Chapter 13

Mutable Data Structures

Let's extend our source language to support boxes. Once again, we'll rewind to a simple language so we can study the effect of adding boxes without too much else in the way. That is, we'll define BCFAE, the combination of boxes, conditionals, functions and arithmetic expressions. We'll continue to use `with` expressions with the assumption that the parser converts these into function applications. In particular, we will introduce four new constructs:

```
<BCFAE> ::= ...
          | {newbox <BCFAE>}
          | {setbox <BCFAE> <BCFAE>}
          | {openbox <BCFAE>}
          | {seqn <BCFAE> <BCFAE>}
```

We can implement BCFAE by exploiting boxes in Scheme. This would, however, sheds little light on the nature of boxes. We should instead try to model boxes more explicitly.

What other means have we? If we can't use boxes, or any other notion of state, then we'll have to stick to mutation-free programs to define boxes. Well! It seems clear that this won't be straightforward.

Let's first understand boxes better. Suppose we write

(**define** *b1* (*box* 5))
(**define** *b2* (*box* 5))
(*set-box! b1* 6)
(*unbox b2*)

What response do we get?

This suggests that whatever is bound to *b1* and to *b2* must inherently be different. That is, we can think of each value being held in a different place, so changes to one don't affect the other.[1] The natural representation of a "place" in a modern computer is, of course, a memory cell.

[1] Here's a parable adapted from one I've heard ascribed to Guy Steele. Say you and I have gone on a trip. Over dinner, you say, "You know, I have a Thomas Jefferson $2 note at home!" That's funny, I say; so do I! We wonder whether it's actually the *same* $2 bill that we both think is ours alone. When I get home that night, I call my spouse and ask her to tear my $2 bill in half. You then call your spouse and ask, "Is our $2 bill intact?" Guy Steele is Solomonic.

13.1 Implementation Constraints

Before we get into the details of memory, let's first better understand the operational behavior of boxes. Examine this program:

```
{with {b {newbox 0}}
  {seqn {setbox b {+ 1 {openbox b}}}
        {openbox b}}}
```

which is intended to be equivalent to this Scheme program:

```
(local ([define b (box 0)])
  (begin
    (set-box! b (+ 1 (unbox b)))
    (unbox b)))
```

which evaluates to 1, that is, the mutation in the first operation in the sequence has an effect on the output of the second (which would otherwise have evaluated to 0). Now let's consider a naive interpreter for `seqn` statements. It's going to interpret the first term in the sequence in the environment given to the interpreter, then evaluate the second term in the same environment:

```
[seqn (e1 e2)
      (begin
        (interp e1 env)
        (interp e2 env))]
```

Besides the fact that this simply punts to Scheme's **begin** form, this *can't possibly be correct*! Why not? Because the environment is the only term common to the interpretation of *e1* and *e2*. If the environment is immutable—that is, it doesn't contain boxes—and if we don't employ any global mutation, then the outcome of interpreting the first sub-expression can't possibly have any effect on interpreting the second![2] Therefore, something more complex needs to happen.

One possibility is that we update the environment, and the interpreter always returns both the value of an expression *and* the updated environment. The updated environment can then reflect the changes wrought by mutation. The interpretation of `seqn` would then use the environment resulting from evaluating the first sequent to interpret the second.

While this is tempting, it can significantly alter the intended meaning of a program. For instance, consider this expression:

```
{with {a {newbox 1}}
  {seqn {with {b 3}
          b}
        b}}
```

[2] Depends on what we mean by "effect". The first branch of the sequence could, of course, fail to terminate or could result in an error, which are observable effects. But they are not effects that permit the evaluation of the second branch of the sequence.

This program should halt with an error, because static scope dictates that the second sequent (`b`) contains an unbound identifier. But passing the environment from the first sequent to the second would bind `b`. In other words, this strategy destroys static scope.

