Continuations and Coroutines

by

C. T. Haynes, D. P. Friedman and M. Wand

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 158

Continuations and Coroutines

June, 1984

by

C. T. Haynes, D. P. Friedman and M. Wand

This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567 and MCS 83-03325.

Continuations and Coroutines*

Christopher T. Haynes

Daniel P. Friedman

Mitchell Wand

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, Indiana 47405 USA

Abstract

The power of first class continuations is demonstrated by implementing a variety of coroutine mechanisms using only continuations and functional abstraction. The importance of general abstraction mechanisms such as continuations is discussed.

1 Introduction

A variety of coroutine mechanisms have been proposed, some so complicated that there has been significant confusion over their semantics [1]. Yet most languages provide no coroutine mechanism, and do not provide sufficient power to allow the user to define one. In this paper we demonstrate that a wide variety of coroutine mechanisms may easily be defined by the user given a single control abstraction, called a *continuation*. The power of continuations stems from their first class nature: they may be passed to and returned from functions, be stored in data structures, and have indefinite extent.

---

* A preliminary version of this paper was presented at the *1984 ACM Symposium on LISP and Functional Programming*. This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567 and MCS 83-03325.

In Scheme 84 [2] the call-with-current-continuation expression of the form `(call/cc` *Exp*`)` applies the function which results from evaluating *Exp* to the continuation of the entire `call/cc` expression. This continuation is a first class functional object that, when invoked, returns the value of its single parameter as the value of the entire `call/cc` expression. The control and environment information associated with continuations is recorded in storage that is dynamically allocated. It is reclaimed only when all references to the continuation have been abandoned. This allows complete management of the control behavior of the computation, including arbitrary backtracking and other forms of context switching such as interrupt driven multiprocessing [3]. (In GEDANKEN [4], labels are also first class control objects, but are less general than continuations; they refer only to statements, rather than to arbitrary points within an expression. Reynolds also notes that GEDANKEN labels may be used to implement coroutines.)

In the next section we provide an overview of Scheme 84, a Scheme dialect which provides first class functional and continuation objects. We then demonstrate how a basic coroutine mechanism may be defined in Scheme 84 using continuations. This basic mechanism is then elaborated to provide additional features, such as a simple interface between a group of coroutines and a stream. Finally, we conclude with a discussion of the importance of powerful abstractions, such as first class continuations, for language extensibility.

2 An Overview of Scheme 84

Scheme was designed and first implemented at MIT in 1975 by Gerald Jay Sussman and Guy Lewis Steele, Jr. [5,6] as part of an effort to understand the actor model of computation [7]. Scheme may be described as a dialect of Lisp that is applicative order, lexically scoped, and properly tail-recursive. Most importantly, Scheme—unlike all other Lisp dialects—treats functions and continuations as first class objects.

A subset of Scheme 84 [2] is composed of the following syntactic constructs:

```
⟨expression⟩ ::=
     ⟨constant⟩
   | ⟨identifier⟩
   | ⟨syntactic extension⟩
   | (lambda ( {⟨identifier⟩} ) {⟨expression⟩} )
   | (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩ )
   | (set! ⟨identifier⟩ ⟨expression⟩ )
   | (call/cc ⟨expression⟩ )
   | ⟨application⟩

⟨syntactic extension⟩ ::= (⟨keyword⟩ {⟨expression⟩} )

⟨application⟩ ::= ( ⟨expression⟩ {⟨expression⟩} )
```

`lambda` is the sole binding operator and evaluates its expressions sequentially, returning the value of the last. `set!` side-effects an existing identifier binding or initializes a global identifier. `call/cc` is described below. An application evaluates its expressions (in an unspecified order) and applies the value of the first expression to a list of arguments formed from the values of the remaining expressions.

Scheme 84 provides a syntactic preprocessor that examines the first object in each expression. If the object is a syntactic extension (macro) keyword, the procedure associated with the indicated syntactic extension is invoked on the expression, and the expression is replaced by the resulting transformed expression. If the object is not a keyword or core expression identifier (`lambda`, `if`, *etc.*), then it is assumed that the expression is a normal function application.

