Chapter 9

Implementing Recursion

We have now reduced the problem of creating recursive functions to that of creating cyclic environments.

The interpreter’s rule for \texttt{with} looked like this:

\[
\texttt{with (bound-id named-expr bound-body)}
\begin{array}{l}
(\text{interp bound-body}) \\
\quad \text{fun-defs} \\
\quad (\text{aSub bound-id}) \\
\quad (\text{interp named-expr}) \\
\quad \text{fun-defs} \\
\quad sc) \\
\text{sc})
\end{array}
\]

It is tempting to write something similar for \texttt{rec}, perhaps making a concession for the recursive environment by using a different constructor:

\[
\texttt{rec (bound-id named-expr bound-body)}
\begin{array}{l}
(\text{interp bound-body}) \\
\quad \text{fun-defs} \\
\quad \text{[recSub] bound-id} \\
\quad (\text{interp named-expr}) \\
\quad \text{fun-defs} \\
\quad sc)
\end{array}
\]

This is, however, unlikely to work correctly. The problem is that it interprets the named expression in the environment \texttt{env}. We have decided in Section 8.3 that the named expression must syntactically be a \texttt{fun} (using, say, the parser to enforce this restriction), which means its value is going to be a closure. That closure is going to capture its environment, which in this case will be \texttt{env}, the ambient environment. But \texttt{env} doesn’t have a binding for the identifier being bound by the \texttt{rec} expression, which means the function won’t be recursive. So this does us no good at all.

Rather than hasten to evaluate the named expression, we could pass the pieces of the function to the procedure that will create the recursive environment. When it creates the recursive environment, it can
generate a closure for the named expression that closes over this recursive environment. In code,

\[
\text{[rec (bound-id named-expr bound-body)}
\begin{align*}
\text{ (interp bound-body)}
\text{ (cyclically-bind-and-interp bound-id)}
\text{ named-expr}
\text{ sc})]
\end{align*}
\]

This puts the onus on \textit{cyclically-bind-and-interp}, but hopefully also gives it the pieces it needs to address the problem. That procedure is expected to create and return the appropriate environment, which associates the bound identifier with a closure whose environment is the containing environment.

Recall that Section \[7\] introduced a second representation for environments, so we now have two. We will study the implementation of recursion for each environment representation. In both cases, we’re trying to provide an implementation for a procedure with this type:

\[
;\;\text{cyclically-bind-and-interp : symbol \textit{fun} env \rightarrow env}
\]

(Section \[8.3\] explains why the second argument is guaranteed to be a \textit{fun} and not any other kind of expression.)

\section{9.1 Procedural Representation of Recursive Environments}

Assume that we’re using Scheme procedures to represent environments. Clearly, \textit{cyclically-bind-and-interp} must begin as follows:

\[
\text{(define (cyclically-bind-and-interp bound-name named-expr env)}
\begin{align*}
\text{ ...)}
\end{align*}
\]

We know that the following code pattern must exist because of how procedural representations of environments work:

\[
\text{(define (cyclically-bind-and-interp bound-name named-expr env)}
\begin{align*}
\text{ :)
\text{ (lambda (want-name)}
\text{ (cond)
\text{ [(symbol=? want-name bound-name)}
\text{ ...]
\text{ [else (lookup want-name env)]})
\text{ ...)}
\end{align*}
\]

If the symbols match, what do we want to return? Looking up identifiers in environments produces values. Recall that the named expression must be a function, so its value must be a closure. Thus, the response if the symbols match must yield a closure:

\[
\text{(define (cyclically-bind-and-interp bound-name named-expr env)}
\begin{align*}
\text{ :)
\text{ (lambda (want-name)
\end{align*}
\]

What’s not yet clear is what environment to close over. It clearly can’t be `env` (defeats the purpose of `rec`), and it must be something with this additional binding. So how about we give a name to this new environment that knows about the binding for `bound-name`?

```scheme
(define (cyclically-bind-and-interp bound-name named-expr env)
  (define rec-ext-env
    (lambda (want-name)
      (cond
        [(symbol=? want-name bound-name)
          (closureV (fun-param named-expr)
            (fun-body named-expr)
            ...]
        [else (lookup want-name env)])))))

Having named it, it’s now easy to fill in the two ellipses. What environment do we want to close over in the closure? One that has a binding for the function named in `bound-name`. This is the environment `rec-ext-env`. What do we want to return from this procedure? The recursively extended environment. This is also `rec-ext-env`. Thus, ignoring the box momentarily,

```scheme
(define (cyclically-bind-and-interp bound-name named-expr env)
  (define rec-ext-env
    (lambda (want-name)
      (cond
        [(symbol=? want-name bound-name)
          (closureV (fun-param named-expr)
            (fun-body named-expr)
            ...]
        [else (lookup want-name env)])))))

The relevant portions of the interpreter are in Figure 9.1 and Figure 9.2.

