The scheme environment: continuations

Article *in* ACM SIGPLAN Lisp Pointers · June 1987

DOI: 10.1145/1317193.1317197

1 author:

William D. Clinger
Northeastern University
58 PUBLICATIONS **2,114** CITATIONS

SEE PROFILE

The Scheme Environment: Continuations

William Clinger
Tektronix, Inc.

Tail recursion, the topic of my previous article, is a fairly old concept, much older than Scheme. For example, I have heard that an early Algol 60 compiler from Burroughs compiled tail recursion properly. Tail recursion was popularized, however, through a sequence of papers by Guy Steele that led up to Rabbit, the Scheme compiler contained in his master's thesis [6, 7].

Continuations are also a fairly old concept. Their importance became evident in the pioneering work of Christopher Strachey, Peter Landin, John Reynolds, and others who during the late 1960's were using lambda calculus as a formal tool to describe the semantics of Algol 60 and other programming languages [4, 5]. Carl Hewitt borrowed the concept from this work on formal semantics, giving it prominence in his message-passing model of computation known as actors. Continuations were carried over into Scheme, which Guy Steele and Gerry Sussman invented as a concrete implementation of actors.

Continuations are an advanced programming concept, not because they're hard to understand but because we can understand and reason about most programs without thinking about continuations. Continuations become important only when we deal with exceptions and non-trivial control structures.

The model of computation we ordinarily use is a *direct model*, in which procedures return and expressions have values (and, perhaps, side effects to the store). The direct model occasionally breaks down, as when a division by zero causes an escape from the context of the division to another part of the program, or when a Common Lisp program uses `go` or `return-from` or `throw` to transfer control, or when a subtask yields control to another subtask. We are then left waving our hands if we try to explain things in terms of the direct model, because procedures aren't returning and expressions are doing weird things.

We therefore switch to the *continuation model* of computation, which hypothesizes that every expression is evaluated with respect to an implicit default continuation representing the program context that awaits the result of the expression. The continuation represents a future course for the program. It is oracular in the sense that if you tell the continuation κ what the expression evaluates to, and you also tell it the values of all variables and the contents of all the accessible data structures, then κ will tell you the eventual result of the entire program. But there's nothing mysterious about this oracle, since it works simply by running the rest of the program.

In procedural languages like Scheme, a continuation is passed as a tacit argument to every procedure, and indeed to every expression. In the direct model we speak of a procedure returning or an expression yielding a result, but in the continuation model we speak of the procedure or expression sending its result to its implicit continuation.

The reason the continuation model is useful is that a procedure or expression can ignore its implicit continuation and use some other continuation instead. For example, the `reset` procedure might want to ignore its implicit continuation and pass control to a continuation that implements a read/eval/print loop. As another example, the expression `(/ 3 0)` might want to ignore its implicit continuation and pass control to an error-handling continuation instead. Better still, it will pass its implicit continuation κ along to the error-handling continuation so the error handler

can have the option of sending "infinity" or some other value to κ.

From the point of view of the continuation model, the direct model is a simplified form of the continuation model in which the implicit continuation is always used. Though the continuation model is more sophisticated and more accurate, we prefer the direct model for its simplicity in the limited but important domain where the two models coincide, much as we prefer the Newtonian model of space and time to special relativity for the limited but important domain of objects whose speed is small compared to that of light.

In Scheme, continuations are very much like procedures of one argument, which accept the result of a subcomputation and then compute the result of the entire program. For example, the continuation for the expression `x` in the expression `(+ x 3)` is like a procedure that will accept the current value of `x`, add three to it, and pass the result on to the continuation for the entire expression `(+ x 3)`.