We use a few syntactic extensions. `let` makes local identifier bindings. `define` changes an existing identifier binding or initializes a global identifier.

$$\texttt{(rec } I\ E\texttt{)} \equiv \texttt{(let ([}I\ \texttt{'*]) (set! } I\ E\texttt{))}$$

evaluates E in an environment which binds I to the value of E itself.* Evaluating E should not result in dereferencing I. This presents no problem in the usual case when E evaluates to a closure.

$$\texttt{(case } Tag\ \texttt{[}A_1\ \ E_1\texttt{]}\ \ldots\ \texttt{[}A_n\ \ E_n\texttt{])}$$

evaluates *Tag* and then returns the value of the first E_i such that A_i matches the value of *Tag*.†

`call/cc` evaluates its argument and applies it to the current continuation represented as a functional object of one argument.‡ In order to specify the current continuation at any point in a computation, a *continuation semantics* is necessary. We provide such a semantics with the meta-circular Scheme 84 interpreter in Figure 1. The value passed to the continuation is the result of the computation up to the point where it is invoked.

In a continuation semantics, every recursive procedure receives a continuation parameter. Rather than simply return, the procedure either passes its result directly to this

---

* ≡ indicates that an expression of the form on the left is transformed into one of the form on the right, and brackets are interchangeable with parentheses.

† Some readers may be put off by the number of parentheses in Scheme programs. However, we feel these parentheses are justified for several reasons. In the first place, there are more advantages of such minimal syntax than is generally realized. A simple, unambiguous syntax aids comprehension and provides the basis for a truly extensible language. (Users may easily define their own syntactic extensions, or macros, thereby extending the syntax of the language to meet their needs.) Secondly, the difficulties associated with heavily parenthesized expressions are greatly reduced with some practice and intelligent tools, such as editors and pretty-printers [8]. Finally, in this paper we are concerned with semantics and do not wish to be burdened by elaborate syntax. The obdurate reader may imagine these semantics expressed in his or her favorite syntax.

‡ Using this primitive we can define `catch`, a version of Landin's J operator [9,10,6].

```
(define meaning
   (lambda (e r k) ; e = expression, r = environment, k = continuation
      (case (type-of-expression e)
         [constant (k e)]
         [identifier (k (R-lookup e r))]
         [lambda (k (lambda (actuals k)
                       (evaluate-all (body-pts e)
                          (extend-env r (formals-pt e) actuals)
                          k)))]
         [if (meaning (test-pt e) r
                (lambda (v) (if v (meaning (then-pt e) r k)
                                  (meaning (else-pt e) r k))))]
         [set! (meaning (val-pt e) r
                  (lambda (v) (k (store! (L-lookup (id-pt e) r) v))))]
         [call/cc (meaning (fn-pt e) r
                     (lambda (f)
                        (f
                           (list (lambda (actuals dummy) (k (car actuals))))
                           k)))]
         [application (meaning-of-all (comb-parts e) r
                         (lambda (vals) ((car vals) (cdr vals) k)))])))

(define evaluate-all
   (lambda (exp-list r k)
      (meaning (car exp-list) r
         (if (null? (cdr exp-list))
             k
             (lambda (v) (evaluate-all (cdr exp-list) r k))))))

(define meaning-of-all
   (lambda (exp-list r k)
      (meaning (car exp-list) r
         (lambda (v) (if (null? (cdr exp-list))
                         (k (cons v nil))
                         (meaning-of-all (cdr exp-list) r
                            (lambda (vr) (k (cons v vr)))))))))
```

Figure 1. Meta-circular interpreter for a subset of Scheme 84

continuation, or passes the continuation (possibly embellished) to some other procedure.

For example, consider `meaning` (Figure 1). The constant, identifier, and lambda expressions pass their values directly to the continuation `k`. In the `if` case, `meaning` is recursively invoked on the `test-pt` (predicate) of the expression in the current environment, and with a new continuation, denoted by a lambda expression, which will be passed the value of the `test-pt`. Depending on this value, `meaning` is invoked on the `then-pt` or `else-pt` with the original continuation `k` of the `if` expression.

3 Implementation of Coroutines

Heretofore, coroutine languages have generally been statement oriented, and the resume statement which transfers control from one coroutine to another could not return a value. Thus coroutines have had to communicate information through shared free variables. Because Scheme is expression oriented, we must associate some value with the resume expression. What better candidate for this value than a value passed by the resuming coroutine? Though shared variables may still be necessary for some types of communication, such resume values suffice in most cases for inter-coroutine communication.

Thus, in our Scheme implementation of coroutines, `resume` will be a function of two arguments: the coroutine to be resumed, and the value to be passed to the resumed coroutine. This value becomes the value of the resume expression by which the receiving coroutine last relinquished control.

We implement a coroutine as a procedure with some local state. Each coroutine must have its own private resume function, because the resume function records the current state of the coroutine's computation (its continuation!) in a variable local to the coroutine. This variable is called LCS, for Local Control State. Since the LCS variable is part of the implementation of our coroutine facility, and not part of the programmer's abstraction of coroutines, the LCS variable should not be in the scope of the programmer's coroutine code. This goal will be achieved using the scoping mechanisms of Scheme.