This definition raises two natural questions:

1. Is this really a recursive environment? Yes it is, though you’ll just have to take the word of the authors of DrScheme that `local` does indeed define `rec-ext-env` as a recursive procedure, so references to that name in the procedure’s body will indeed refer back to the same procedure.
2. Doesn’t the boxed reference to \textit{rec-ext-env} have the same problem we were trying to avoid with expressions such as \{rec \{x \ x\} x\}? Actually, it doesn’t. The reference here is “under a lambda”, that is, it is separated from the binding instance by a procedure declaration. Therefore, when the named expression portion of the \texttt{local} is evaluated, it associates a closure with \textit{rec-ext-env} that doesn’t get invoked until much later—by which time the recursive environment of the \texttt{local} is safely defined. This is the same issue we discussed in Section 8.3.

Reassuring as these responses may be, there is still something deeply unsatisfying about this solution. We set out to add recursive functions to RCFAE. We reduced this to the problem of defining recursive environments, which is legitimate (and, arguably, recursive environments are easier to think about than recursive functions themselves). But we then \texttt{implemented} recursive environments by falling right back on Scheme’s recursive functions: an abuse of meta-interpretive power, if ever there was any! What we’d like is a much more coherent, self-contained account of \texttt{rec} that doesn’t rely on advanced knowledge of Scheme (or, at least, no knowledge of features that we don’t also find in more mainstream programming languages).

As an aside, this discussion highlights both a power and peril of meta-interpretive choices. The power of choosing the procedural representation is that we can add recursion to the language very easily. If our goal is to add it as quickly as possible, while minimizing error, it makes sense to exploit the effort put into implementing recursion for Scheme. But the peril is that this implementation does not hold descriptive power: it still begs the question of how to implement recursion from scratch.

\textbf{Problem 9.1.1} \textit{Is it possible to implement recursive environments using the procedural representation without employing Scheme’s constructs for creating recursive procedures? That is, can FAE alone express recursive functions?}

\section{Datatype Representation of Recursive Environments}

Let’s now turn our attention to the datatype representation of environments. In the procedural case we extended the environment to capture cyclic bindings, so we do the same here:

\begin{verbatim}
(define-type SubCache
  [mtSub]
  [aSub (name symbol?)
      (value RCFAE-Value?)
      (env SubCache?)]
  [aRecSub (name symbol?)
      (value RCFAE-Value?)
      (env SubCache?)])
\end{verbatim}

We will again push the work from the interpreter into the helper function:

\begin{verbatim}
[rec (bound-id named-expr bound-body)
  (interp bound-body
   (cyclically-bind-andinterp bound-id
    named-expr
    se))])
\end{verbatim}
so the interesting work happens in cyclically-bind-and interp.

Before we can create a cyclic environment, we must first extend it with the new variable. We don’t yet know what it will be bound to, so we’ll stick a dummy value into the environment:

`; cyclically-bind-and-interp : symbol RCFAE env → env`

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (numV 1729)]
    [define new-env (aRecSub bound-id value-holder env)])
    ...))
```

If the program uses the identifier being bound before it has its real value, it’ll get the dummy value as the result. But because we have assumed that the named expression is syntactically a function, this can’t happen.

Now that we have this extended environment, we can interpret the named expression in it:

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1729)])
    [define new-env (aRecSub bound-id value-holder env)]
    [define named-expr-val (interp named-expr new-env)])
    ...))
```

Because the named expression is a closure, it will close over the extended environment (new-env). Notice that this environment is half-right and half-wrong: it has the right names bound, but the newest addition is bound to the wrong (indeed, dummy) value.

Now comes the critical step. The value we get from evaluating the named expression is the same value we want to get on all subsequent references to the name being bound. Therefore, the dummy value—the one bound to the identifier named in the rec—needs to be replaced with the new value.

To perform this replacement, we need to ensure that the environment is mutable. We will use Scheme values known as boxes to implement this. The type of the value field of a aRecSub therefore really needs to satisfy this predicate:

```
(define (boxed-RCFAE-Value? v)
  (and (box? v)
    (RCFAE-Value? (unbox v))))
```

This forces us to box the dummy value also:

