An Explicit-Continuation Metacircular Evaluator

The vanilla metacircular evaluator gives a lot of information about environments, but not much information about control—when the evaluation of a subexpression returns a value, what happens next? In the explicit-continuation evaluator given here, a new parameter to the `eval` procedure—called a continuation—is a procedure that tells exactly what to do next. The evaluation of every expression takes place in the context of some continuation.

This continuation is thus a sort of function that takes a value produced by a computation, and then does something with the value. For example, in the `read-eval-print` loop, at the point of evaluation, the continuation is “given value v, print it, then go through the loop again.”

Since `lambda` is how we describe functions (procedures) in Scheme, let’s use them to describe continuations. Simplify the above continuation to `(lambda (v) v)`—just return the value. When evaluating the recursive call `(fact 2)` in the evaluation of `(fact 4)`, the continuation is then (effectively) `(lambda (v) (* 4 (* 3 v)))`.

Here, in fact, is an explicit-control version of `factorial`, analogous to the explicit-control evaluator:

```scheme
(define (factorial n k)
  (if (= n 0)
      (k 1)
      (factorial (- n 1) (lambda (v) (k (* n v))))))
```

Notice that the recursive call is tail recursive, and thus creates an iterative process. To compute factorial of 3, we evaluate:

```scheme
(factorial 3 (lambda (v) v))
```
How does this computation take place in the substitution model? We have:

(define (factorial n k)
  (if (= n 0)
      (k 1)
      (factorial (- n 1) (lambda (v) (k (* n v))))))

(factorial 3 (lambda (v) v))
(factorial 2 (lambda (v)
      ((lambda (v') v') (* 3 v))))
(factorial 2 (lambda (v) (* 3 v)))
(factorial 1 (lambda (v)
      ((lambda (v') (* 3 v')) (* 2 v))))
(factorial 1 (lambda (v) (* 3 (* 2 v))))
(factorial 0 (lambda (v)
      ((lambda (v') (* 3 (* 2 v'))) (* 1 v))))
(factorial 0 (lambda (v) (* 3 (* 2 (* 1 v)))))
((lambda (v) (* 3 (* 2 (* 1 v)))) 1)
6

(Note some v have been changed to v' so you don’t get confused by multiple binding of the same variable.)

This implementation of factorial is said to be tail recursive because no “extra work” accumulates outside the call to factorial—the procedure is able to maintain all data relevant to the computation in its two arguments.\[1\]

We are going to design a metacircular evaluator that has an explicit continuation—every expression is evaluated in the context of an environment and a continuation. In this initial presentation, we simplify the language we’re evaluation to make the evaluator easier to read.

\[1\] Of course, the fiction of “constant space” is evident, since the size of these two arguments is growing.
**Simplifications:** Definitions are written (define ⟨name⟩ ⟨body⟩); all procedures have only one argument. Primitive procedures have been curried: we write ((* 3) 5) instead of (* 3 5). For example:

```scheme
(define fact
  (lambda (n) (if ((= n) 0) 1 ((* n) (fact ((- n) 1))))))
```

The existence of the explicit control lets us introduce two (non Scheme standard) constructs: enter and exit. Here's an example of what they do:

```scheme
(try '(enter ((+ 5) (exit 3))))
;Value: 3
```

```scheme
(((lambda (x)
    (lambda (y) ((* ((+ x) y))
        (enter ((- x) (exit y))))))
  10)
  2))
;Value: 24
```

Constructs like enter and exit are useful when designing and implementing **error handlers** for systems where we need to “jump out” of a control context, for example:

```scheme
(define (lookup-variable v env-names)
  (if (null? env-names)
      (error "Name not in list")
      (if (eq? v (car env-names))
          0
          (1+ (lookup-variable v (cdr env-names))))))
```

