Mutable References and Aliasing

CPSC 509: Programming Language Principles

Ronald Garcia*

12 February 2014

Adding References to TFL

In the semantics for IMP, we used stores σ to model the values of *locations* that could be reassigned as the program runs. In contrast, the language TFL has variables that are only bound to one value and never reassigned. In these notes, we add a notion of assignment to our TFL tiny functional language, without changing TFL's variables.

To do this, we add a language feature typically called *mutable references* to TFL. Mutable references are a new kind of value, just like numbers are a kind of value. A reference is used to *refer* to some other value, much like pointers in a language like C or C++. You can think of a reference as the address in memory (in the store, to be precise) where some value is stored. In this language, variables can be bound to references, which in turn refer to cells that contain other values.

First, let's extend TFL with three new operations.

$$x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad t \in \textsc{Term}, \quad v \in \textsc{Val}, \quad l \in \textsc{Loc}$$
$$t \quad ::= \quad x \mid n \mid t\ t \mid \lambda x.t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid t + t \mid \mathsf{ref}\ t \mid\ !t \mid t := t$$

The expression ref t evaluates the expression t to form a value, stores that value in a reference, and returns that reference as its result. The expression $!t$ evaluates t, expecting to get a reference as its result, and then returns the current value associated with the reference (this is called *dereferencing* a reference). Finally, the expression $t_1 := t_2$ evaluates t_1 to get a reference, evaluates t_2 to get a value v_1, and then stores the value in the reference and returns the value as its result. While this operation produces a value, its *side effect*, reassigning a reference, is what we really care about.

For a simple example, consider the program:

```
let x = ref 5
in let y =!x
   in let z = x := 6
      in y + !x
```

It binds x to a reference cell containing 5, binds y to the result of dereferencing x (which is 5), assigns 6 to the reference bound to x, binding 6 as well fo the variable z, and then sums y with the value referenced by x.

Modeling References

To model these language features, we introduce a notion of *stores* σ, which map a *finite* number of *locations* l to *storable objects*.

$$l \in \textsc{Loc}$$
$$\sigma \in \textsc{Sto} = \textsc{Loc} \xrightarrow{\text{fin}} \textsc{SO}$$

*© 2014 Ronald Garcia.

We use the notation $\xrightarrow{\text{fin}}$ to refer to *finite partial functions*, which are partial functions with the added constraint that the partial function only maps a finite number of elements of the domain (which means that treated as a set of pairs, a finite partial function is itself finite).

In general, a language may differentiate between what is storable and what counts as a term, value, etc. In this particular language, however, the storable objects SO will be *exactly* the values.

$$\begin{array}{rcl} v \in \textsc{Val} = \textsc{So} & & \\ v & ::= & n \mid \lambda x.t \mid l \end{array}$$

We include locations l among the values of this language, which means that locations must also be terms:

$$t ::= \ldots \mid l$$

However, locations cannot appear in source programs: they only appear at runtime. These locations play the role of what we have been informally calling "reference cells".

Before we define the big-step semantics of TFL, we will add one more feature to the language: a sequencing operator:

$$t ::= \ldots \mid t;t$$

The sequencing operator, as we'll see, is simply a convenience when we don't care about the result of an assignment operator.

Now here is a big-step semantics for TFL with references.

$\Downarrow \sqsubseteq (\textsc{Term} \times \textsc{Sto}) \times (\textsc{Val} \times \textsc{Sto})$

$$\text{(num)}\frac{}{\langle n, \sigma\rangle \Downarrow \langle n, \sigma\rangle} \qquad \text{(plus)}\frac{\langle t_1, \sigma_1\rangle \Downarrow \langle n_1, \sigma_2\rangle \quad \langle t_2, \sigma_2\rangle \Downarrow \langle n_2, \sigma_3\rangle}{\langle t_1 + t_2, \sigma_1\rangle \Downarrow \langle n, \sigma_3\rangle} \text{ where } n = n_1 + n_2$$

$$\text{(lambda)}\frac{}{\langle \lambda x.t, \sigma\rangle \Downarrow \langle \lambda x.t, \sigma\rangle} \qquad \text{(loc)}\frac{}{\langle l, \sigma\rangle \Downarrow \langle l, \sigma\rangle}$$

$$\text{(app)}\frac{\langle t_1, \sigma_1\rangle \Downarrow \langle \lambda x.t_{11}, \sigma_2\rangle \quad \langle t_2, \sigma_2\rangle \Downarrow \langle v_2, \sigma_3\rangle \quad \langle [v_2/x]t_{11}, \sigma_3\rangle \Downarrow \langle v, \sigma_4\rangle}{\langle t_1\ t_2, \sigma_1\rangle \Downarrow \langle v, \sigma_4\rangle}$$