Even if we were to devise a sophisticated form of this environment-passing strategy (such as removing all new bindings introduced in a sub-expression), it still wouldn't be satisfactory. Consider this example:

```
{with {a {newbox 1}}
  {with {f {fun {x} {+ x {openbox a}}}}
    {seqn
      {setbox a 2}
      {f 5}}}}
```

We want the mutation to affect the box stored in the closure bound to `f`. But that closure already closes over the environment present at the time of evaluating the named expression—an environment that still reflects that `a` is bound to `1`. Even if we update the environment after the `setbox` operation, we cannot use the updated environment to evaluate the closure's body, at least not without (again!) introducing the potential to violate static scope.

As an aside, notice that in the program fragment above, changing the value of `a` is *not a violation of static scope*! The scoping rule only tells us where each identifier is bound; it does not (in the presence of mutation) fix the value bound to that identifier. To be pedantic, the value bound to the identifier does in fact remain the same: it's the same box for all time. The content of the box can, however, change over time.

We thus face an implementation quandary. There are two possible evaluation strategies for this last code fragment, both flawed:

- Use the environment (which maps `a` to `1`) stored in the closure for `f` when evaluating `{f 5}`. This will, however, ignore the mutation in the sequencing statement. The program will evaluate to `6` rather than `7`.
- Use the environment present at the time of procedure invocation: `{f 5}`. This will certainly record the change to `a` (assuming a reasonable adaptation of the environment), but this reintroduces dynamic scope.

To see the latter, we don't even need a program that uses mutation or sequencing statements. Even a program such as

```
{with {x 3}
  {with {f {fun {y} {+ x y}}}
    {with {x 5}
      {f 10}}}}
```

which should evaluate to `13` evaluates to `15` instead.

13.2 Insight

The preceding discussion does, however, give us some insight into a solution. It tells us that we need to have *two* repositories of information. One, the environment, is the guardian of static scope. The other

will become responsible for tracking dynamic changes. This latter entity is known in the parlance as the *store*. Determining the value bound to an identifier will become a two-step process: we will first use the environment to map the identifier to something that the store will then map to a value. What kind of thing is this intermediary? It's the index that identifies a mutable cell of memory—that is, it's a *memory location*.

Using this insight, we slightly alter our environments:

```
(define-type Env
  [mtSub]
  [aSub (name symbol?)
        (location number?)
        (env Env?)])
```

The store is really a partial function from address locations to values. This, too, we shall implement as a data structure.

```
(define-type Store
  [mtSto]
  [aSto (location number?)
        (value BCFAE-Value?)
        (store Store?)])
```

Correspondingly, we need two lookup procedures:

```
;; env-lookup : symbol Env → location

(define (env-lookup name env)
  (type-case Env env
    [mtSub () (error 'env-lookup "no binding for identifier")]
    [aSub (bound-name bound-location rest-env)
          (if (symbol=? bound-name name)
              bound-location
              (env-lookup name rest-env))]))

;; store-lookup : location Store → BCFAE-Value

(define (store-lookup loc-index sto)
  (type-case Store sto
    [mtSto () (error 'store-lookup "no value at location")]
    [aSto (location value rest-store)
          (if (= location loc-index)
              value
              (store-lookup loc-index rest-store))]))
```

Notice that the types of the two procedures compose to yield a mapping from identifiers to values, just like the erstwhile environment.

Let's now dive into the terms of the interpreter. We'll assume that two identifiers, *env* and *store*, are bound to values of the appropriate type. Some cases are easy: for instance,

[num (*n*) (numV *n*)]
[id (*v*) (*store-lookup* (*env-lookup v env*) *store*)]
[fun (*bound-id bound-body*)
  (closureV *bound-id bound-body env*)]

would all appear to be unchanged. Now consider the conditional:

[if0 (*test truth falsity*)
  (**if** (*numV-zero?* (*interp test env store*))
    (*interp truth env store*)
    (*interp falsity env store*))]

Suppose, with this implementation, we evaluate the following program:

```
{with {b {newbox 0}}
  {if0 {seqn {setbox b 5}
             {openbox b}}
       1
       {openbox b}}}
```

We would want this to evaluate to `5`. However, the implementation does not accomplish this, because mutations performed while evaluating the *test* expression are not propagated to the conditional branches.

