In the previous lecture, we discussed how to implement state in our language without resorting to Scheme’s boxes. The idea was to model memory as an abstract store that maps locations to values. The interpreter is then responsible for recording updates to the store; specifically, it takes a store as an additional argument, and returns both a value and an updated store.

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

```scheme
(define-datatype Value Store Value
  [(value BCFA-value?) (store Store?)])
```

and reflect it in the type of the interpreter:

```scheme
;; interp : BCFAE Env Store \rightarrow Value Store
```

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

## 1 An Example of Evaluation using Store-Passing Style

Now that we have boxes in our language, we can model things 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 variable `ls` is bound to a box initially containing 0; the function `toggle` flips the light switch by mutating the value inside this box:

```scheme
{with {ls {newbox 0}}
  {with {toggle {fun {dum}
    {if0 {openbox ls}
      {seqn
        {setbox ls 1}
        1}
      {seqn
        {setbox ls 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. To understand this behavior, we want to apply the function twice with the same (dummy) argument, and see why it returns different values. We can simply add the results of two applications:

```scheme
{with {ls {newbox 0}}
  {with {toggle {fun {dum}
    {if0 {openbox ls}
      {seqn
        {setbox ls 1}
        1}
      {seqn
        {setbox ls 0}
        0)}
    })}
  {if0 {openbox ls}
    {seqn
      {setbox ls 1}
      1}
    {seqn
      {setbox ls 0}
      0})}
```

...}}
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 \texttt{with} expression:

\begin{verbatim}
(with {ls {newbox 0}}
  ...
)
\end{verbatim}
does two things: it allocates the number 0 as some store location (say 100), and then binds the identifier \texttt{ls} to this location. At this stage, we have:

\[ env = [ls \rightarrow (locV 100)], \quad sto = [100 \rightarrow (numV 0)] \]

After the second \texttt{with} expression:

\begin{verbatim}
(with {ls {newbox 0}}
  {with {toggle {fun {dum}
        {if0 {openbox ls}
          {seqn
            {setbox ls 1}
            1}
          {seqn
            {setbox ls 0}
            0})}}}}
  ...
)
\end{verbatim}

the environment and store are:

\[ env = [ls \rightarrow (locV 100),\; \text{toggle} \rightarrow (closV \{\text{fun ...}\} [ls \rightarrow (locV 100)]),\; \text{toggle} \rightarrow \ldots] [100 \rightarrow (numV 0)] \]

\[ sto = [100 \rightarrow (numV 0)] \]

Now we come to the two applications of \texttt{toggle}. Let's examine the first call. Recall the type of \texttt{interp}: it consumes an expression, an environment, and a store. Thus, the interpretation of the first application looks like:

\[ (\text{interp } \{\text{toggle} 1729\} [ls \rightarrow (locV 100), \text{toggle} \rightarrow \ldots] [100 \rightarrow (numV 0)]) \]

In the body of \texttt{toggle}, we have:

\[ env = [ls \rightarrow (locV 100)] \]

\[ sto = [100 \rightarrow (numV 0)] \]

Hence, the expression \{\texttt{openbox ls}\} returns 0. The \texttt{then} branch of the \texttt{if0} expression:

\begin{verbatim}
{seqn
  {setbox ls 1}
  1}
\end{verbatim}

modifies the store; after the \texttt{setbox}, the environment and store are:

\[ env = [ls \rightarrow (locV 100)] \]

\[ sto = [100 \rightarrow (numV 1)] \]

For the first application of \texttt{toggle}, the interpreter returns a \texttt{Value} \texttt{Store} where the value is \( (numV 1) \) and the store is \[100 \rightarrow (numV 1)] \).

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

\[ (\text{interp } \{\text{toggle} 1729\} [ls \rightarrow (locV 100), \text{toggle} \rightarrow \ldots] [100 \rightarrow (numV 1)]) \]
This time, in the body of toggle, the expression \( \{\text{openbox } 1s\} \) evaluates to 1, so we follow the else branch. The interpreter returns the value \((\text{numV } 0)\) and the store \([100 \rightarrow (\text{numV } 0)]\).

Look carefully at the two \((\text{interp} \ldots)\) lines above which evaluate the two invocations of toggle. Although both invocations took the same argument, they were evaluated with different stores; thus, they returned 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 with the store \([100 \rightarrow (\text{numV } 0)]\).

