1 Background

In past lectures we have freely employed Scheme boxes, but we haven’t really given an account of how they work (beyond a vague 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. However, the dialect of Scheme we have used so far has, for the most part, been devoid of state. Indeed, Haskell has 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 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. If 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 1 ... 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.

¹The 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 Implementing Boxes: Introduction

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 \( \text{BCFAE} \), the combination of boxes, conditionals, functions and arithmetic expressions. We’ll continue to use \( \text{with} \) expressions with the assumption that the parser converts these into function applications. In particular, we will introduce four new constructs:

\[
\text{BCFAE} ::= \ldots
| \{\text{newbox } \text{<BCFAE>}\}
| \{\text{setbox } \text{<BCFAE> } \text{<BCFAE>}\}
| \{\text{openbox } \text{<BCFAE>}\}
| \{\text{seqn } \text{<BCFAE> } \text{<BCFAE>}\}
\]

(we’ve given these different names from the constructs in Scheme on purpose, to avoid name clashes both in our program and cognitively; this is why we called a function \text{fun} rather than \text{lambda}).

We can perform the usual sleight of hand, namely using a powerful meta-interpreter: that is, we can use Scheme’s boxes to implement boxes in the interpreted language. As we decided for recursion earlier, however, this would be a fairly pointless exercise, since it wouldn’t tell us any more about how boxes actually worked. Instead, we will try to model boxes through other means.

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

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

What response do we get?

This suggests that the whatever is bound to \( b1 \) and \( b2 \) must somehow be inherently different. That is, we can think of each value being held in a different place, so changes to one don’t affect the other.\(^2\) The natural representation of a “place” in a modern computer is, of course, a memory cell.

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

\[
\begin{align*}
\{\text{with } b \{\text{newbox 0}\}\} \\
\{\text{seqn } \{\text{setbox } b \{+ 1 \{\text{openbox } b\}\}\} \\
\{\text{openbox } b\}\}
\end{align*}
\]

which is intended to be equivalent to this Scheme program:

\[
\begin{align*}
\text{(local } \{(\text{define } b \text{ (box 0))}\}) \\
\text{(begin} \\
\text{\{\text{set-box! } b \{+ 1 \{\text{unbox } b\}\}\}) \\
\text{\{\text{unbox } b\})}
\end{align*}
\]

which evaluates to 1. Let’s consider a naïve interpreter for \text{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:

\[
\begin{align*}
\text{\{seqn } e1 \text{ e2}\} \\
\text{\{begin} \\
\text{\{interp } e1 \text{ env\}} \\
\text{\{interp } e2 \text{ env\}}
\end{align*}
\]

\(^2\)Here’s a parable that I’ve adapted from one I’ve heard ascribed to Guy Steele.

Say you and I both work in the same place, and are currently together on a trip abroad. Over dinner, you mention that you happen to have a Thomas Jefferson $2 bill. Why, 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 wife and ask her to create a distinctive tear in the corner of my $2 bill. You then call your spouse and ask, “Is our $2 bill intact?”

2
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! Therefore, something more complex needs to happen.

One possibility is that we update the environment somehow. That is, instead of the interpreter returning just the value of an expression, it returns the value and some updated environment. The update to the environment somehow reflects the changes wrought by mutation. This is tempting, until we consider an example such as this:

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

Notice that in this example, we really want the mutation to affect the box stored in the closure bound to `f`. This is not a violation of static scope! The scoping rule only tells us which identifier we will see at each location; it does not fix the value bound to that identifier.

We thus face an implementation quandary when it comes to implementing sequence statements. There are two possibilities:

- Use the environment (which maps `a` to a box containing `1`) stored in the closure for `f` when evaluating `{f 5}`. This will, however, ignore the change made to `a`’s box’s contents 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 sample program involving mutation. 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.

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 maintainer of static scope. The other will become responsible for tracking dynamic changes. This latter entity is memory, 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 a location, then use the store to map the location to a value. First, we slightly alter our environments:

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

On to memory. In a modern computer, memory is a mapping from addresses to the contents of memory cells. In particular, this mapping is a function. This suggests a natural representation of memory, namely as a Scheme procedure. This is not necessarily the most efficient representation of memory (for instance, it probably won’t yield random access), but we’ve already seen that it’s possible to improve a simple and inefficient representation of a function (the environment, specifically). So we could safely use a function as a representation of memory in our interpreter.

We could use such a representation, but for reasons we’ll see later in this course, we choose to use datatypes instead.

---

3This is not strictly true. Do you see why?
4To 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, however, does change over time.
(define-datatype Store Store? 
  [mtSto]
  [aSto (location number?)
      (value BCFA-value?)
      (store Store?)])

Correspondingly, we need two lookup procedures:

;; env-lookup : symbol Env → location 
(define (env-lookup name an-env)
  (cases Env an-env
    [mtSub () (error "env-lookup"
                  (string-append "unbound: " (symbol→string name)))]
    [aSub (bound-name bound-value rest-env)
      (if (symbol=? bound-name name)
          bound-value
          (env-lookup name rest-env))]))

;; store-lookup : location Store → BCFA-value 
(define (store-lookup location a-store)
  (cases Store a-store
    [mtSto () (error "store-lookup"
                 (string-append "invalid store location: "
                               (number→string location)))]
    [aSto (filled-location filled-value rest-store)
      (if (= location filled-location)
          filled-value
          (store-lookup location rest-store))]))

Notice that the types of the two procedures ensure that composing them appropriately will still map identifiers to values, as before.

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 (param body)
  (closureV param body env)]

would all appear to be unchanged. Let’s consider a slightly more complex term, namely the conditional. We might assume that it looks somewhat like this:

[if0 (test then else)
  (if (numV-zero? (interp test env store))
      (interp then env store)
      (interp else env store))]

This, however, does nothing to address the problem that forced us to introduce the store, namely that it must somehow reflect changes as the program evaluates. That is, given the program

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

we want this to evaluate to 5. This requires that the modification to the box bound to b be seen when computing the result of the sequence in the test expression; it means that the change must also persist into the else branch.
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 then or else branches. 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.