COMP 617 RAP Seminar, Fall 2006

Presentor: Walid Taha
Scribe: Gregory Malecha
10/19/2006

This lecture covered the syntax and semantics of adding reference types to the λ calculus.

1 What are References?

Up until this point, the λ calculus has been a purely functional language, values could be constructed and accessed, but never changed. References are one way to add mutability to the λ calculus.

The idea behind references is that a value points to a location and the location holds a value. When we want the value stored in a reference, we simply look up the value associated with the particular location.

A more common paradigm in mainstream languages is to make all variables references and have separate notions of what constitutes a value depending on which side of an assignment an expression occurs on. Although this is a viable approach, the approach of creating explicit reference cells is easier to integrate with the λ calculus.

2 Extending the Syntax

We extend the syntax with four constructions, a construction for making references, one for eliminating reference, one for performing assignment and finally with a value which is the result of constructing a reference. The augmented BNF looks as follows.

$$e ::= x \mid \lambda x.e \mid e\ e \mid \text{ref}\ e \mid !e \mid e\ :=\ e \mid l$$

The notation follow that of OCaml exactly. There are two things to note from this definition. First, the l represents a location and is used strictly as an intermediate form. A programmer would not be able to construct arbitrary location values. Second, we do not restrict the left hand side of the assignment operation to a variable. Such a restriction would not weaken the expressive nature of the language in anyway, but would simply require that values are bound to names before assignment. The following program shows an example in which the left hand side is not a variable.

$$(\lambda x.x)\ (\text{ref}\ x) := 1$$

If we restricted the left hand side to variables, then we would have to re-write this program as:

$$\text{let}\ x\ =\ (\lambda x.x)\ (\text{ref}\ x)\ \text{in}\ x := 1$$

NOTE: The `let` construct was introduced several lectures ago and is purely syntactic sugar for the application of a lambda abstraction.

3 Semantics of References

Because of the side-effects, we choose to use call-by-value semantics because it is much more understandable. Recall that the CBV semantics for the basic λ calculus are given by:

$$\frac{e \mapsto e'}{e\ e_1 \mapsto e'\ e_1}\ \text{T-App1} \qquad \frac{e_1 \mapsto e_1'}{(\lambda x.e)\ e_1 \mapsto (\lambda x.e)\ e_1'}\ \text{T-App2}$$

$$\frac{}{\lambda x.e\ v \mapsto e[x := v]}\ \text{T-App}$$

The existance of side-effects also dictates that introduction of a level of indirection when constructing evaluating references. This is because the evaluation of sub-terms can affect the evaluation of the larger term in which it resides in non-local ways. The level of indirection is referred to as a *store* and is represented by the Σ function which has type $\Sigma : \mathbb{L} \mapsto \mathbb{V}$ where $\mathbb{L}$ represents the set of locations and $\mathbb{V}$ represents the set of values.

The basic λ calculus rules augmented with stores are given below.

$$\frac{\Sigma, e \mapsto \Sigma', e'}{\Sigma, e\ e_1 \mapsto \Sigma', e'\ e_1}\ \text{T-App1} \qquad \frac{\Sigma, e_1 \mapsto \Sigma', e_1'}{\Sigma, (\lambda x.e)\ e_1 \mapsto \Sigma', (\lambda x.e)\ e_1'}\ \text{T-App2}$$

$$\frac{}{\Sigma, \lambda x.e\ v \mapsto \Sigma, e[x := v]}\ \text{T-App}$$

Note that when sub-terms are evaluated, they can generate possibly different environments which need to be passed on to subsequent evaluation. This is referred to as *threading* the store.

Finally, we can introduce the rules for evaluating references.

$$\frac{\Sigma, e \mapsto \Sigma', e'}{\Sigma, \text{ref}\ e \mapsto \Sigma', \text{ref}\ e'}\ \text{T-Ref1} \qquad \frac{}{\Sigma, \text{ref}\ v \mapsto \Sigma \oplus l \to v, l}\ \text{T-Ref}$$

$$\frac{\Sigma, e \mapsto \Sigma', e'}{\Sigma, !e \mapsto \Sigma', !e'}\ \text{T-Bang1} \qquad \frac{}{\Sigma, !l \mapsto \Sigma, \Sigma(l)}\ \text{T-Bang}$$

$$\frac{\Sigma, e \mapsto \Sigma', e'}{\Sigma, e := e_1 \mapsto \Sigma', e' := e_1}\ \text{T-Assign1} \qquad \frac{\Sigma, e_1 \mapsto \Sigma', e_1'}{\Sigma, l := e_1 \mapsto \Sigma', l := e_1'}\ \text{T-Assign2}$$

$$\frac{}{\Sigma, l := v \mapsto \Sigma \oplus l \to v, v}\ \text{T-Assign}$$

The value returned from an assignment has some variability. OCaml chooses to return the unit value, whereas values such as C and Java choose to return the value being assigned. A third alternative would be to return the location being assigned to. Our choice allows us to string assignments together such as:

$$x := (y := (z := 1))$$