In short, what we really want is a *(potentially) modified store* to result from evaluating the condition's test expression. It is this store that we must use to evaluate the branches of the conditional. But the ultimate goal of the interpreter is to produce answers, not just stores. What this means is that the interpreter must now return two results: the value corresponding to the expression, and a store that reflects modifications made in the course of evaluating that expression.

13.3 An Interpreter for Mutable Boxes

To implement state without relying on Scheme's mutation operations, we have seen that we must modify the interpreter significantly. The environment must map names to locations in memory, while the store maps these locations to the values they contain. Furthermore, we have seen that we must force the interpreter to return not only the value of each expression but also an updated store that reflects mutations made in the process of computing that value.

To capture this pair of values, we introduce a new datatype,[3]

(**define-type** Value×Store
  [v×s (*value BCFAE-Value?*) (*store Store?*)])

and reflect it in the type of the interpreter:

;; interp : BCFAE Env Store → Value×Store

Before defining the interpreter, let's look at how evaluation proceeds on a simple program involving boxes.

[3] In Scheme source programs, we would write Value×Store as `Value*Store` and v×s as `v*s`.

13.3.1 The Evaluation Pattern

Now that we have boxes in our language, we can model objects that have state. For example, let's look at a simple stateful object: a light switch. We'll use number to represent the state of the light switch, where `0` means off and `1` means on. The identifier `switch` is bound to a box initially containing `0`; the function `toggle` flips the light switch by mutating the value inside this box:

```
{with {switch {newbox 0}}
  {with {toggle {fun {dum}
                  {if0 {openbox switch}
                       {seqn
                         {setbox switch 1}
                         1}
                       {seqn
                         {setbox switch 0}
                         0}}}}
    ...}}
```

(Since `toggle` doesn't require a useful argument, we call its parameter `dum`.) The interesting property of `toggle` is that it can have different behavior on two invocations with the same input. In other words, the function has memory. That is, if we apply the function twice to the *same* (dummy) argument, it produces different values:

```
{with {switch {newbox 0}}
  {with {toggle {fun {dum}
                  {if0 {openbox switch}
                       {seqn
                         {setbox switch 1}
                         1}
                       {seqn
                         {setbox switch 0}
                         0}}}}
    {+ {toggle 1729}
       {toggle 1729}}}}
```

This expression should return `1`—the first application of `toggle` returns `1`, and the second returns `0`. To see why, let's write down the environment and store at each step.

The first `with` expression:

```
{with {switch {newbox 0}}
  ...}
```

does two things: it allocates the number `0` at some store location (say 100), and then binds the identifier `switch` to this location. We'll assume locations are represented using the boxV constructor, defined below. This gives the following environment and store:

env = [*switch* → 101]
store = [101 → (boxV 100), 100 → (numV 0)]

Notice that composing the environment and store maps `switch` to a box containing `0`.

After the second `with` expression:

```
{with {switch {newbox 0}}
  {with {toggle {fun {dum}
                  {if0 {openbox switch}
                       {seqn
                         {setbox switch 1}
                         1}
                       {seqn
                         {setbox switch 0}
                         0}}}}
    ...}}
```

the environment and store are:

env = [*toggle* → 102, *switch* → 101]
store = [102 → (closureV '{fun ...} [*switch* → 101]),
101 → (boxV 100),
100 → (numV 0)]

Now we come to the two applications of `toggle`. Let's examine the first call. Recall the type of *interp*: it consumes an expression, an environment, and a store. Thus, the interpretation of the first application is:

(*interp* '{toggle 1729}
[*toggle* → 102, *switch* → 101]
[102 → (closureV '{fun ...} [*switch* → 101]),
101 → (boxV 100),
100 → (numV 0)])

Interpreting `switch` results in the value (boxV 100), so interpreting `{openbox switch}` reduces to a store dereference of location 100, yielding the value (numV 0).

The successful branch of the `if0` expression:

```
{seqn
  {setbox switch 1}
  1}}
```

modifies the store; after the `setbox`, the environment and store are:

env = [*toggle* → 102, *switch* → 101]
store = [102 → (closureV '{fun ...} [*switch* → 101]),
101 → (boxV 100),
100 → (numV 1)]

For the first application of `toggle`, the interpreter returns a Value×Store where the value is (numV 1) and the store is as above.