## 2 Implementing the Interpreter

We have seen an example of how a store-passing interpreter evaluates an expression with boxes. Keep this example in mind as we turn to writing the interpreter. To start, recall the type of interp:

\[
;; \text{interp : BCFAE Env Store \rightarrow Value\$ Store}
\]

Terms that are already syntactically values do not affect the store (since they require no further evaluation). Therefore, these simply bundle the incoming store out as their outgoing store:

\[
[num \,(n) \,(v-s \,(\text{numV } \,n) \,\text{store})]
[\text{id} \,(v) \,(v-s \,(\text{env-lookup} \,v \,\text{env}) \,\text{store})]
[\text{fun} \,(\text{param} \,\text{body})]
\]

\[v-s \,(\text{closureV} \,\text{param} \,\text{body} \,\text{env}) \,\text{store}])\]

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

\[
[\text{if0} \,(\text{test} \,\text{then} \,\text{else})]
\]

\[
\begin{align*}
\text{cases} & \text{Value}\$\text{Store} \,(\text{interp} \,\text{test} \,\text{env} \,\text{store}) \\
\text{v-s} & \,(\text{test-val} \,\text{test-sto})
\end{align*}
\]

\[
\begin{align*}
\text{if} & \,(\text{numV-zero?} \,\text{test-val}) \\
\text{interp} & \,(\text{then} \,\text{env} \,\text{test-sto}) \\
\text{interp} & \,(\text{else} \,\text{env} \,\text{test-sto}))
\end{align*}
\]

In particular, note carefully the store used to interpret the then and else terms. It’s the store that results from evaluating the test term, not the value named by \text{store}. The store bound to \text{test-sto} is newer than that bound to \text{store}: it reflects mutations made while evaluating the test expression.\footnote{An extremely useful exercise is to download the interpreter, use the wrong store (for instance, use \text{store} instead of \text{test-sto} as the third argument to \text{interp} for the test expression), then write a test program that actually catches the interpreter producing faulty output. This will greatly help you understand how the store flows during the course of interpretation.}

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

\[
\begin{align*}
&\text{with} \,(\{b \,(\text{newbox} \,4)\}) \\
&\{\text{openbox} \,b\} \\
&\{\text{with} \,(\{\text{setbox} \,b \,5\}) \\
&\{\text{openbox} \,b\})\}
\end{align*}
\]

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 left-to-right) for applications. Thus, the rule for addition is

\[
\begin{align*}
\text{add} \,(l \,r) \\
\text{cases} & \text{Value}\$\text{Store} \,(\text{interp} \,l \,\text{env} \,\text{store})
\begin{align*}
\text{v-s} & \,(\text{lhs-val} \,\text{lhs-sto})
\end{align*}
\begin{align*}
\text{cases} & \text{Value}\$\text{Store} \,(\text{interp} \,r \,\text{env} \,\text{lhs-sto})
\begin{align*}
\text{v-s} & \,(\text{rhs-val} \,\text{rhs-sto})
\end{align*}
\begin{align*}
\text{v-s} & \,(\text{numV+} \,\text{lhs-val} \,\text{rhs-val} \,\text{rhs-sto}))
\end{align*}
\end{align*}
\]

\footnote{An extremely useful exercise is to download the interpreter, use the wrong store (for instance, use \text{store} instead of \text{test-sto} as the third argument to \text{interp} for the test expression), then write a test program that actually catches the interpreter producing faulty output. This will greatly help you understand how the store flows during the course of interpretation.}
Notice again the stores used in the two invocations of the interpreter as well as the one returned with the resulting value.

The last term in the existing interpreter that needs modification is that for applications. This looks more complex, but for the most part it’s really the same pattern carried through:

\[
\text{app (fun-exp expr arg-exp expr)} \\
\begin{cases}
\text{Value} \ast \text{Store (interp fun-exp env store)} \\
\text{v-s (fun-val fun-sto)} \\
\text{app (fun-exp expr arg-exp expr)} \\
\text{cases BCFA-value fun-val} \\
\text{[closureV (cl-param cl-body cl-env)} \\
\text{interp cl-body (aSub cl-param arg-val cl-env)} \\
\text{arg-sto]} \\
\text{[else (error "interp can only apply functions")]}])])
\end{cases}
\]

Finally, we need to demonstrate the interpretation of boxes. After all this groundwork, what remains is actually quite simple. First, we must extend our notion of values:

\[
\text{(define-datatype BCFA-value BCFA-value?)} \\
\begin{cases}
\text{numV (n number?)} \\
\text{closureV (param symbol?) (body BCFAE?) (env Env?)} \\
\text{locV (location number?)})
\end{cases}
\]

Given this new kind of value, here’s the interpretation of the four new constructs:

\[
\text{seqn (e1 e2)} \\
\begin{cases}
\text{Value} \ast \text{Store (interp e1 env store)} \\
