The Store
(Adding Mutation to the Interpreter, Part II)
Lecture Notes for cs173, Fall 2001

sk and rob

October 17, 2001
updated October 21, 2001

1 Motivating the Store

Consider the Rip program

\[
(+ (\text{let} \begin{array}{l}
\{ \times 3 \} \\
\times
\end{array})
\times)
\]

This should program should result in an error since the last \(\times\) is unbound. In our current interpreter, however, we get 3. The problem is that the environment (in which \(\times\) is bound to 3) is passed from the left-hand-side of the \(\text{addE}\) to the right-hand-side, so the \(\times\) is evaluated in the wrong environment.

Unfortunately, we had a good reason for passing the environment from left to right. Mutations that occur in one part of an expression may need to be visible in another part of the expression. But clearly from this example, environments cannot be passed from one part of an expression to another, since we will violate our scoping rules.

In other words, environments are tree structures. A variable created with a \text{let construct}, for example, should only be visible to its sub-expressions. A mutation of a bound instance of a variable, however, must be visible to the entire scope of the variable.

In the figure below, circle A is an expression and circles B and C are both sub-expressions of A. Because environments are trees, A’s environment is available to B and C. But if B extends this environment, the extended environment will not be available to C. B, however, can mutate A’s environment, and this mutation must be visible to C. Thus, while the environment needs to preserve its tree structure, an accumulation of mutations needs to be threaded, as shown by the dotted line in the figure.
The problem with our earlier approaches is that we were threading mutations by threading the environment. Since we know that the environment cannot be threaded, let’s separate these entities. We will treat environments exactly as in our original interpreter (so as to preserve the scoping properties of the language constructs), and we will thread an accumulation of mutations through interpretation. We call an accumulation of mutations a store.

To make the distinction between the environment and the store useful, we must use both of these entities to get from names to values. The simplest way to accomplish this is to use some intermediate object to which environments map names, and then use that object to look up values in the store. Since we will eventually map the store to the computer’s memory, it’s useful to call these intermediate objects locations. The contracts for environments and stores are as follows:

\[
\begin{align*}
\text{dsub} & : \text{var-name} \rightarrow \text{location} \\
\text{store} & : \text{location} \rightarrow \text{value}
\end{align*}
\]

\section{2 Mutation Done Right}

We update our contract for the interpreter. Since we are threading the store, we need to accept a store as an argument and return the (potentially updated) store.

\[
\text{interp} : \text{RP} \times \text{dsub} \times \text{store} \rightarrow \text{value} \times \text{store}
\]

Value expressions in our interpreter are easy to handle; we simply bundle the same store that we received in the return value:

\[
\begin{align*}
(\text{define} & (\text{interp} \ expr \ dsub \ store) \\
& \begin{cases}
\text{cond} \\
\quad [(\text{numE?} \ expr) \ (\text{values} \ expr \ \text{store})] \\
\quad [(\text{procE?} \ expr) \ (\text{values} \ (\text{make-closure} \ expr \ \text{dsub}) \ \text{store})]
\end{cases}
\end{align*}
\]

For a varE, we need to lookup the value of the variable. Because we are using an environment and a store, we lookup the location of the variable name in the environment, and then lookup the value located at that location:
Let’s now consider an *addE*. The store can be mutated in the left-hand-side of the addition, and any mutations need to be passed along to the right-hand-side:

\[
(\text{addE? expr}) (\text{let-values} (\text{[l-val l-store})
\quad (\text{interp (addE-left expr dsub store)})
\quad \text{let-values} (\text{(r-val r-store})
\quad (\text{interp (addE-right expr dsub l-store)})
\quad (\text{values (numE+ l-val r-val)})
\quad \text{r-store})))
\]

What happens when we reach a *setE*? We need to interpret the val expression. Since this expression can mutate the environment, we use the resulting store in the invocation of update-store:

\[
(\text{setE? expr}) (\text{let-values} (\text{[v-val v-store})
\quad (\text{interp (setE-val expr dsub store)})
\quad (\text{values v-val})
\quad (\text{update-store v-store})
\quad (\text{lookup-dsub (setE-var-name expr dsub v-val)}))]
\]

We are left with the problem of interpreting procedure applications. We interpret the procedure part of the application and thread the resulting store to the interpretation of the argument. We then call do-app using the resulting procedure value, argument value, and store. Note that we do not pass the dsub into do-app: since we are implementing a statically-scoped interpreter, we should use the dsub contained in the closure p-val:

\[
(\text{appE? expr}) (\text{let-values} (\text{[p-val p-store})
\quad (\text{interp (appE-proc expr dsub store)})
\quad \text{let-values} (\text{[a-val a-store})
\quad (\text{interp (appE-arg expr dsub p-store)})
\quad \text{do-app p-val a-val a-store})))
\]

The function do-app now must not only extend the environment with the argument of the procedure, but also allocate a new location in the store for this environment name to point to:

\[
\text{do-app : closure} \times \text{value} \times \text{store} \rightarrow \text{value} \times \text{store}
\]

\[
\text{(define (do-app clos argV store)}
\quad \text{(let-values (}[\text{ext-dsub ext-store})
\quad \text{ext-dsub-and-store (closure-dsub clos)}
\quad \text{store (interp (procE-body (closure-proc clos)) argV))}
\quad (\text{interp (procE-arg-name (closure-proc clos))})
\quad \text{store (procE-body (closure-proc clos))})
\]
We must now define the helper procedure `extend-dsub-and-store`, which ensures that we use a consistent new location:

```
extend-dsub-and-store : dsub × store × var-name × value → dsub × store
```

```
(define (extend-dsub-and-store dsub store var-name val)
  (let ([new-locn (next-store-locn)])
    (values
      (extend-dsub dsub var-name new-locn)
      (extend-store store new-locn val))))
```

How do we implement `next-store-locn`? The contract of the function is

```
next-store-locn : → number
```

The most straightforward way is to mutate a counter. This ensures that we always allocate new locations:

```
(define next-locn
  (let ([last −1])
    (lambda ()
      (begin
        (set! last (add1 last))
        last)))
```

It may be slightly unsatisfying to have used `set!` to implement mutation. As it stands, every part of our interpreter, except `next-locn`, could be implemented in a language without mutation (such as Haskell). If this matters, we can solve this problem the same way we solved the problem of implementing mutation in the first place. That is, we can thread the next available store location through the computation in parallel with the store itself (or turn the store into a structure of its contents and its next available location). We’ve chosen, however, not to do this in these notes because it just makes the code more unwieldy.