Implementing Recursion

Shriram Krishnamurthi

2003-09-19

We’ve seen that there are (at least) two, fairly distinct ways of representing environments. To implement recursive environments, we need to provide an implementation for each representation. In both cases, we’re trying to provide an implementation for the procedure with this type.

;; aRecSub: symbol symbol RCFAE env → env

1 Procedural Representation of Recursive Environments

Assume that we’re using Scheme procedures to represent environments. Let’s construct this stepwise.

Clearly, \texttt{aRecSub} must begin as follows:

\begin{verbatim}
(define (aRecSub bound-name fun-param fun-body env)
  
  ;

  ;

  ;

  lambda (want-name)
    (cond
      [(symbol= bound-name want-name)]
      [else (lookup want-name env)])

)
\end{verbatim}

We know that somewhere, the following code pattern must reside by nature of the procedural representation of environments:

\begin{verbatim}
(define (aRecSub bound-name fun-param fun-body env)
  
  ;

  ;

  ;

  lambda (want-name)
    (cond
      [(symbol= bound-name want-name)]
      [else (lookup want-name env)])

)
\end{verbatim}

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:

\begin{verbatim}
(define (aRecSub bound-name fun-param fun-body env)
  
  ;

  ;

  ;

  lambda (want-name)
    (cond
      [(symbol= bound-name want-name)]
      [else (lookup want-name env)])

)
\end{verbatim}

What’s not yet clear is what environment to close over. It clearly can’t be \texttt{env} (defeats the purpose of \texttt{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 \texttt{bound-name}?

\begin{verbatim}
(define (aRecSub bound-name fun-param fun-body env)
  
  ;

  ;

  ;

  lambda (want-name)
    (cond
      [(symbol= bound-name want-name)]
      [else (lookup want-name env)])

)
\end{verbatim}
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,

\[
\text{(define (aRecSub bound-name fun-param fun-body env)}
\]

\[
\text{(local (define rec-ext-env)}
\]

\[
\text{(lambda (want-name)}
\]

\[
\text{(cond}
\]

\[
\text{[(symbol=? want-name bound-name)}
\]

\[
\text{(closureV fun-param fun-body rec-ext-env]}]
\]

\[
\text{[else (lookup want-name env)])})}
\]

\[
\text{rec-ext-env])}
\]

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 rec-ext-env have the same problem we were trying to avoid with expressions such as \( \{ \text{rec } (x \ x) \ x \} \)? 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 local is evaluated, it associates a closure with rec-ext-env that doesn’t get invoked until much later—by which time the recursive environment of the local is safely defined.

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 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 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).

Puzzle

[Note: This is a very difficult exercise!] Is it possible to implement recursive environments using the procedural representation without employing Scheme’s constructs for creating recursive procedures? That is, can lambda alone do the trick?

2 Datatype Representation of Recursive Environments

Let’s now turn our attention to the datatype representation of environments. First, we extend the datatype:

\[
\text{(define-datatype Env Env?}
\]

\[
\text{[mtSub]}
\]

\[
\text{[aSub (name symbol?)}
\]

\[
\text{ (value RCFA-value?)}
\]

\[
\text{ (env Env?)]}
\]

\[
\text{[aRecSub (name symbol?)}
\]

\[
\text{ (value ???)].}
\]
What is the type of the value field? Obviously it must include a RCFA-value, but in fact we’ll need a little more than that, as we will discover shortly.

Assuming we have RCFA-value as the type of the value field, let’s start writing the interpreter. We can be a bit lazy in the core interpreter, pushing all the work onto a helper function. Thus,

```scheme
(define (interp expr env)
  (cases RCFAE expr
    ;
    [rec (bound-id named-expr bound-body)
      (cases RCFAE named-expr
        [fun (fun-param fun-body)
          (interp bound-body
            (cyclically-bind-and-interp bound-id named-expr env))]
        [else (error "rec "expecting a syntactic function")]))
    ;
  )
)
```

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:

```scheme
;; 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))
  ])
    [named-expr-val (interp named-expr new-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 syntactically verified that the named expression is a function, this can’t happen.¹ (Why not?)

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

```scheme
(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))
  ])
    [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.² The type of the value field of a `aRecSub` therefore really needs to be

```scheme
(lambda (x)
  (and (box? x)
    (RCFA-value? (unbox x))))
```

¹Suppose 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”; when a computation produces that value, the evaluator should halt with an error.

²A 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.
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)]
    [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 interp-and-cyclic-bind 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.

```
(define (lookup name an-env)
  (cases Env an-env)
    ...
    [aRecSub (bound-name bound-value-box rest-env)
      (if (symbol=? bound-name name)
        (unbox bound-value-box)
        (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.

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 ostensibly 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 closure bound to `fac`. So the environment is really

\[
(aRecSub \ 'fac \\
  (box (closureV \ 'n \\
    (if0 \ \ldots) \\
    □)) \\
  (mtSub))
\]

where □ is a reference back to this very same environment! Writing this out one more level,

\[
(aRecSub \ 'fac \\
  (box \ (closureV \ 'n \\
    (if0 \ \ldots) \\
    (aRecSub \ 'fac \\
      (box \ (closureV \ 'n \\
        (if0 \ \ldots) \\
        (aRecSub \ 'fac \\
          (box \ □) \\
          (mtSub)))))) \\
  (mtSub))
\]

and so on. 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.

**Puzzles**

In the datatype implementation of environments:

1. Does anything in the rest of the code depend on the named expression being a syntactic function? Put differently, can we remove the check for a syntactic function in the `rec` clause of the interpreter? Are there non-syntactic expressions that can evaluate safely?

2. If we remove this check, what happens to programs that try to access the identifier being bound prematurely? Is this behavior desirable and, if not, what changes would you make to improve it?