can implement recursion. Here is an example.

```
(* a non-terminating function on naturals, nat -> nat *)
fun loop (x:nat) : nat = loop x

(* create a reference to this function *)
val r = ref loop

fun fact n = if n = 0 then 1
             else n * ((!r) (n-1))

val _ =  r := fact
```

Arrays. Here is an implementation of arrays using references and functions.

```
(* A nat array is a function from a given index to a nat *)
type natArray = ref (nat -> nat)

(* returns a function that returns 0 for each index *)
fun new (): natArray = ref (fn n:nat. 0)

(* the lookup function takes and array and an index
   and applies the array to the index
 *)
lookup a i = (!a) i

update a i v =
   let val oldArray = !a
       fun newArray n = if n = i then
                          v
                        else
                          oldArray n
```

4 Garbage Collection

We have not provided a primitive for performing deallocation. Instead we rely on garbage collection. This is not just for the purpose of performance. Without garbage collection, proving type safety of programs with reference is impossible.

Consider this program fragment.

```
val r  = ref 4
val s = ref 5
val x = (r,s)
     .
     .
     .
     .
dealloc r
val t = ref true
```

Now what is the type of `x`? Initially `x` had type `int` $\times$ `int`. But the allocation made it to a cell that is returned to the memory manager. Now suppose that at the next allocation the memory manager returns that cell. In that case `x` would have type `int` $\times$ `bool`! This would make it impossible to prove the preservation theorem.

5 A Language with References

$$\begin{array}{rcl} \tau & ::= & \texttt{unit} \mid \tau \rightarrow \tau \mid \texttt{ref}\ \tau \\ v & ::= & \star \mid \texttt{fix}\ f(x:\tau_1)\ :\ \tau_2\ \texttt{is}\ t\ \texttt{end} \\ t & ::= & v \mid x \mid t\ t \mid \texttt{ref}\ t \mid !t \mid t := t \end{array}$$

Let's develop the dynamic semantics for this language.

First, we need to close the language to make sure that during evaluation terms evaluate to proper terms.

What should `ref t` evaluate to?

In previous extensions such as with sum types, we simply allowed `inl v` and `inr v` to be values. Can we, similarly, have `ref v` be a value?

No, not quite. The problem is with assignment. When we assign to a reference, we want all other expression that hold that reference to know of the new contents. In other words, assignment requires that we have a "thing" to change; in this case the thing will be a *location*.

We extend values to contain locations.

$$v ::= \star \mid \texttt{fix}\ f(x:\tau_1)\ :\ \tau_2\ \texttt{is}\ t\ \texttt{end} \mid \ell$$

Here the meta variable ℓ ranges over an unspecified infinite set of locations $\mathcal{L}$.

We will store the contents of location inside a *store.*

5.1 Stores

A store is a partial function from the set of locations to values.

$$\sigma : \mathcal{L} \rightarrow v$$

For example, the following store maps the location ℓ_i to i.

$$\sigma = \{(l_1, 1), (l_2, 2), (l_3, 3), (l_4, 4)\}$$

Note that we can also give a typing to a store, as a partial function from locations to types, e.g.,

$$\Lambda = \{(l_1, \texttt{nat}), (l_2, \texttt{nat}), (l_3, \texttt{nat}), (l_4, \texttt{nat})\}$$

6 Store Semantics

Evaluation will take place in the context of a store. The store will model the world where the effects take place.

$$\frac{\sigma, t_1 \ \rightarrow \ \sigma', t_1'}{\sigma, t_1 \ t_2 \ \rightarrow \ \sigma', t_1' \ t_2}$$

$$\frac{\sigma, t_2 \ \rightarrow \ \sigma', t_2'}{\sigma, \texttt{fix}\ f\texttt{(}x : \tau_1\texttt{)}\ :\ \tau_2\ \texttt{is}\ t\ \texttt{end}\ t_2 \ \rightarrow \ \sigma', \texttt{fix}\ f\texttt{(}x : \tau_1\texttt{)}\ :\ \tau_2\ \texttt{is}\ t\ \texttt{end}\ t_2'}$$

$$\frac{}{\sigma, (\texttt{fix}\ f\texttt{(}x : \tau_1\texttt{)}\ :\ \tau_2\ \texttt{is}\ t\ \texttt{end}\ v) \ \rightarrow \ \sigma, [v/x, \texttt{fix}\ f\texttt{(}x : \tau_1\texttt{)}\ :\ \tau_2\ \texttt{is}\ t\ \texttt{end}/f]\ t}$$

$$\frac{\sigma, t \ \rightarrow \ \sigma', t'}{\sigma, \texttt{ref}\ t \ \rightarrow \ \sigma', \texttt{ref}\ t'}$$

$$\frac{l \notin \mathrm{dom}(\sigma)}{\sigma, \texttt{ref}\ v \ \rightarrow \ \sigma[l \leftarrow v], l}$$

$$\frac{\sigma, t \ \rightarrow \ \sigma', t'}{\sigma, !t \ \rightarrow \ \sigma', !t'}$$

$$\frac{}{\sigma, !l \ \rightarrow \ \sigma, \sigma[l]}$$

$$\frac{\sigma, t_1 \ \rightarrow \ \sigma', t_1'}{\sigma, t_1 := t_2 \ \rightarrow \ \sigma', t_1' := t_2}$$

$$\frac{\sigma, t_2 \ \rightarrow \ \sigma', t_2'}{\sigma, l := t_2 \ \rightarrow \ \sigma', l := t_2'}$$

$$\frac{}{\sigma, l := v \ \rightarrow \ \star, \sigma[l \leftarrow v]}$$

7 Type System

$$\frac{}{\Lambda;\Gamma \vdash \star : \texttt{unit}}$$

$$\frac{\Lambda(l) = \tau}{\Lambda;\Gamma \vdash \ell : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Lambda;\Gamma \vdash x : \tau}$$

$$\frac{\Lambda;(\Gamma, x:\tau) \vdash t : \tau_2}{\Lambda;\Gamma \vdash \texttt{fix}\ f(x:\tau_1)\ :\ \tau_2\ \texttt{is}\ t\ \texttt{end} : \tau_1\ \rightarrow\ \tau_2}$$

$$\frac{\Lambda;\Gamma \vdash t_1 : \tau_1\ \rightarrow\ \tau_2 \quad \Lambda;\Gamma \vdash t_2 : \tau_1}{\Lambda;\Gamma \vdash t_1\ t_2 : \tau_2}$$

$$\frac{\Lambda;\Gamma \vdash t : \tau}{\Lambda;\Gamma \vdash \texttt{ref}\ t : \tau\ \texttt{ref}}$$

$$\frac{\Lambda;\Gamma \vdash t_1 : \tau\ \texttt{ref} \quad \Lambda;\Gamma \vdash t_2 : \tau}{\Lambda;\Gamma \vdash t_1 := t_2 : \texttt{unit}}$$

$$\frac{\Lambda;\Gamma \vdash t : \tau\ \texttt{ref}}{\Lambda;\Gamma \vdash !t : \tau}$$