We implement the coroutine abstraction with a function `make-coroutine` that takes as its argument another function, which contains the programmer's coroutine code. `make-coroutine` then passes to this function a resume function. Thus `make-coroutine` will be of type

$$(\textit{resume-function} \rightarrow x) \rightarrow y$$

for some types x and y that we have not yet elaborated.

The result of `make-coroutine` is a procedural object with a private variable, LCS, which holds the current state of its computation (obtained with `call/cc`). Each time the coroutine object is invoked, it receives a value, which is sent to the continuation LCS. Thus the interface returned by make-coroutine should look like `(lambda (v) (LCS v))`, closed in an environment in which LCS is properly bound. This function receives a value `v` and sends it to the current (application time) value of LCS, as desired. (Note that the interface could not be simply LCS, which would always use the closure-time value of LCS.) Since `make-coroutine` returns a function of one argument, its type is

$$(\textit{resume-function} \rightarrow x) \rightarrow (\textit{value} \rightarrow z).$$

Each time a coroutine is resumed (including the first time) it must also receive a value, so we further require that the argument to `make-coroutine` be a function which accepts a resume function and returns a function of one argument, a *receiver* that accepts the initial value with which the coroutine is to be resumed. The type of `make-coroutine` is now

$$(\textit{resume-function} \rightarrow (\textit{value} \rightarrow z)) \rightarrow (\textit{value} \rightarrow z).$$

Here z represents the type of value returned by the coroutine. In the simplest view of coroutines, they just resume one another and never return a value; so at this time the type of z is immaterial. Later we will give a meaningful semantics to values returned by coroutines.

Thus the following expression returns a new coroutine procedure that executes *Body*, with `resume` bound to the local resuming function and `init` bound to the value with which the coroutine is initially resumed or invoked.

```
(make-coroutine
   (lambda (resume)
      (lambda (init)  Body)))
```

Our implementation starts as:

```
(define make-coroutine
   (lambda (f)
      (call/cc
         (lambda (maker)
            (let ([LCS '*])
               (let ([resume
                        (lambda (dest val)
                           (call/cc
                              (lambda (self)
                                 (set! LCS self)
                                 (dest val))))])
                  (let ([receiver (f resume)])
                     ???)))))))
```

`resume` grabs the current state of the computation, stores it in LCS, and invokes the destination with the given value, as desired.

The tricky part of this implementation is setting LCS to the correct initial continuation, which should look like `(lambda (v) (receiver v))`. We can create a continuation like this by doing a `call/cc` in the context `(receiver (call/cc (lambda (initk) ...)))`. Having created this continuation, we can store it in `LCS` and then exit from `make-coroutine` by invoking the continuation `maker` (which we were clever enough to grab when we entered `make-coroutine`). The value which we should return to the maker is just `(lambda (v) (LCS v))`, as we decided before. Now we have:

```
(define make-coroutine
   (lambda (f)
      (call/cc
         (lambda (maker)
            (let ([LCS '*])
               (let ([resume
                        (lambda (dest val)
                           (call/cc
                              (lambda (self)
                                 (set! LCS self)
                                 (dest val))))])
                  (let ([receiver (f resume)])
                     (receiver
                        (call/cc
                           (lambda (initk)
                              (set! LCS initk)
                              (maker
                                 (lambda (v)
                                    (LCS v)))))))))))))
```

The last `call/cc` expression, however, is just `(resume maker (lambda (v) (LCS v)))`! Also, receiver is used only once, so we can substitute `(f resume)` for its use. The last `let` expression may then be replaced by

`((f resume) (resume maker (lambda (v) (LCS v))))`.

The currying of `f`, which has so far been valuable in the logical development of `make-coroutine`, is superfluous (except by those who like currying for its own sake, or who have other reasons for its use). Uncurrying `f`, the type of `make-coroutine` becomes

$$(\mathit{resume\text{-}function} \times \mathit{value} \rightarrow z) \rightarrow (\mathit{value} \rightarrow z)$$

and we have our final version:

```
(define make-coroutine
  (lambda (f)
    (call/cc
      (lambda (maker)
        (let ([LCS '*])
          (let ([resume
                  (lambda (dest val)
                    (call/cc
                      (lambda (k)
                        (set! LCS k)
                        (dest val))))])
            (f resume
              (resume maker
                (lambda (v) (LCS v))))
            (error 'coroutine-fell-off-end)))))))
```

As noted above, coroutines generally call one another in turn, but what if one just returns a value? (Remember, in our implementation they are procedures, and any procedure can return a value.) The `(error 'coroutine-fell-off-end)` expression above ensures that something predictable happens in this case.

