Mutation

sk, dbtucker, and kfisler

September 29, 2003

1 Background

Mutation allows a programmer to change the values associated with names. In other words, it endows a program with *state*. Mutation (aka assignment) is a standard feature in most programming languages. However, the dialect of Scheme we have used so far has, for the most part, been devoid of state. Indeed, some languages (such as Haskell) have no direct mutation operations at all. It is, therefore, possible to design and use languages—even quite powerful ones—that have no direct notion of state. Simply because this idea—that one can program without state—hasn't caught on in the mainstream is no reason to reject it.[1]

That said, state does have its place in computation. If we create programs to model the real world, then some of those programs are going to have to accommodate the fact that there the real world has events that truly alter it. For instance, cars really do consume fuel as they run, so a program that models a fuel tank needs to record changes in fuel level.

Despite that, it makes sense to shirk state where possible because state makes it harder to reason about programs. Once a language has mutable entities, it becomes necessary to talk about the program *before* a mutation happened and *after* the mutation (i.e., the different "states" of the program). Consequently, it becomes much harder to determine what a program actually *does*, because any such answer becomes dependent on *when* one is asking: that is, they become dependent on time.

Because of this complexity, programmers should use care when introducing state into their programs. A legitimate use of state is when it models a real world entity that really is itself changing: that is, it models a *temporal* or *time-variant* entity. Contrast that with a use of state in the following loop:

```
{
  int i;
  sum = 0;
  for (i = 0; i < 100; i = i++)
    sum += f(i);
}
```

There are two instances of state here (the mutation of `i` and of `sum`); *neither* of these is essential. Any other part of the program that depends on the value of `sum` remaining unchanged is going to break. You might argue that at least the changes to `i` are innocuous, since the identifier is local to the block defined above; however, even that assumption fails in a multi-threaded program, as most large programs tend to be today! Indeed, the use of state is the source of most problems in multi-threaded software. In contrast, the following program

(*foldl* + 0 (*map f* (*build-list* 99 *add1*)))

(where (*build-list* 99 *add1*) generates (*list* 0 ... 99)) computes the same value, but is thread-safe by virtue of being functional (mutation-free). Better still, a compiler that can be sure that this program will not be run in a multi-threaded context can generate the mutation-based version from this specification.

[1] The family of languages we've studied so far have been inspired by the lambda calculi designed starting in the 1930s by Alonzo Church. He initially conceived of them as purely theoretical models for use when exploring the limitations of computable functions. It appears, however, that even back then, Church realized these functions may have broader use.

2 A Mutation Construct and its Behavior

Let's add a set (assignment) construct to our language. For concrete syntax, we'll use

```
{set <id> <RCFAE>}
```

as in

```
{set x {+ x 1}}
```

How should set expressions behave? Let's work this out through a few examples:

- `{set x 5}`

  This should report an error, since *x* is free.

- `{with {x 3} {set x 5}}`

  should return 5 (set expressions will return the value they just assigned to the variable, assuming that variable is bound).

- `{with {x 3} {set x {+ 4 3}}}`

  should return 7.

- `{with {x 4} {with {y {set x 6}} x}}`

  should return 6.

- ```
  {with {x 4}
    {with {y x}
      {with {z {set x 5}}
        y}}}
  ```

  should return 4. This is our first interesting semantic decision: should variables alias one another? In other words, should changing the value of x *also* change the value of y? We will adopt the interpretation that set should only change the value of the named variable. (The Scheme **set!** operator implements this behavior, while assignment in C would change the values of both x and y. This is because C implements *pointer* assignment instead of *variable* assignment. As a result, aliasing can be a serious problem in C programs.)

- ```
  {with {y 1}
    {with {f {fun {x} {+ x y}}}
      {with {z {set y 8}}
        {f 5}}}}
  ```

  This could return 6 or 13, depending upon whether the set affects the value of *x* stored in the closure. Changes to variable values should affect values stored in closures.

  But wait – isn't this *dynamic scoping*, which we've decided is a Very Bad Thing? No, this isn't dynamic scope. Scoping determines which *binding instance* of an identifer each bound instance refers to. It does not govern changes to the values of bound variables. Arguably, if you could capture variables in environments, but not trust mutation to affect their values, mutation would become a much less useful construct in conjunction with closures.