Now consider the second application of *toggle*. It uses the store returned from the first application, so its interpretation is:

```
(interp '{toggle 1729}
        [toggle → 102, switch → 101]
        [102 → (closureV '{fun ...} [switch → 101]),
         101 → (boxV 100),
         100 → (numV 1)])
```

This time, in the body of `toggle`, the expression `{openbox switch}` evaluates to `1`, so we follow the failing branch of the conditional. The interpreter returns the value (numV 0) and a store whose location 100 maps to (numV 0).

Look carefully at the two (*interp* ···) lines above that evaluate the two invocations of `toggle`. Although both invocations took the *same* expression and environment, they were evaluated in different *stores*; that is the difference that led to the different results. Notice how the interpreter passed the store through the computation: it passed the original store from addition to the first `toggle` application, which return a modified store; it then passed the modified store to the second `toggle` application, which returned yet another store. The interpreter returned this final store with the sum of `0` and `1`. Therefore, the result of the entire expression is `1`.

13.3.2 The Interpreter

This style of passing the current store in and updated store out of every expression's evaluation is called *store-passing style*. We must now update the CFAE interpreter to use this style, and then extend it to support the operations on boxes.

Terms that are already syntactically values do not affect the store (since they require no further evaluation). Therefore, they return the store unaltered:

```
[num (n) (v×s (numV n) store)]
[id (v) (v×s (store-lookup (env-lookup v env) store) store)]
[fun (bound-id bound-body)
     (v×s (closureV bound-id bound-body env) store)]
```

The interpreter for conditionals reflects a pattern that will soon become very familiar:

```
[if0 (test truth falsity)
    (type-case Value×Store (interp test env store)
      [v×s (test-value test-store)
           (if (num-zero? test-value)
               (interp truth env test-store)
               (interp falsity env test-store))])]
```

In particular, note the store used to interpret the branches: It's the store that *results from evaluating the condition*. The store bound to *test-store* is "newer" than that bound to *store*, because it reflects mutations made while evaluating the test expression.

Exercise 13.3.1 *Modify this interpreter to use the wrong store—in this case, store rather than test-store in the success and failure branches—and then write a program that actually catches the interpreter producing faulty output. Until you can do this, you have not truly understood how programs that use state should evaluate!*

When we get to arithmetic expressions and function evaluation, we have a choice to make: in which order do we evaluate the sub-expressions? Given the program

```
{with {b {newbox 4}}
  {+ {openbox b}
     {with {dummy {setbox b 5}}
       {openbox b}}}}
```

evaluating from left-to-right yields `9` while evaluating from right-to-left produces `10`! We'll fix a left-to-right order for binary operations, and function-before-argument (also a kind of "left-to-right") for applications. Thus, the rule for addition is

```
[add (l r)
     (type-case Value×Store (interp l env store)
       [v×s (l-value l-store)
            (type-case Value×Store (interp r env l-store)
              [v×s (r-value r-store)
                   (v×s (num+ l-value r-value)
                        r-store)])])]
```

Carefully observe the stores used in the two invocations of the interpreter as well as the one returned with the resulting value. It's easy to make a mistake!

To upgrade a CFAE interpreter to store-passing style, we must also adapt the rule for applications. This looks more complex, but for the most part it's really just the same pattern carried through:

```
[app (fun-expr arg-expr)
     (type-case Value×Store (interp fun-expr env store)
       [v×s (fun-value fun-store)
            (type-case Value×Store (interp arg-expr env fun-store)
              [v×s (arg-value arg-store)
                   (local ([define new-loc (next-location arg-store)])
                     (interp (closureV-body fun-value)
                             (aSub (closureV-param fun-value)
                                   new-loc
                                   (closureV-env fun-value))
                             (aSto new-loc
                                   arg-value
                                   arg-store)))])])]
```

Notice that every time we extend the environment, we map the binding to a new location (using *next-location*, defined below). This new location is then bound to the result of evaluating the argument. As a result, tracing the formal parameter through both the environment and the store still yields the same result.