(How would you design an error handler for your metacircular evaluator that bumps the user up to the read-eval-print loop of the evaluator, and not the underlying Scheme system?)
(define (eval exp env cont)
  (cond ((constant? exp) (cont exp))
       ((variable? exp) (cont (lookup env exp)))
       ((define? exp)
         (let ((new-env (extend (def-name exp)
                                 '*UNEVALUATED*
                                 env)))
           (eval (def-body exp)
                 new-env
                 (lambda (v)
                   (set-cdr! (car new-env) v)
                   (cont (cons '*define* new-env)))))
       ((if? exp)
         (eval (predicate exp)
               env
               (lambda (p)
                 (eval (if p (then-part exp) (else-part exp))
                       env
                       cont))))
       ((enter? exp)
         (eval (enter-body exp)
               (extend '*exit-continuation* cont env)
               cont))
       ((exit? exp)
         (eval (exit-body exp)
               env
               (lookup env '*exit-continuation*))))
In this simple language, all procedures have one argument. A lambda-expression is then interpreted as a lambda-expression in the underlying Scheme system with two parameters x and k: the procedure implements a function which needs to know two things—its input (named by x) and what to do with its output (named by continuation k).

The continuation cont is applied to the procedure when it is (statically) defined, answering the question “what do we do with the procedure when it is created?” The continuation k is applied to the value returned by the procedure body when it is (dynamically) used.

(((lambda? exp)
 (cont (lambda (x k)
 (eval (body exp)
 (extend (binder exp) x env)
 k)))))

The above definition makes sense in the context of the dual definition of function application. The function is evaluated with the continuation of evaluating the argument; the continuation for the latter is to apply the evaluated function to the evaluated argument with the current continuation cont—so we know what to do with the answer returned by the function application.

(((application? exp)
 (eval (function-of exp)
 env
 (lambda (f)
 (eval (argument-of exp)
 env
 (lambda (a) (f a cont)))))))
(else ;; it’s a sequence of stuff!
  (let ((first (first (first-exp exp))
         (rest (rest-exps exp)))
    (eval first
     env
     (lambda (a)
      (eval rest
       (if ((begins-with ’*define*) a)
        (cdr a)
        env)
       cont))))))))

Initial continuation and global environment

(define (initial-continuation x) x)

(define (extend var val env) (cons (cons var val) env))

(define (lookup env var)
  (cond ((null? env) (error "Unbound variable"))
         ((eq? (caar env) var) (cdar env))
         (else (lookup (cdr env) var))))

(define (unop op)
  (lambda (x k) (k (op x))))

(define (binop op)
  (lambda (x k) (k (lambda (y kk) (kk (op x y))))))

6
(define (terop op)
  (lambda (x k)
    (k (lambda (y kk)
        (kk (lambda (z kkk)
            (kkk (op x y z)))))))

(define initial-global-environment
  (extend 'cons (binop cons)
    (extend '-' (binop -)
      (extend '*' (binop *)
        (extend '+' (binop +)
          (extend '=' (binop =)
            (extend '<' (binop <)
              (extend '1+ (unop + '()))))))))

(define (try exp)
  (eval exp
    initial-global-environment
    initial-continuation))
Constructors, destructors, other silly stuff

(define constant? integer?)
(define (variable? v) (not (pair? v)))
(define (begins-with atom)
    (lambda (exp)
        (if (pair? exp) (equal? (car exp) atom) #f)))

(define lambda? (begins-with 'lambda))
(define binder caadr)
(define body caddr)

(define define? (begins-with 'define))
(define define-of car)
(define (rest-of exp)
    (if (null? (cddr exp))
        (cdadr exp)
        (cadr exp)))
(define def-name caadr)
(define (def-body exp)
    (if (null? (cdddr exp))
        (caddr exp)
        (cddr exp)))

(define if? (begins-with 'if))
(define predicate caadr)
(define else-part cadddr)
(define then-part caddr)

(define sequence? (begins-with 'begin))
(define first-exp car)
(define (rest-exps exp)
    (let ((r (cdr exp)))
        (if (pair? r)
            (if (null? (cdr r)) (car r) r)
            r)))

(define enter? (begins-with 'enter))
(define exit? (begins-with 'exit))
(define enter-body caadr)
(define exit-body caddr)

(define (application? exp)
    (if (pair? exp)
        (not ((begins-with 'define) (car exp)))
        #f))

(define function-of car)
(define argument-of caddr)