Now that we know how the set construct should behave, it's time to extend the interpreter.

3 Implementing Mutation

If you recall our discussion from last week about syntactic- versus meta- interpreters, it is clear that we have two options: we can implement set using Scheme's assignment operators (a meta-interpreter), or we can implement it without using assignment operators (a syntactic interpreter). In this case, even the meta-interpreter is instructive, so we'll do that one for now.

The first (easy) step is to extend the data defintion. For purposes of simplicity, we'll add mutation to the FAE language (ie, the language without recursion and assuming with is handled in the parser), calling the new language FAE! (the ! indicates mutation, a convention adopted from Scheme).

```
(define-datatype FAE! FAE!?
  [num (n number?)]
  [add (lhs FAE!?) (rhs FAE!?)]
  [sub (lhs FAE!?) (rhs FAE!?)]
  [id (name symbol?)]
  [set (var symbol?) (new-val FAE!?)]
  [fun (param symbol?) (body FAE!?)]
  [app (fun-expr FAE!?) (arg-expr FAE!?)])
```

Extending the parser is easy, so we won't review that here. Here's the interpreter, ready for us to add the set implementation:

```
;; interp : FAE! SubCache → FWA-value
;; evaluates FAE! expressions by reducing them to their corresponding values
(define (interp expr sc)
  (cases FAE! expr
    [num (n) (numV n)]
    [add (l r) (numV+ (interp l sc) (interp r sc))]
    [sub (l r) (numV- (interp l sc) (interp r sc))]
    [id (v) (lookup v sc)]
    [set (var new-val-expr) ...]
    [fun (param body)
         (closureV param body sc)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr sc)]
                 [define arg-val (interp arg-expr sc)])
           (cases FWA-value fun-val
             [closureV (cl-param cl-body cl-cache) (interp cl-body
                                                           (aSub cl-param
                                                                 arg-val
                                                                 cl-cache))]
             [else (error 'interp "can only apply functions")]))]))
```

Where do we currently associate values with identifiers? In the environment, here referred to as *sc* (for "substitution cache"). This suggests that the right way to implement mutation is to change/update the environment to reflect the new value (remember that Scheme has an assignment operator called **set!**). Let's try the following code for set, along with the associated helper function:

```
;; interp : FAE! SubCache → FWA-value
;; evaluates FAE! expressions by reducing them to their corresponding values
(define (interp expr sc)
  (cases FAE! expr
    ...
    [set (var new-val-expr)
         (local ([define new-val (interp new-val-expr sc)])
           (begin (set! sc (change-val var new-val sc))
```

```
                    new-val))]
    ))

;; change-val : symbol FWA-value SubCache → SubCache
;; replaces value for variable in cache with new value
(define (change-val name val sc)
  (cases SubCache sc
    [mtSub () (error 'change-val "no binding for identifier")]
    [aSub (bound-name bound-value rest-sc)
          (if (symbol=? bound-name name)
              (aSub bound-name val rest-sc)
              (aSub bound-name bound-value (change-val name val rest-sc)))]))
```

If we run this on the example

```
{with {x 4} {with {y {set x 6}} x}}
```

(which we decided should return 6), we get 4. What happened?

The **set!** changed the value of *sc* that was current when we were processing the set expression. This does NOT change the value of *sc* in other calls to *interp*. Specifically, we will call *interp* once to evaluate the named-expr in the inner with (the set), and then again to evaluate the body (the reference to x). The **set!** changed the value of *sc within* the call to *interp* on the named-expr. It does not change the value of *sc* on the call to *interp* on the body, so the update to x isn't visible.

The real problem here is that we tried to change the wrong thing. We want to change the *contents* of the environment (*sc*), not the environment itself. If we change the contents, then anywhere we refer to the same environment structure sees the new contents. If we change what *sc* refers to, that change will only be visible locally.

Scheme's **set!** operator, like our set operator, expects a syntactic symbol as its first argument. This means that we cannot use **set!** to alter the contents stored at variables. Recall from the recursion lecture, however, that we also saw another Scheme assignment operator, *set-box!*. With *set-box!*, we changed the contents stored in the box, not the box itself. So *set-box!* seems better suited to our purposes.