Finally, we need to demonstrate the interpretation of boxes. First, we must extend our notion of values:

```
(define-type BCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body BCFAE?)
            (env Env?)]
  [boxV (location number?)])
```

Given this new kind of value, let's study the interpretation of the four new constructs.

Sequences are easy. The interpreter evaluates the first sub-expression, ignores the resulting value (thus having evaluated the sub-expression only for the effect it might have on the store)—notice that *e1-value* is bound but never used—and returns the result of evaluating the second expression in the store (potentially) modified by evaluating the first sub-expression:

```
[seqn (e1 e2)
      (type-case Value×Store (interp e1 sc store)
        [v×s (e1-value e1-store)
             (interp e2 sc e1-store)])]
```

The essence of `newbox` is to obtain a new storage location, wrap its address in a boxV, and return the boxV as the value portion of the response accompanied by an extended store.

```
[newbox (value-expr)
        (type-case Value×Store (interp value-expr sc store)
          [v×s (expr-value expr-store)
               (local ([define new-loc (next-location expr-store)])
                 (v×s (boxV new-loc)
                      (aSto new-loc expr-value expr-store)))])]
```

To modify the content of a box, the interpreter first evaluates the first sub-expression to a location, then updates the store with a *new value for the same location*. Because all expressions must return a value, `setbox` chooses to return the new value put in the box as the value of the entire expression.

```
[setbox (box-expr value-expr)
        (type-case Value×Store (interp box-expr sc store)
          [v×s (box-value box-store)
               (type-case Value×Store (interp value-expr sc box-store)
                 [v×s (value-value value-store)
                      (v×s value-value
                           (aSto (boxV-location box-value)
                                 value-value
                                 value-store))])])]
```

Opening a box is straightforward: get a location, look it up in the store, and return the resulting value.

```
[openbox (box-expr)
         (type-case Value×Store (interp box-expr sc store)
```

```
[v×s (box-value box-store)
     (v×s (store-lookup (boxV-location box-value)
                        box-store)
          box-store)])]
```

Of course, the term "the store" is ambiguous, because there are different stores before and after evaluating the sub-expression. Which store should we use? Does this interpreter do the right thing?

All that remains is to implement *next-location*. Here's one implementation:

```
(define next-location
  (local ([define last-loc (box −1)])
    (lambda (store)
      (begin
        (set-box! last-loc (+ 1 (unbox last-loc)))
        (unbox last-loc)))))
```

This is an extremely unsatisfying way to implement *next-location*, because it ultimately relies on a box! However, this box is not essential. Can you get rid of it?

The core of the interpreter is in Figure 13.1 and Figure 13.2.

Exercise 13.3.2 *Define next-location so it does not have side-effects.*
Hint: *You may need to modify the interpreter to do this.*

13.4 Scope versus Extent

Notice that while closures refer to the environment of definition, they do not refer to the corresponding store. The store is therefore a global record of changes made during execution. As a result, stores and environments have different *patterns of flow*. Whereas the interpreter employs the same environment for both arms of an addition, for instance, it cascades the store from one arm to the next and then back out alongside the resulting value. This latter kind of flow is sometimes called *threading*, since it resembles the action of a needle through cloth.

These two flows of values through the interpreter correspond to a deep difference between names and values. A value persists in the store long after the name that introduced it has disappeared from the environment. This is not inherently a problem, because the value may have been the result of a computation, and some other name may have since come to be associated with that value. In other words, identifiers have *lexical scope*; values themselves, however, potentially have indefinite, *dynamic extent*.

Some languages confuse these two ideas. As a result, when an identifier ceases to be in scope, they remove the value corresponding to the identifier. That value may be the result of the computation, however, and some other identifier may still have a reference to it. This premature removal of the value will, therefore, inevitably lead to a system crash. Depending on how the implementation "removes" the value, however, the system may crash later instead of sooner, leading to extremely vexing bugs. This is a common problem in languages like C and C++.

In most cases, garbage collection (Section 23) lets languages dissociate scope from the reclamation of space consumed by values. The performance of garbage collectors is often far better than we might naïvely imagine (especially in comparison to the alternative).