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1729)])
    [define new-env (aRecSub bound-id value-holder env)])
    ...))
```

1 Suppose we lifted this restriction on the named expression. In a more sophisticated implementation, we would then introduce a special kind of value that designates “there’s no value here (yet)”; when a computation produces that value, the evaluator should halt with an error.

2 A Scheme box is a mutable cell. Boxes have three operations: box : Value → box, which creates a fresh cell containing the argument value; unbox : box → Value, which returns the value stored in a box; and set-box! : box Value → void, which changes the value held in a box but returns no value of interest.
[define named-expr-val (interp named-expr new-env)]

Now that we have a box in the environment, it’s ripe for mutation:

(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1729))]
         [define new-env (aRecSub bound-id value-holder env)]
         [define named-expr-val (interp named-expr new-env)])
    (set-box! value-holder named-expr-val)))

Since any closures in the value expression share the same binding, they automatically avail of this update. Finally, we must remember that cyclically-bind-and-interp has to actually return the updated environment for the interpreter to use when evaluating the body:

(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1729))]
         [define new-env (aRecSub bound-id value-holder env)]
         [define named-expr-val (interp named-expr new-env)])
    (begin
      (set-box! value-holder named-expr-val)
      new-env)))

There’s one last thing we need to do. Because we have introduced a new kind of environment, we must update the environment lookup procedure to recognize it.

[aRecSub (bound-name boxed-bound-value rest-sc)
  (if (symbol=? bound-name name)
    (unbox boxed-bound-value)
    (lookup name rest-sc))]

This only differs from the rule for aSub in that we must remember that the actual value is encapsulated within a box. Figure 9.1 and Figure 9.3 present the resulting interpreter.

Working through our factorial example from earlier, the ambient environment is (mtSub), so the value bound to new-env in cyclically-bind-and-interp is

(aRecSub 'fac
  (box (numV 1729))
  (mtSub))

Next, named-expr-val is bound to

(closureV 'n
  (if0 · · ·)
  (aRecSub 'fac
    (box (numV 1729))
    (mtSub)))
Now the mutation happens. This has the effect of changing the value bound to `fac in the environment:

\[
\begin{align*}
    (aRecSub \ 'fac \\
    \quad (box \ (closureV \ \cdots)) \\
    \quad (mtSub))
\end{align*}
\]

But we really should be writing the closure out in full. Now recall that this is the same environment contained in the closure bound to `fac. So the environment is really

\[
\begin{align*}
    (aRecSub \ 'fac \\
    \quad (box \ (closureV \ 'n \\
    \quad \quad (if0 \ \cdots) \\
    \quad \quad \Box)) \\
    \quad (mtSub))
\end{align*}
\]

where \(\Box\) is a reference back to this very same environment! In other words, we have a cyclic environment that addresses the needs of recursion. The cyclicity ensures that there is always “one more binding” for \(\text{fac}\) when we need it.

**Problem 9.2.1** The two implementations differ slightly in the way they treat illegal named expressions (i.e., ones that are not syntactic procedures). Do you see why? How would you make them behave identically?
;; interp : FAE env \rightarrow RCFAE-Value

(define (interp expr sc)
  (type-case RCFAE expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l sc) (interp r sc))]
    [sub (l r) (num− (interp l sc) (interp r sc))]
    [mult (l r) (num∗ (interp l sc) (interp r sc))]
    [if0 (test then else)
      (if (num-zero? (interp test sc))
        (interp then sc)
        (interp else sc))]
    [id (v) (lookup v sc)]
    [fun (bound-id bound-body)
      (closureV bound-id bound-body sc)]
    [app (fun-expr arg-expr)
      (local (define fun-val (interp fun-expr sc)))
      (interp (closureV-body fun-val)
        (aSub (closureV-param fun-val)
          (interp arg-expr sc)
          (closureV-sc fun-val))))]
    [rec (bound-id named-expr bound-body)
      (interp bound-body
        (cyclically-bind-and-interp bound-id
          named-expr
          sc))])

Figure 9.1: Common Core Interpreter for Recursion
9.2. DATATYPE REPRESENTATION OF RECURSIVE ENVIRONMENTS

(define-type RCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
    (body RCFAE?)
    (sc SubCache?)])

;; cyclically-bind-and-interp : symbol fun env → env

(define (cyclically-bind-and-interp bound-name named-expr env)
  (local ((define rec-ext-env
    (lambda (want-name)
      (cond
        [(symbol=? want-name bound-name)
          (closureV (fun-param named-expr)
            (fun-body named-expr)
            rec-ext-env))
        [else (lookup want-name env))]))
    rec-ext-env))

Figure 9.2: Recursion with Procedural Representation of Environments
(define-type RCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
    (body RCFAE?)
    (sc SubCache?)])

(define (boxed-RCFAE-Value? v)
  (and (box? v)
       (RCFAE-Value? (unbox v))))

(define-type SubCache
  [mtSub]
  [aSub (name symbol?)
    (value RCFAE-Value?)
    (env SubCache?)]
  [aRecSub (name symbol?)
    (value boxed-RCFAE-Value?)
    (env SubCache?)])

;; lookup : symbol env → RCFAE-Value

(define (lookup name sc)
  (type-case SubCache sc
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-sc)
      (if (symbol=? bound-name name)
          bound-value
          (lookup name rest-sc))]
    [aRecSub (bound-name boxed-bound-value rest-sc)
      (if (symbol=? boxed-bound-value (unbox boxed-bound-value))
          (lookup name rest-sc))])
  )

;; cyclically-bind-and-interp : symbol RCFAE env → env

(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ((define value-holder (box (numV 1729)))
    [define new-env (aRecSub bound-id value-holder env)]
    [define named-expr-val (interp named-expr new-env)])
  (begin
    (set-box! value-holder named-expr-val)
    new-env))

Figure 9.3: Recursion with Data Structure Representation of Environments