3.1 Implementing Mutation with Scheme Boxes

How might we use boxes? We could change the environment so that each identifier maps to a box that contains a value, rather than to a value. We can define environments to use boxes by changing the predicate that recognizes values in the *SubCache* datatype:

```
(define-datatype SubCache SubCache?
  [mtSub]
  [aSub (name symbol?)
        (value (lambda (x) (and (box? x) (FWA-value? (unbox x)))))
        (sc SubCache?)])
```

This change forces changes in the interpreter code. Specifically, anywhere we use or insert values into the environment need to get modified. These happen in the *id* and *app* cases.

```
;; interp : FAE! SubCache → FWA-value
;; evaluates FAE! expressions by reducing them to their corresponding values
(define (interp expr sc)
  (cases FAE! expr
    [num (n) (numV n)]
    [add (l r) (numV+ (interp l sc) (interp r sc))]
    [sub (l r) (numV- (interp l sc) (interp r sc))]
    [id (v) (unbox (lookup v sc))]
    [set (var new-val-expr) ...]
    [fun (param body)
```

```
      (closureV param body sc)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr sc)]
                 [define arg-val (interp arg-expr sc)])
           (cases FWA-value fun-val
             [closureV (cl-param cl-body cl-cache) (interp cl-body
                                                           (aSub cl-param
                                                                 (box arg-val)
                                                                 cl-cache))]
             [else (error 'interp "can only apply functions")]))])))
```

Now, we turn to implementing set. What should set do? To set the value of a variable, we need to change the value stored in the box for that variable. This yields the following code:

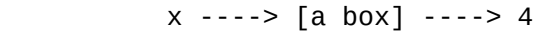
```
;; interp : FAE! SubCache → FWA-value
;; evaluates FAE! expressions by reducing them to their corresponding values
(define (interp expr sc)
  (cases FAE! expr
    [num (n) (numV n)]
    [add (l r) (numV+ (interp l sc) (interp r sc))]
    [sub (l r) (numV- (interp l sc) (interp r sc))]
    [id (v) (unbox (lookup v sc))]
    [set (var new-val-expr)
         (local ([define var-box (lookup var sc)]
                 [define new-val (interp new-val-expr sc)])
           (begin (set-box! var-box new-val)
                  new-val))]
    [fun (param body)
         (closureV param body sc)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr sc)]
                 [define arg-val (interp arg-expr sc)])
           (cases FWA-value fun-val
             [closureV (cl-param cl-body cl-cache) (interp cl-body
                                                           (aSub cl-param
                                                                 (box arg-val)
                                                                 cl-cache))]
             [else (error 'interp "can only apply functions")]))])))
```

Notice that we change the box contents, but not the environment. We don't need a separate traversal to find the box, because *lookup* already returns the box. Calling *set-box!* will change the contents of the box, and since the box is still referred to in the environment, the changes will be visible in the environment.

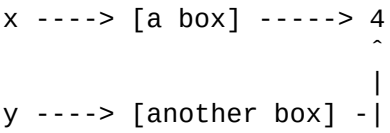
It may be easier to see how this code is working through some simple diagrams. Within an environment, each variable is associated with a box, and each box contains a value. Recall our example of

```
{with {x 4}
  {with {y x}
    {with {z {set x 5}}
      y}}}
```

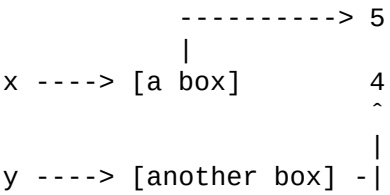
The environment is initially empty. After we bind *x* to 4, we have a picture like:



When we introduce the binding for y, we create a new box (in the app case) and set its contents to be the same contents as were in x's box (from the id case). The picture therefore becomes:



The set tells us to eval the expression and change what x's box refers to to the new contents:



This illustrates how we achieve the desired semantics for set. Calling *set* on *x* doesn't affect *y* because we change the contents of *x*'s box. If we wanted aliasing, the arrow from *y* would have pointed to the box for *x*, not to a different box with the same contents. Our other tricky example involved having sets affect the values stored in closures. This diagram also illustrates how our code handles this case: the environment stores the boxes, so changes to the boxes are picked up within the closures.

In programming language's parlance, the *environment* actually refers to the mapping from identifiers to boxes. The association between boxes and values is called a *store*. Introducing bindings affects the environment; mutation affects the store.