As mentioned above, procedures take a continuation as an extra, hidden argument. We can see how this works by rewriting a familiar procedure definition like

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (- n 1)))))
```

in *continuation-passing style*. In continuation-passing style we make the implicit continuation argument visible by adding it as an explicit argument, we replace implicit returns by explicit calls to the continuation, and we replace implicit creations of new continuations by explicit creations of new procedures (accomplished, of course, by a lambda expression):

```
(define (fact n k)
  (if (zero? n)
      (k 1)
      (fact (- n 1)
            (lambda (v)
              (k (* n v))))))
```

Instead of returning the value 1 when `n` is zero, we send it explicitly to the continuation. Instead of calling `fact` recursively with a new implicit continuation that will multiply the result by `n` before sending it to the original continuation, we pass that new continuation explicitly. We can test our continuation-passing version of the factorial procedure by giving it an identity function as the outermost continuation:

`(fact 10 (lambda (v) v))` $\Rightarrow$ 3628800

(Of course, the "$\Rightarrow$" notation is itself a relic of the direct model, so we have to be careful here. What we have done is to think about the computation of the factorial using the continuation model, only to revert to the direct model for reporting the result of that computation.)

An interesting thing happened when we rewrote the factorial procedure in continuation-passing style: It became tail-recursive. This is a general phenomenon. If we were completely thorough in our conversion to continuation-passing style, then every procedure call would have become a tail-recursive call. (That isn't true here because we aren't being thorough about calls to procedures like `zero?` and `*`.) Is continuation-passing style a magic way to obtain the benefits of tail-recursion from non-tail-recursive programs?

No, we got rid of the non-tail-recursive call to `fact` only by creating the new continuation

```
(lambda (v) (k (* n v)))
```

Creating a new continuation takes space and time. The only thing we have accomplished through our conversion to continuation-passing style is that we have made the expense of creating a new continuation more explicit.

If we convert a tail-recursive procedure like

```
(define (fact2 n p)
  (if (zero? n)
      p
      (fact2 (- n 1) (* n p))))
```

into continuation-passing style, then the continuation-passing version doesn't have to create any new continuations because it can just use the original continuation:

```
(define (fact2 n p k)
  (if (zero? n)
      (k p)
      (fact2 (- n 1)
             (* n p)
             k)))
```

If we accept that the last argument is really implicit, this is how a properly tail-recursive implementation would compile the original version of `fact2`. An improperly tail-recursive implementation would compile `fact2` as though it were written as

```
(define (fact2 n p k)
  (if (zero? n)
      (k p)
      (fact2 (- n 1)
             (* n p)
             (lambda (v) (k v)))))
```

The only difference between these two is that the continuation for the recursive call is `k` in the properly tail-recursive implementation but is `(lambda (v) (k v))` in the improperly tail-recursive implementation. Since `(lambda (v) (k v))` and `k` are equivalent up to `eq?`-ness (which isn't defined on the continuations these expressions represent), `(lambda (v) (k v))` amounts to a copy of `k`. To use more traditional compiler jargon, the closure implied by the lambda expression `(lambda (v) (k v))` would be implemented as a stack frame, with the (possibly implicit) dynamic link pointing to `k`. The fact that the closure is unnecessary means that the stack frame is unnecessary.

The procedures we have examined so far can be explained perfectly well using the direct model. Here is a straightforward translation into continuation-passing style of a procedure that *can't* be explained using the direct model:

```
; Given a list of numbers representing resistances,
; returns the equivalent resistance resulting from
; the connection of these resistances in parallel.
; The answer is the reciprocal of the sum of the reciprocals,
```

```
; except that if any of the resistances are zero then the
; answer is also zero.

(define (parallel-resistance resistances k0)
  (define (sum-of-reciprocals l k1)
    (cond ((null? l)
           (k1 0))
          ((zero? (car l))
           ; note use of k0 instead of k1
           (k0 0))
          (else (sum-of-reciprocals
                 (cdr l)
                 (lambda (v) (k1 (+ (/ 1 (car l)) v)))))))
  (sum-of-reciprocals resistances
                      (lambda (v) (k0 (/ 1 v)))))
```

The reason the original program can't be explained using the direct model is that there is one place in this program, indicated by the comment, where the default continuation `k1` is ignored and the result is sent to the continuation `k0` instead. The original program looked something like

