Adding Mutation to the Interpreter, Part I
Lecture Notes for cs173, Fall 2001

sk and rob

October 15, 2001

1 Motivation

When converting a program into a web program, environments in the original program are represented in the URL (assuming we use the GET method) of the web version of the program. We can pass around and update the environment by using hidden fields.

Consider a web page with “environment” \([x \mapsto 3]\). When we clone this page, the new page will have the same environment. But what if we mutate the value of \(x\) in one of the pages? We would like this mutation to be visible in both pages, since they have the same environment. This won’t happen, however, if we use hidden fields to implement mutation. For assignment 3, most teams solved this problem by using cookies (or something equivalent) to allow mutations to be visible across cloned pages.

To get a better understanding of mutation, let’s explore the subject by adding a mutation construct to our Rip interpreter.

2 Mutation in Rip

We augment Rip with a syntax for mutation:

\[
\{ := \text{var} \ \text{val} \}
\]

with the corresponding structure definition:

*(define-struct setE (var val))*

where \text{var} is a symbol and \text{val} is an RP which must reduce to a value. How should we update the interpreter? Let’s start by producing the template for the \text{setE} clause:

\[
[(\text{setE? expr}) \ldots (\text{setE-var expr}) \ldots (\text{interp (setE-val expr)} \ldots)]
\]

A \text{setE} should produce a new environment with the value of \text{setE-var} mutated to \text{setE-val}. Perhaps the natural solution is to use the function
**update-dsub**: dsub × var-name × val → dsub

It is an easy exercise to implement `update-dsub`. We can now fill in our template:

\[
((\text{setE?\ }\text{expr})\ (\text{update-dsub\ }\text{dsub})
\quad \begin{cases}
    (\text{setE-var\ }\text{expr}) \\
    (\text{interp\ (setE-val\ }\text{expr}\ \text{dsub}))
\end{cases}
\]

One problem is that we are now returning an environment when we interpret a `setE`. This violates our contract for `interp`, which dictates that we return a value. From this point on, let’s assume that interpretation of a `setE` yields the new value of the variable. To fix our current solution, we need to update our environment and return the new value. Our `setE` clause, therefore, might become

\[
((\text{setE?\ }\text{expr})\ (\text{let\ }\begin{cases}
    (\text{new-val\ (interp\ (setE-val\ }\text{expr}\ \text{dsub})))
\end{cases}
\quad \begin{cases}
    (\text{begin}) \\
    (\text{update-dsub\ }\text{dsub})
\end{cases}
\quad \begin{cases}
    (\text{setE-var\ }\text{expr}) \\
    \text{new-val})
\end{cases}
\]

This solution fails miserably. Consider the Rip code

\[
(\text{let\ }\begin{cases}
    (\times\ 1)
\end{cases}
\quad \begin{cases}
    (\text{let\ }\begin{cases}
        (\text{dummy\ :=\ }\times\ 2)
    \end{cases}
\quad \times))\]

This program (incorrectly) returns 1. The variable `dummy` is bound to 2, but the fact that the environment was mutated is completely lost by the time the final `\times` is evaluated.

It’s clear we need to modify the environment and return a value, but we need to preserve the modified environment also. So why not just return *both* the modified environment and the value? We first update our contract:

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

The `env` on the left-hand-side of the contract is just as before; it’s the environment in which we interpret the given expression. The `env` on the right-hand-side is the (potentially mutated) environment that results from interpreting the expression.