\text{v-s (e1-val e1-sto)} \\
\text{interp e2 env e1-sto})]
\end{cases}
\]

Sequences are easy. The interpreter evaluates the first sub-expression, ignores the resulting value (thus doing so only for the effect it will have on the store), and returns the result of evaluating the second expression in the (potentially) modified store.

\[
\text{newbox (value-exp expr)} \\
\begin{cases}
\text{Value} \ast \text{Store (interp value-exp env store)} \\
\text{v-s (value-val value-sto)} \\
\text{local ([define new-loc (next-location value-sto)]}) \\
\text{v-s (locV new-loc)} \\
\text{aSto new-loc value-val value-sto))])}
\end{cases}
\]

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

\[
\text{setbox (box-exp expr value-exp)} \\
\begin{cases}
\text{Value} \ast \text{Store (interp box-exp expr box-sto)} \\
\text{v-s (box-val box-sto)} \\
\text{cases BCFA-value box-val} \\
\text{locV (box-loc)}
\end{cases}
\]
To modify the content of a box, the interpreter first ensures that the first sub-expression evaluates to a location (i.e. the \textit{locV} variant). If it does, it then updates the store with a \textit{new value for the same location} (notice that \texttt{aSto} maps \texttt{box-loc} to the new value, where \texttt{box-loc} is the very value obtained by opening the box value). Because all expressions return values, \texttt{setbox} chooses to return the new value put in the box as the value of the entire expression.

\begin{verbatim}
(openbox (box-expr)
  (cases Value·Store (interp box-expr env store)
    [v-s (box-val box-sto)]
    (cases BCFA-value box-val
      [locV (box-loc)
        (v-s (store-lookup box-loc box-sto)
          box-sto)]
      [else (error "openbox "expects box as arg")]]))
)
\end{verbatim}

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

All that remains is to implement \texttt{next-location}. Here's one implementation:

\begin{verbatim}
(define next-location
  (local ([define last-loc (box -1)])
    (lambda (store)
      (begin
        (set-box! last-loc (+ 1 (unbox last-loc)))
        (unbox last-loc)))))
\end{verbatim}

This might seem like an extremely unsatisfying way to implement \texttt{next-location}, because it ultimately relies on boxes! However, these boxes are quite inessential. As a puzzle, we ask you to consider a purely functional implementation of this procedure.

\section{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 \textit{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 \textit{threading}, since it vaguely resembles the action of a needle through cloth.\footnote{So why isn't it called \textit{sewing} instead? Beats me.} This interpreter is said to employ \textit{store-passing style}.

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, names of identifiers have \textit{lexical scope}; values themselves, however, have \textit{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 will 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 difficult bugs. This is a common problem in languages like C and C++.

There are many reasons why C and C++ have adopted these broken policies of not distinguishing between scope and extent. These reasons roughly fall into three 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, have both financial and social consequences (most significantly, serious security problems). 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 in this course should know better. We should recognize their techniques for what they are, which is 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.

Puzzles

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.

2. Implement the interpreter using Scheme procedures to represent memory.

3. Define \textit{next-location} in a purely functional manner.

4. We said initially that purely functional programming did not have state; yet we have used it to implement state. Doesn’t that mean it’s just as dangerous? Or could we, for instance, safely multi-thread the purely functional interpreter that implements state without having to worry about race conditions? Reconcile and explain.

5. Modify the implementation of stores so that they have at most one assignment for each location. (Currently, new assignments to a location mask the old thanks to the way we’ve defined \textit{store-lookup}, but the data structure still has a record of the old assignments.)

6. Modify \texttt{seqn} to permit an arbitrary number of sub-expressions, not just two. They should evaluate in left-to-right order.