Mutable References and Aliasing

CPSC 509: Programming Language Principles

Ronald Garcia∗

12 February 2014

Adding References to TFL

So far, we have been working with languages where variables are bound once (either as the result of a procedure call or by let-binding).

In these notes, we add a notion of assignment to our TFL tiny functional language, without changing TFL’s variables. Recall that TFL is very similar to our earlier language with procedures but we’ve switched to using integers rather than numerals in the language, now that numerals have worn out their pedagogical value.

To do this, we add something called *mutable references* to TFL. Mutable references are a new kind of value, just like numbers are a 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 other values.

To create this language, we extend TFL with three new operations.

\[
x \in \text{VAR}, \quad n \in \mathbb{Z}, \quad t \in \Lambda, \quad v \in \text{VAL}, \quad l \in \text{LOC}
\]

\[
t ::= x \mid n \mid tt \mid \lambda x.t \mid \text{let } x = t \text{ in } t \mid t + t \mid \text{ref } t \mid !t \mid t := t
\]

The expression \text{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 \text{!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 \text{t1 := t2} evaluates \text{t1} to get a reference, evaluates \text{t2} to get a value \text{v1}, 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:

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

It binds \text{x} to a reference cell containing \text{5}, binds \text{y} to the result of dereferencing \text{x} (which is \text{5}), assigns \text{6} to the reference bound to \text{x}, and then sums \text{y} with the value referenced by \text{x}.

∗© Ronald Garcia. Not to be copied, used, or revised without explicit written permission from the copyright owner.
Modeling References

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

$$ l \in \text{LOC} $$

$$ \sigma \in \text{STO} = \text{LOC}^* \Rightarrow \text{STO} $$

We use the notation $\Rightarrow$ 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 STO will be exactly the values.

$$ v \in \text{VAL} = \text{STO} $$

$$ v ::= n \mid \lambda x.t \mid l $$

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.

$$ \Downarrow: (A \times \text{STO}) \times (\text{VAL} \times \text{STO}) $$

\[
\begin{align*}
(n, \sigma) \Downarrow (n, \sigma) & \quad \text{(num)} \\
(t_1, \sigma_1) \Downarrow (n_1, \sigma_2) \quad (t_2, \sigma_2) \Downarrow (n_2, \sigma_3) & \quad \text{where } n = n_1 + n_2 \quad \text{(plus)} \\
(\lambda x.t, \sigma) \Downarrow (\lambda x.t, \sigma) & \quad \text{(lambda)} \\
(t_1 + t_2, \sigma_1) \Downarrow (n, \sigma_3) & \quad \text{(loc)} \\
(t_1, \sigma_1) \Downarrow (\lambda x.t_1, \sigma_2) & \quad \text{(app)} \\
(t_2, \sigma_2) \Downarrow (v_2, \sigma_3) & \quad \text{where } x \notin \text{FV}(t_2) \quad \text{(let)} \\
(t_1 t_2, \sigma_1) \Downarrow (v, \sigma_4) & \quad \text{(seq)} \\
\langle l, \sigma \rangle \Downarrow (\text{STO}(l), \sigma_2) & \quad \text{(ref)} \\
\langle \text{ref } t, \sigma_1 \rangle \Downarrow (l, \sigma_2[l \mapsto v]) & \quad \text{(asgn)}
\end{align*}
\]

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

The big-step relation for each expression now threads a store $\sigma$ 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 $\text{let}$ is defined in terms of $\lambda$, now $t_1; t_2$ is defined in terms of $\text{let}$. Since the $x$ cannot 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:

\[ \text{eval} : \Lambda^0 \rightarrow \text{RESULT} \]
\[ \text{eval}(t) = \text{unload}(v, \sigma) \text{ iff } \left\langle t, \emptyset \right\rangle \downarrow \left\langle v, \sigma \right\rangle. \]

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:

\[ \text{RESULT} = \{ \text{procedure, reference} \} \cup \mathbb{Z} \]
\[ \text{unload}(n, \sigma) = n \]
\[ \text{unload}(\lambda x.t, \sigma) = \text{procedure} \]
\[ \text{unload}(l, \sigma) = \text{reference} \]

Alternatively, we can be more sophisticated, and try to look up the underlying non-reference value.

\[ \text{RESULT} = \{ \text{procedure} \} \cup \mathbb{Z} \]
\[ \text{unload}(n, \sigma) = n \]
\[ \text{unload}(\lambda x.t, \sigma) = \text{procedure} \]
\[ \text{unload}(l, \sigma) = \text{unload}(\sigma(l), \sigma) \]

Observe that \(\text{unload}\), and in turn \(\text{eval}\), is undefined in the case of a cyclic reference, where one location points to another that points back. This is an easy situation to produce.

\[
\begin{align*}
\text{let } x &= \text{ref } 5 \\
\text{in let } y &= \text{ref } x \\
\text{in } x := y; \\
\text{x}
\end{align*}
\]

Try evaluating this program and see what the resulting store looks like. One could alternatively define \(\text{unload}\) to detect cycles in the reference graph and produce some meaningful result in those cases (Dr. Racket has exactly this kind of behavior).