```
(define (parallel-resistance resistances)
  (define (sum-of-reciprocals l)
    (cond ((null? l) 0)
          ((zero? (car l)) ???)
          (else (+ (/ 1 (car l)) (sum-of-reciprocals (cdr l))))))
  (/ 1 (sum-of-reciprocals resistances)))
```

except that the mystery expression `???` was a call to the continuation originally passed to `parallel-resistance`, and there was some additional machinery used to make that continuation explicit so it could be called by the mystery expression.

Why was that additional machinery necessary? It would not have been necessary if the program had been written in continuation-passing style in the first place, but procedures written in continuation-passing style don't interface well with procedures that aren't written in continuation-passing style. Because continuation-passing style is verbose and awkward, people prefer to read and write programs normally, using implicit continuations.

Programming languages therefore need mechanisms for making the implicit continuation explicit, a feat I will refer to as capturing. Many languages have only crude and limited mechanisms for capturing continuations. For example, Fortran and Pascal use statement labels as GOTO targets, which are the only explicit continuations in those languages. C's `setjmp` and `longjmp` are more useful because you can send results to the continuation captured by a `setjmp`. The main thing wrong with `setjmp` and `longjmp` is that they are not abstract, and of course the continuations that they capture are not first class objects because they have only dynamic extent. Common Lisp has three distinct mechanisms for capturing continuations: `tagbody`, `block`, and `catch`. (The `tagbody` mechanism is also used by `do`, `prog`, et cetera, and the `block` mechanism is used by `do`, `prog`, `defun`, et cetera.) The importance of continuations to Common Lisp is suggested by the fact that Common Lisp devotes three of its eight or so environments entirely to names for continuations captured by these mechanisms. As in C, Common Lisp's continuations have only dynamic extent.

Scheme relies upon a procedure named `call-with-current-continuation` to capture implicit continuations. `Call-with-current-continuation` takes one argument, say f, which is itself a procedure of one argument. It converts its implicit current continuation into an explicit Scheme

procedure k and calls f, passing k as the argument. The procedure k acts just like the continuation from whence it came. If and when k is called with an argument v, it will ignore whatever implicit continuation is in effect at that time and will give v instead to the continuation from which k was created.

For example, the complete definition of the original version of `parallel-resistance` was

```
(define (parallel-resistance resistances)
  (call-with-current-continuation
    (lambda (return)
      (define (sum-of-reciprocals l)
        (cond ((null? l) 0)
              ((zero? (car l)) (return 0))
              (else (+ (/ 1 (car l)) (sum-of-reciprocals (cdr l))))))
      (/ 1 (sum-of-reciprocals resistances)))))
```

The procedure created by `call-with-current-continuation` is a first class object, as are all objects in Scheme. That means it has unlimited extent, can be stored in variables or data structures, and can be called as many times as desired. The following transcript, for example, shows the interactive definition of a procedure `outermost-return` that takes one argument and will abort the computation in progress by returning its argument directly to the interactive read/eval/print loop. The reason this example cannot be written in other dialects of Lisp is that it calls `outermost-return` after the continuation represented by `outermost-return` has already been used, and it calls `outermost-return` more than once.

```
> (define outermost-return)  ; declare a variable to be assigned later
outermost-return
> (call-with-current-continuation
    (lambda (k)
      (set! outermost-return k)))
#!unspecified
> (outermost-return 1)
1
> (list 1 (outermost-return 2) 3)
2
> (letrec ((useless (lambda (n)
                      (if (zero? n)
                          ; escape from the waiting conses
                          (outermost-return "What, me worry?")
                          (cons 'worry (useless (- n 1)))))))
    (useless 1000))
"What, me worry?"
```

Though this example is silly, the technique is broadly useful. It can for example be used to establish checkpoints in large programs that may be resumed multiple times.

As another silly example, here is an extremely perverse definition of a procedure that adds non-negative integers:

```
(define (add x y)
  (letrec ((goto (lambda (k) (k #t)))
           (L1 (lambda () #f))                    ; will be assigned
           (store-in-L1! (lambda (x) (set! L1 x)))
           (call/cc call-with-current-continuation))
    (call/cc store-in-L1!)
    ; L1 will act like a goto target placed here
    (if (not (zero? x))
        (begin (set! x (- x 1))
               (set! y (+ y 1))
               (goto L1)))
    y))
```

As a last toy example of `call-with-current-continuation`, I offer Eugene Kohlbecker's classic puzzle: What does the following program print?