For convenience, we use the Scheme constructs `values` and `let-values` which allow functions to return and accept multiple values. Their use should be clear in the code that follows. We update our interpreter clause-by-clause:

\[
\text{define} (\text{interp\ expr\ dsub})
\quad \begin{cases}
    (\text{cond}) \\
    \quad \begin{cases}
        ((\text{numE?\ }\text{expr})\ (\text{values\ expr\ dsub}))
    \end{cases}
\end{cases}
\]

2
Because a number is a constant, evaluating it can’t possibly result in any mutations. Therefore, the interpreter returns the same environment that it accepts. Variables and procedures are also values, so we can treat them similarly:

\[
((\text{varE? } \text{expr}) \ (\text{values (lookup-dsub expr dsub) dsub})) \\
((\text{procE? } \text{expr}) \ (\text{values (make-closure expr dsub) dsub}))
\]

The interpretation of an \textit{addE} is more complicated because of the possibility of a mutation during the computation. We pick a left-to-right order of evaluation, and hence interpret the left-hand-side first, and then use the resulting environment in the interpretation of the right-hand-side:

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

We now come to the interpretation of a \textit{setE}. We will first interpret the \textit{val} expression, and then return the resulting value along with the mutated environment:

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

We are left with \textit{appE’s} to deal with. Not surprisingly, we will interpret the procedure part of the application, and then use the resulting environment to interpret the argument part of the application:

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

Recall the function \textit{do-app}, which remains unchanged:

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

\[
\text{define (do-app clos argV)} \\
\quad (\text{interp (procE\text{-body clos) \\
\quad (extend\text{-dsub clos (procE\text{-arg-name clos)} \\
\quad \quad argV)))}
\]

3 Testing Our Solution

Let’s test that mutated environments are handled correctly in a Rip \textit{addE}:
Because the mutated environment from the left-hand-side of the addition is passed to
the interpretation of the right hand side, this program correctly produces the value 6.

Let’s consider a simple example of mutation occurring in the procedure half of an
application:

(let {f {proc (x) 0}}
  {{:= f {proc (x) x}}
   1}})

Our interpreter returns the correct value 1, but this test is unsatisfying because it never
uses the variable f after it is mutated. We create a similar test case which tests the
Persistence of the mutation:

(let {f {proc (x) 0}}
  {{:= f {proc (x) (+ x 1))}
   (f 3))})

If the mutation of f (incorrectly) does not affect {f 3}, then the entire program returns 1 (because the f that gets applied is the one that always returns 0). But this does not happen; the expression {f 3} evaluates to 4, and the entire program evaluates (correctly) to 5.

Now consider what happens when we mutate a variable that occurs inside the body of
a procedure to be applied, as in

(let {a 1}
  {{proc (x) (+ x a))
   (= a 2))})

This program should return 4, but it instead returns 3 because the mutation is shadowed by the use of the closure in which a is bound to 1. We try to fix this problem by passing the mutated environment into the closure. We need to change do-app:

**do-app : closure \times value \times dsub \rightarrow value**

(let {do-app clos argV mutated-dsub}
  (interp (procE-body (closure Proc clos))
    (extend-dsub mutated-dsub
      (procE-arg-name (closure Proc clos))
      argV)))

We need to pass the mutated environment to do-app, so we modify the appE clause of
the interpreter:
[(\(appE\ expr\)) (let-values\ ((\(p-val\ p-dsub\))
  (interp\ (appE-proc\ expr\)\ dsub))]
 (let-values\ ((\(a-val\ a-dsub\))
  (interp\ (appE-arg\ expr\)\ p-dsub))]
 (do-app\ p-val\ a-val\ a-dsub))]

We now get the correct answer 4 for the previous test case, but we’ve created another problem—our interpreter is now dynamically scoped! A program such as

\[
(let\ \{x\ 3\}
  (let\ \{f\ \{proc\ \{y\ \{+\ x\ y\}\}\}
    (let\ \{x\ 4\}
      \{f\ 5\})))
\]

returns 9 instead of 8. Notice that to demonstrate this error, we didn’t even need to use any mutation in the program. In a program without mutations, all the \(dsubs\) returned by interpretation will be the same as the ones given to the interpreter as an argument. If the \(dsubs\) never change, we can just as well remove them as a second return value entirely. If we do that, we see that we’ve reduced our interpreter to the classic dynamically scoped interpreter of September 21st. That makes it easy to see that we’ve introduced dynamic scoping.

In short, we’ve created a big mess. Next time we’ll see how to save ourselves.