5.5.4 Interpreter with continuations

We have experienced the power of `call/cc`: it allows us to escape the expression evaluation structure of the language. It gives us the ability to implement "hairy control structures," such as backtracking with `amb` and coroutines. Let's now look at how `call/cc` can implemented in the language.

But we have almost already done this. We implemented an `amb` in section 5.4.2 by changing the form of the execution procedures to continuation-passing style, with success and failure continuations. The transformation of the execution procedures to support `call/cc` is similar but easier: we need only one continuation procedure for each execution procedure, and `call/cc` is implemented by just providing that continuation to the interpreted program!

The general pattern of an execution procedure will be:

```
(lambda (environment continue)
  ;; continue = (lambda (value) ...)
  ;; "return" a value by (continue value)
  )
```

So we start with a new version of `eval`:[32]

```
(define (c:eval expression environment continue)
  ((analyze expression) environment continue))

(define (analyze expression)
  (make-executor (c:analyze expression)))

(define (default-analyze expression)
  (cond ((application? expression)
         (analyze-application expression))
        (else (error "Unknown expression type" expression))))

(define c:analyze
  (simple-generic-procedure 'c:analyze 1 default-analyze))
```

So we see that the result of `(analyze expression)` is an execution procedure that takes an environment and a continuation procedure.

We transform `analyze-application` in the same way:

[32] In this evaluator we use the `c:` prefix to distinguish analogous procedures, as explained in footnote 19 on page 260.

```
(define (analyze-application expression)
  (let ((operator-exec (analyze (operator expression)))
        (operand-execs (map analyze (operands expression))))
    (lambda (environment continue)
      (c:execute-strict operator-exec environment
         (lambda (proc)
           (c:apply proc
                    operand-execs
                    environment
                    continue))))))
```

where we have factored out `execute-strict`, as in our implementation of codeamb, because it is shared by other parts of the code:

```
(define (c:execute-strict executor env continue)
  (executor env
            (lambda (value)
              (c:advance value continue))))
```

All the simple expressions are handled as before, but returning by calling the provided continuation:

```
(define (analyze-self-evaluating expression)
  (lambda (environment continue)
    (continue expression)))

(define (analyze-variable expression)
  (lambda (environment continue)
    (continue
     (lookup-variable-value expression environment))))

(define (analyze-quoted expression)
  (let ((qval (text-of-quotation expression)))
    (lambda (environment continue)
      (continue qval))))
```

In this interpreter the analysis of `lambda` expressions is a bit more sophisticated. We separated the procedures with simple Scheme-like parameter lists from more general procedures with declarations, like `lazy` on the parameters. This simplifies the code for `apply`, allowing us to break it into smaller pieces

```
(define (analyze-lambda expression)
  (let ((vars (lambda-parameters expression))
        (body-exec (analyze (lambda-body expression))))
    (if (simple-parameter-list? vars)
        (lambda (environment continue)
          (continue
           (make-simple-compound-procedure vars
                                           body-exec
                                           environment)))
        (lambda (environment continue)
          (continue
           (make-complex-compound-procedure vars
                                            body-exec
                                            environment))))))
```

We distinguish simple `lambda` expressions by a simple test:

```
(define (simple-parameter-list? vars)
  (or (null? vars)
      (symbol? vars)
      (and (pair? vars)
           (symbol? (car vars))
           (simple-parameter-list? (cdr vars)))))
```

And the handler for conditionals is as in the interpreter for `amb`, but with only one continuation.

```
(define (analyze-if expression)
  (let ((predicate-exec (analyze (if-predicate expression)))
        (consequent-exec (analyze (if-consequent expression)))
        (alternative-exec (analyze (if-alternative expression))))
    (lambda (environment continue)
      (define (decide predicate-value continue)
        (if predicate-value
            (consequent-exec environment continue)
            (alternative-exec environment continue)))
      (c:execute-strict predicate-exec environment
        (lambda (pval)
          (decide pval continue))))))
```

So the pattern is clear, and there is no reason to go further into the details, except for `call/cc`. The way this works is that `call/cc` is the name of a unique object that can be distinguished:

```
(define call/cc (list 'call/cc-tag))

(define (call/cc? p) (eq? p call/cc))
```

This object is treated as a special strict primitive procedure:

```
(define (c:apply-strict procedure args continue)
  (cond ((strict-primitive-procedure? procedure)
         (continue (apply-primitive-procedure procedure args)))
        ((call/cc? procedure)
         (c:deliver-continuation (car args) continue))
        ((simple-compound-procedure? procedure)
         (c:compound-apply procedure args continue))
        (else (error "Bad strict procedure" procedure args)
              'to-retain-stack)))
```

where the application of `call/cc` to a receiver procedure applies the receiver procedure to the continuation, as its argument.

```
(define (c:deliver-continuation receiver continue)
  (c:apply-strict receiver
                  (list continue)
                  continue))
```

Isn't that simple?!

5.6 Power and responsibility

In this chapter we have seen that we have great power from the Church-Turing universality of computation. We can never complain: "I cannot express this in the language I must use." If we know the tricks of interpretation and compilation we can always escape from the confines of any language because it is always possible to build an appropriate domain-specific language for the problem at hand. The exposition here uses Scheme as the underlying language and builds powerful Lisp-based languages on top of Scheme. The reason we use Lisp syntax here is because it greatly simplifies the exposition of these ideas. (See exercise 5.7 on infix notations. If we had to do this in a language with a complicated syntax the exposition would be many times longer and more tedious.) But the power of interpretation is available in any Turing-universal language.

It is important for future flexibility that the languages we build be simple and general. They must have very few mechanisms: primitives, means of combination, and means of abstraction. We want to be able to extend them as needed and to be able to mix and match the parts of programs. And, most important, when