If textual abstraction is preferred to functional abstraction, and one is always willing to call the resume-function `resume` and the initial value `init-value`, then one may use the syntactic extension:

```
(coroutine E) ≡
  (make-coroutine
    (lambda (resume init-value) E))
```

4 Extensions of the Coroutine Mechanism

Dahl and Hoare [11] describe an extension to the basic coroutine mechanism described above, which they incorrectly perceive as that of Simula [1]. In addition to the continuation LCS of the last point at which the coroutine relinquished control, each coroutine also records its *caller continuation*, CC. A coroutine may be invoked by either a resume or a *call* operation. In the latter case, the continuation of the caller is recorded in the CC of the called coroutine. When one coroutine resumes another, the CC of the resumer becomes the CC of the resumed coroutine. A coroutine can then invoke its caller continuation by an explicit *detach* operation, or by simply returning a value.

To implement Dahl-Hoare style coroutines, we represent coroutines not as functions of one argument (a value to be passed to the LCS) but as a function of two arguments, a value and a caller continuation. The caller continuation is recorded in the CC variable of the resumed or called coroutine, and the value is then passed to the LCS as before. The resume procedure now passes both the resumer's caller-continuation and the resumer value to the destination coroutine. The new detach procedure also grabs the current continuation and saves it in LCS, and then simply invokes CC with the given value. The functional argument `f` of `make-coroutine` now receives this detach-function, as well as the resume-function and initial value. See Figure 2.

One can enter a group of coroutines in this discipline by using the function `call`, which receives the destination coroutine and a value to pass to it, grabs the current continuation, and passes this continuation and the given value to the called coroutine.

```
(define call
   (lambda (dest val)
      (call/cc
         (lambda (k) (dest k val)))))
```

In some situations, it may be necessary to pass control to, and return control from,

```
(define make-coroutine
   (lambda (f)
      (call/cc
         (lambda (maker)
            (let ([LCS '*] [CC '*])
               (let ([resume (lambda (dest val)
                                (call/cc
                                   (lambda (k)
                                      (set! LCS k)
                                      (dest CC val))))]
                     [detach (lambda (val)
                                (call/cc
                                   (lambda (k)
                                      (set! LCS k)
                                      (CC val))))])
                  (detach ((f resume detach)
                           (call/cc
                              (lambda (k)
                                 (set! LCS k)
                                 (maker (lambda (cont val)
                                           (set! CC cont)
                                           (LCS val)))))))))))))
```

Figure 2. Dahl-Hoare Style Coroutine

a group of coroutines without need for the generality of the Dahl-Hoare style facility just discussed. In our implementation, the fact that coroutines are invoked not by a primitive resume operation, but by application to a standard functional object (the resume-function), allows us to easily achieve other forms of control. For example, by creating a coroutine that simply exports its resume-function, we can create an interface between a group of coroutines and other parts of a program. To accomplish this, we define a coroutine `interface` that passes its resume-function to its initial value:

```
(define interface
   (coroutine (init-value resume)))
```

(Recall that `coroutine` is a syntactic extension that binds `init-value` and `resume`.) By passing this coroutine a continuation as its initial value, we can obtain the coroutine's resume-function at any point in a program.

```
(let ([interface-resume (call/cc interface)])
  ...)
```

By invoking this resume-function with another coroutine and a value, the indicated coroutine will be resumed with the indicated value. The resumed coroutine may then resume other coroutines, with control being passed around within some coroutine group until it is desired that control return to the point at which the group was entered. It is then only necessary for one of the coroutines to resume the interface coroutine via `(resume interface` *Value*`)`. This causes control to return to the caller of the interface-resume function, with some indicated resume value. (Of course a group of coroutines could have several interface coroutines of this sort if required.)

