Chapter 12

Church and State

In past programs we have freely employed Scheme boxes, but we haven’t really given an account of how they work (beyond an informal intuition). In this lecture, we will discuss the meaning of boxes. Boxes provide a way to employ mutation—the changing of values associated with names—to endow a language with state.

Mutation is a standard feature in most programming languages. The programs we have written in Scheme have, however, been largely devoid of state. Indeed, Haskell has no mutation operations at all. It is, therefore, possible to design and use languages—even quite powerful ones—that have no explicit notion of state. Simply because the idea that one can program without state hasn’t caught on in the mainstream is no reason to reject it.

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:

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

There are two instances of state in this fragment (the mutation of i and of sum), neither of which is es-
sential. Any other part of the program that depends on the value of \( \texttt{sum} \) remaining unchanged is going to malfunction. You might argue that at least the changes to \( \texttt{i} \) are innocuous, since the identifier is local to the block defined above; even that assumption, however, fails in a multi-threaded program! 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 \( \texttt{(build-list 99 add1)} \) generates \( \texttt{(list 0 \ldots 99)} \)) 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, if it proves to be more efficient, generate the mutation-based version from this specification.
Chapter 13

Implementing State

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:

\[
<\text{BCFAE}> ::= <\text{num}>
| \{+ <\text{BCFAE}> <\text{BCFAE}>\}
| \{- <\text{BCFAE}> <\text{BCFAE}>\}
| <\text{id}>
| \{\text{fun} \{<\text{id}>\} <\text{BCFAE}>\}
| \{<\text{FWAE}>, <\text{FWAE}>\}
| \{\text{if0} <\text{BCFAE}> <\text{BCFAE}> <\text{BCFAE}>\}
| \{\text{newbox} <\text{BCFAE}>\}
| \{\text{setbox} <\text{BCFAE}> <\text{BCFAE}>\}
| \{\text{openbox} <\text{BCFAE}>\}
| \{\text{seqn} <\text{BCFAE}> <\text{BCFAE}>\}
\]

We can, of course, implement BCFAE by exploiting boxes in Scheme. This would, however, be a meta-interpreter (Section 7) that 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 purely functional programming to define boxes. Well! It seems clear that this won’t be straightforward.

Let’s first understand boxes better. Suppose we write

\[
\text{(define } b1 \text{ (box 5))}
\text{(define } b2 \text{ (box 5))}
\text{(set-box! } b1 \text{ 6)}
\text{(unbox } b2)\]

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.

### 13.1 Implementation Constraints

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

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

which is intended to be equivalent to this Scheme program:

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

which evaluates to 1. 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:

\[
\text{seqn}(e_1 e_2) = (\begin{array}{c}
\text{interp } e_1 \text{ env} \\
\text{interp } e_2 \text{ env}
\end{array})
\]

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 \( e_1 \) and \( e_2 \). 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:

\(^1\)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.

\(^2\)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!) violating 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 strategy 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.