$$\text{(let)}\frac{\langle (\lambda x.t_2)\ t_1, \sigma_1\rangle \Downarrow \langle v, \sigma_2\rangle}{\langle \textsf{let } x = t_1 \textsf{ in } t_2, \sigma_1\rangle \Downarrow \langle v, \sigma_2\rangle} \qquad \text{(seq)}\frac{\langle \textsf{let } x = t_1 \textsf{ in } t_2, \sigma_1\rangle \Downarrow \langle v, \sigma_2\rangle}{\langle t_1;t_2, \sigma_1\rangle \Downarrow \langle v, \sigma_2\rangle} \text{ where } x \notin FV(t_2)$$

$$\text{(ref)}\frac{\langle t, \sigma_1\rangle \Downarrow \langle v, \sigma_2\rangle}{\langle \textsf{ref } t, \sigma_1\rangle \Downarrow \langle l, \sigma_2[l \mapsto v]\rangle} \text{ where } l \notin dom(\sigma_2) \qquad \text{(deref)}\frac{\langle t, \sigma_1\rangle \Downarrow \langle l, \sigma_2\rangle}{\langle !t, \sigma_1\rangle \Downarrow \langle \sigma_2(l), \sigma_2\rangle}$$

$$\text{(asgn)}\frac{\langle t_1, \sigma_1\rangle \Downarrow \langle l, \sigma_2\rangle \quad \langle t_2, \sigma_2\rangle \Downarrow \langle v_2, \sigma_3\rangle}{\langle t_1 := t_2, \sigma_1\rangle \Downarrow \langle v_2, \sigma_3[l \mapsto v_2]\rangle}$$

The semantics extends call-by-value TFL with a store and references.

The big-step relation for each expression now threads a store σ through it. One side effect of this is that expressions like $t_1 + t_2$ now impose a specific order on the evaluation of its arguments (in this case, t_1 is reduced before t_2).

Just as let is defined in terms of λ, now $t_1;t_2$ is defined in terms of let. Since the x can not capture any of t_2's free variables, the value of t_1 is essentially discarded.

The three rules for handling references capture the informal description of their behavior that we gave earlier.

We define our evaluator for these semantics with a little indirection:

$$\begin{array}{c} \textsc{Pgm} = \{\, t \in \textsc{Term} \mid FV(t) = \emptyset \,\}, \quad eval : \textsc{Pgm} \rightharpoonup \textsc{Obs} \\ eval(t) = unload(v, \sigma) \text{ iff } \langle t, \emptyset\rangle \Downarrow \langle v, \sigma\rangle\,. \end{array}$$

It's now up to us to decide what can be observed in the evaluator. One simple definition says that if the big-step semantics produces a reference, then just say so:

$$\textsc{Obs} = \{\,\mathsf{procedure}, \mathsf{reference}\,\} \cup \mathbb{Z}$$
$$unload(n, \sigma) = n$$
$$unload(\lambda x.t, \sigma) = \mathsf{procedure}$$
$$unload(l, \sigma) = \mathsf{reference}$$

1 Mutable References vs. Mutable Variables

Adding mutable references to TFL gives it a lot of the power of IMP in that it supports imperative programming with assignment. However, in some ways TFL is more "expressive" than IMP.

1. IMP has a single storage location associated with each variable. TFL separates variables from assignable storage. Multiple variables can point at the same reference cell, so that modifying the value through one variable makes the change visible through the other variable. This can be seen in the following example program:

   ```
   let x = ref 5
   in let y = x
      in y := 7;
         !x
   ```

   Here, a single reference cell is created containing 5. The reference is bound to both y and x, but it is modified using y and dereferenced using x. The result is 7. Having multiple variables point to the same storage is called *aliasing*: each variable is an alias for the same underlying value. This is a very useful feature for programming, but it makes program analysis much more difficult.

2. Recall that IMP could only assign numbers to variables. TFL with references allows references and even functions to be stored and retrieved. This makes it possible to do things like implement recursion without using the Y combinator:

   ```
   let f = ref 7
   in let x = λn.(!f) n
      in f := x;
         (!f) 3
   ```

   In this example, f is first bound to a reference cell that contains a dummy value. Then a function is bound to x that dereferences f and applies the result to n. This would be a problem if f contained the number 7, but we first replace it with the function x and then begin the chain of calls that causes an infinite loop.

   Note that we could always restrict our references so that they could only hold numbers, just like IMP. Then the language would be less powerful, but sometimes it's helpful to have a more restricted language if the features of that language are all that you need. Then it can be it's easier to implement the language, and it might even be easier to write correct programs.