As an example of the use of this mechanism, consider the problem of generating a stream of values, where the consecutive values are produced by a group of coroutines. We represent a stream as a pair `(cons` *Value Thunk*`)`, where *Value* is the first element of the stream and *Thunk* is a function of no arguments which, when invoked, will return a similar representation of the rest of the stream. Assuming that `corout` is the name of the coroutine group member that is to be resumed first in order to initiate computation of the next stream element (and assuming that the value with which `corout` is resumed is irrelevant) the next stream may be obtained by

```
(cons (interface-resume corout nil) thunk)
```

where `thunk` is a function of no arguments which, when invoked, will return the representation of the rest of the stream. After `corout` has been resumed by `interface-coroutine`, any member of the coroutine group can then return v as the next stream element by executing `(resume interface` v`)`. Thus a coroutine-generated stream may be constructed by

```
(let ([interface-resume (call/cc interface)])
   ((rec thunk
       (lambda ()
          (cons (interface-resume corout nil)
               thunk)))))
```

The stream interface example is of particular importance because in many situations where coroutines have been employed, streams are more appropriate. Yet coroutines are clearly more general than streams. When the transfer of control among coroutines is linear, that is

$$A \Rightarrow B \Rightarrow \ldots \Rightarrow Z \Rightarrow \ldots \Rightarrow B \Rightarrow A$$

(read ⇒ as "resumes"), information is passed in only one direction, and each coroutine divides easily into segments between resumes, then a much more transparent solution may generally be obtained by using streams rather than coroutines. Conway's problem [12], the classic example for coroutines, is of this form. The relationship between coroutines and streams is analogous to that between imperative and applicative programming. If a problem can without undue difficulty be expressed in a purely side-effect-free form, such a solution is usually more transparent and it is well worth the effort to obtain. However, if state transitions are inherent to the problem being solved, clarity is often lost by attempting to force a solution that is side-effect-free.

Thus we would like to be able to use streams wherever appropriate, but have the more powerful but less disciplined coroutine facility available when needed. If part of a problem yields to a stream implementation, but some segment of the problem is best solved with coroutines, with a coroutine-stream interface at our disposal we can combine both techniques in the same program, using each as appropriate.

5 Conclusion

We have demonstrated that the first class functional and control objects of Scheme 84 allow us to implement a coroutine facility, rather than having to incorporate a coroutine mechanism in the core language. We also have the freedom to extend the basic coroutine mechanisms in a variety of ways and to interface coroutines with other language features in ways suitable to a given problem or class of problems.

We believe that a primary responsiblity of language designers is to provide a flexible basis for the creation of abstractions suitable for various classes of problems. The presence of first class continuations in Scheme 84 provides such a basis for the creation of control abstractions such as coroutines. If a mechanism as complicated as coroutines is included in a language design, it will be more complicated than is required for most applications, yet still lack features desirable for some applications. For instance, many coroutine applications do not need the complexity of a detach mechanism, and it is unlikely that even a very elaborate coroutine mechanism will provide as clean a stream interface as the one developed here. By using a clean base language with powerful and orthogonal reflection mechanisms (such as `call/cc`), the programmer is able to create control structures far better tuned to the problem at hand.

References

1. Marlin, C. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, Lecture Notes in Computer Science 95, Springer-Verlag, 1980.

2. Friedman, D.P., Haynes, C.T., Kohlbecker, E., and Wand, M. "The Scheme 84 Reference Manual" Indiana University Computer Science Department Technical Report No. 153 (March, 1984).

3. Wand, M. "Continuation-based multiprocessing," *Conf. Record of the 1980 Lisp Conference*, August 1980, pages 19–28.

4. Reynolds, J. "GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept," *CACM* **13**, May 1970, pages 308–319.

5. Steele, G., and Sussman, G., "The revised report on Scheme: a dialect of Lisp", MIT Artificial Intelligence Memo 452, January 1978.

6. Sussman, G., and Steele, G. "Scheme: an interpreter for extended lambda calculus", MIT Artificial Intelligence Memo 349, December 1975.

7. Hewitt, C. "Viewing control structures as patterns of passing messages", *Artif. Intell.* **8**, 1977, pages 323–363. Also in Winston and Brown [ed], *Artificial Intelligence: an MIT Perspective*, MIT Press, 1979.

8. Sandewall, E., "Programming in an interactive environment: the "Lisp" experience," *Computing Surveys* **10**, March 1978, pages 35–71.

9. Landin, P. "A correspondence between ALGOL 60 and Church's Lambda Notation", *CACM* **8**, 2–3, February and March 1965, pages 89–101 and 158–165.

10. Reynolds, J. "Definitional interpreters for higher order programming languages", *ACM Conference Proceedings 1972*, pages 717–740.

11. Dahl O.-J., and Hoare, C.A.R. "Hierarchical Program Structures," *Structured Programming*, Dahl, O.-J., Dijkstra, E., and Hoare, C.A.R., Academic Press, 1972, pages 157–220.

12. Conway, M. "Design of a separable transition-diagram compiler," *CACM* **6**, 1963, pages 396–408.