```
(define (mondo-bizarro)
  (let ((k (call-with-current-continuation (lambda (c) c))))
    (write 1)
    (call-with-current-continuation (lambda (c) (k c)))
    (write 2)
    (call-with-current-continuation (lambda (c) (k c)))
    (write 3)))
```

In Common Lisp, the `unwind-protect` special form provides a way to guarantee that user-specified code will be executed after an expression is evaluated. Not only does it set up a continuation for the expression that will execute the user-specified code, but it arranges for the user-specified code to be executed whenever the expression "throws out" of the continuation by using an explicit continuation captured by `go`, `block`, or `catch`.

A similar feature is needed occasionally in Scheme. The `unwind-protect` feature is inadequate for Scheme's first class continuations, however, because it protects against throwing *out* of a continuation but does nothing about throwing *in*, as when a continuation is used more than once. Scheme programmers therefore use a generalization of `unwind-protect` called `dynamic-wind`. The `dynamic-wind` procedure takes three arguments, each of which is a *thunk*, that is, a procedure of no arguments. `Dynamic-wind` usually acts as though it were defined by

```
(define (dynamic-wind before main after)
  (before)
  (let ((v (main)))
    (after)
    v))
```

but the true definition of `dynamic-wind` has the following additional behavior. Notice that the implicit continuation for the call to `main` will call `after` before returning the result of the call to `main`. If the call to `main` causes a call to an explicit continuation that would bypass that implicit continuation, however, then `after` will be called before the call to the explicit continuation is completed. Similarly, if the call to `main` causes a continuation to be captured and stored in a variable or data structure, and that explicit continuation is called after `main` has returned, then `before` will be called before the call to that explicit continuation is completed. To extend an example from *Common Lisp: the Language*,

```
(dynamic-wind (lambda () (start-motor))
              (lambda () (drill-hole))
              (lambda () (stop-motor)))
```

not only ensures that the motor is turned off after the hole is drilled, but it also ensures that the motor is turned on again before any suspended drilling is resumed.

The `dynamic-wind` procedure is not a standard Scheme procedure, but a simple and reasonably portable implementation of it appears in [1]. The portable implementation works by redefining `call-with-current-continuation`, which is the only standard mechanism for capturing continuations. Specific implementations of Scheme may have other mechanisms for capturing continuations that would have to be redefined as well.

In a future article I will show how continuations can be used to implement dynamic variables in Scheme. One reason that dynamically bound variables are not a standard part of Scheme is that there are several distinct semantics that one might want for dynamic variables, and it isn't at all clear which if any should be standard. We'll consider only a few of the possible semantics.

Continuations can also be used to implement a number of advanced control structures including agendas and coroutines [2]. With the addition of timed interrupts and a synchronization mechanism, continuations can be used to implement a simple multi-tasking facility. The continuation model is of course important quite apart from Scheme. For example, the semantics of Prolog can be understood quite well by imagining that each expression has not one, but *two* implicit continuations: a failure continuation and a success continuation [3].

* * *

If you have trouble getting onto the Scheme mailing list using the old electronic mail address I gave last month, try Scheme-Request@mc.lcs.mit.edu instead. And a guru has recommended that people who've had trouble sending mail to me should try willc%tekchips.tek.com@relay.cs.net.

[1] Daniel P Friedman and Christopher T Haynes. Constraining control. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 245–254. ACM, January 1985.

[2] Christopher T Haynes, Daniel P Friedman, and Mitchell Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298. ACM, August 1984.

[3] Christopher T Haynes: Logic continuations. In *Proceedings of the Third International Conference on Logic Programming*, pages 671–685. Springer-Verlag, July 1986.

[4] Peter Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8, 2, pages 89–101. February 1965.

[5] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[6] Guy Lewis Steele Jr. Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or lambda: the ultimate GOTO. In *ACM Conference Proceedings*, pages 153-162. ACM, 1977.

[7] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.