Chapter 10

Implementing Recursion

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

The interpreter's rule for `with` in Figure 5.2 was as follows:

```
[with (bound-id named-expr bound-body)
     (interp bound-body
           (aSub bound-id
                 (interp named-expr
                       ds)
                 ds))]
```

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

```
[rec (bound-id named-expr bound-body)
     (interp bound-body
           (recSub bound-id
                 (interp named-expr
                       ds)
                 ds))]
```

Unfortunately, this suffers from a fatal flaw. The problem is that it interprets the named expression in the environment *ds*. We have decided in Section 9.3 that the named expression must syntactically be a `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 *ds*, the ambient environment. But *ds* doesn't have a binding for the identifier being bound by the `rec` expression, which means the function won't be recursive. So this attempt cannot succeed.

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,

```
[rec (bound-id named-expr bound-body)
     (interp bound-body
```

```
(cyclically-bind-and-interp bound-id
                            named-expr
                            env))]
```

(Recall that *ds* is the old name for *env*.) This puts the onus on *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.

Let's turn our attention to *cyclically-bind-and-interp*. First, let's make a note of its contract:

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

(Section 9.3 explains why the second argument should be a fun and not any other kind of expression.)

Before we can create a cyclic environment, we must first extend it with a binding for the new identifier. At this point we know the identifier's name but not necessarily the value bound to it, so we'll place a dummy value in the environment:

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

The named expression evaluates to a closure, which 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 (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. (We didn't have this value before, which is why we had to place a dummy value in the environment.) 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*. To make it mutable, we must use a different kind of value in the environment, ideally a Scheme *box*.[1] Unfortunately, using a box rather than a *RCFAE-Value?* would violate the contract for aSub. Therefore, we will need to add a new variant to the datatype for environments; let's call it an aRecSub, to commemorate the construct that motivated its introduction. An aRecSub is just like an aSub, except its *value* must satisfy this predicate:

[1] 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 (boxed-RCFAE-Value? v)
  (and (box? v)
       (RCFAE-Value? (unbox v))))
```

Consequently, the proper definition of an environment is

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

Now we have to rewrite the code we've written so far, using aRecSub instead of aSub, and boxing the dummy value:

```
(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)])
    ...))
```

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-env)
```

(**if** (*symbol=? bound-name name*)
(*unbox boxed-bound-value*)
(*lookup name rest-env*))]

This only differs from the rule for aSub in that we must remember that the actual value is encapsulated within a box. Figure 10.1 and Figure 10.2 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:

```
(aRecSub 'fac
         (box (closureV ...))
         (mtSub))
```

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

```
(aRecSub 'fac
         (box (closureV 'n
                        (if0 ...)
                        □))
         (mtSub))
```

where □ 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 `fac` when we need it.

Exercise 10.0.3 *Were we able to implement recursive definitions in* F1WAE *(Section 4)? If so, how was that possible without all this machinery?*

Exercise 10.0.4 *Lift this restriction on the named expression. 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.*

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

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

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

;; lookup : symbol env → RCFAE-Value
(define (lookup name env)
  (type-case Env env
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-env)
          (if (symbol=? bound-name name)
              bound-value
              (lookup name rest-env))]
    [aRecSub (bound-name boxed-bound-value rest-env)
             (if (symbol=? bound-name name)
                 (unbox boxed-bound-value)
                 (lookup name rest-env))]))

;; 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 10.1: Recursion: Support Code

```
;; interp : RCFAE env → RCFAE-Value
(define (interp expr env)
  (type-case RCFAE expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l env) (interp r env))]
    [sub (l r) (num- (interp l env) (interp r env))]
    [mult (l r) (num* (interp l env) (interp r env))]
    [if0 (test truth falsity)
         (if (num-zero? (interp test env))
             (interp truth env)
             (interp falsity env))]
    [id (v) (lookup v env)]
    [fun (bound-id bound-body)
         (closureV bound-id bound-body env)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr env)])
           (interp (closureV-body fun-val)
                   (aSub (closureV-param fun-val)
                         (interp arg-expr env)
                         (closureV-env fun-val))))]
    [rec (bound-id named-expr bound-body)
         (interp bound-body
                 (cyclically-bind-and-interp bound-id
                                             named-expr
                                             env))]))
```

Figure 10.2: Recursion: Interpreter