In terms of language design, there are many reasons why C and C++ have adopted the broken policy of not distinguishing between scope and extent. These reasons roughly fall into the following categories:

- Justified concerns about fine-grained performance control.
- Mistakes arising from misconceptions about performance.
- History (we understand things better now than we did then).
- Ignorance of concepts that were known even at that time.

Whatever their reasons, these language design flaws have genuine and expensive consequences: they cause both errors and poor performance in programs. These errors, in particular, can lead to serious security problems, which have serious financial and social consequences. Therefore, the questions we raise here are not merely academic.

While programmers who are experts in these languages have evolved a series of ad hoc techniques for contending with these problems, we students of programming languages should know better. We should recognize their techniques for what they are, namely symptoms of a broken programming language design rather than proper solutions to a problem. Serious students of languages and related computer science technologies take these flaws as a starting point for exploring new and better designs.

Exercise 13.4.1 *Modify the interpreter to evaluate addition from right to left instead of left-to-right. Construct a test case that should yield different answers in the two cases, and show that your implementation returns the right value on your test case.*

Exercise 13.4.2 *Modify* `seqn` *to permit an arbitrary number of sub-expressions, not just two. They should evaluate in left-to-right order.*

Exercise 13.4.3 *New assignments to a location currently mask the old thanks to the way we've defined store-lookup, but the data structure still has a record of the old assignments. Modify the implementation of stores so that they have at most one assignment for each location.*

Exercise 13.4.4 *Use Scheme procedures to implement the store as a partial function.*

```
;; interp : BCFAE Env Store → Value×Store
(define (interp expr env store)
  (type-case BCFAE expr
    [num (n) (v×s (numV n) store)]
    [add (l r)
         (type-case Value×Store (interp l env store)
           [v×s (l-value l-store)
                (type-case Value×Store (interp r env l-store)
                  [v×s (r-value r-store)
                       (v×s (num+ l-value r-value)
                            r-store)])])]
    [id (v) (v×s (store-lookup (env-lookup v env) store) store)]
    [fun (bound-id bound-body)
         (v×s (closureV bound-id bound-body env) store)]
    [app (fun-expr arg-expr)
         (type-case Value×Store (interp fun-expr env store)
           [v×s (fun-value fun-store)
                (type-case Value×Store (interp arg-expr env fun-store)
                  [v×s (arg-value arg-store)
                       (local ([define new-loc (next-location arg-store)])
                         (interp (closureV-body fun-value)
                                 (aSub (closureV-param fun-value)
                                       new-loc
                                       (closureV-env fun-value))
                                 (aSto new-loc
                                       arg-value
                                       arg-store)))])])]
    [if0 (test truth falsity)
         (type-case Value×Store (interp test env store)
           [v×s (test-value test-store)
                (if (num-zero? test-value)
                    (interp truth env test-store)
                    (interp falsity env test-store))])]
    ⋮
```

Figure 13.1: Implementing Mutable Data Structures , Part 1

```
⋮
[newbox (value-expr)
        (type-case Value×Store (interp value-expr env store)
          [v×s (expr-value expr-store)
               (local ([define new-loc (next-location expr-store)])
                 (v×s (boxV new-loc)
                      (aSto new-loc expr-value expr-store)))])]
[setbox (box-expr value-expr)
        (type-case Value×Store (interp box-expr env store)
          [v×s (box-value box-store)
               (type-case Value×Store (interp value-expr env box-store)
                 [v×s (value-value value-store)
                      (v×s value-value
                           (aSto (boxV-location box-value)
                                 value-value
                                 value-store))])])]
[openbox (box-expr)
         (type-case Value×Store (interp box-expr env store)
           [v×s (box-value box-store)
                (v×s (store-lookup (boxV-location box-value)
                                   box-store)
                     box-store)])]
[seqn (e1 e2)
      (type-case Value×Store (interp e1 env store)
        [v×s (e1-value e1-store)
             (interp e2 env e1-store)])])
```

Figure 13.2: Implementing Mutable Data Structures, Part 2