Programming with Continuations

by

Daniel P. Friedman, Christopher T. Haynes & Eugene Kohlbecker

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 151

Programming with Continuations

by

Daniel P. Friedman, Christopher T. Haynes & Eugene Kohlbecker

November, 1983

Appears in *Program Transformation and Programming Environments,* (P. Pepper, ed.), Springer-Verlag, Berlin Heidelberg (1984), pp. 263–274.

Research reported herein was supported in part by the National Science Foundation, under grants numbered MCS 79–04183 and MCS 83–04567.

Programming with Continuations

Daniel P. Friedman

Christopher T. Haynes

Eugene Kohlbecker

Computer Science Department
Indiana University
Bloomington, Indiana 47405 USA

First Class Continuations are described, and a means for their explicit access is defined by a meta-circular interpreter. We pose and solve the "Devils and Angels Problem" that utilizes the indefinite extent of continuations. Some ramifications of continuations with respect to language implementation and non-blind backtracking are presented.

1 Introduction

Progress in programming language design has often been achieved by making an abstraction a "first class object", one that can be passed to and returned from procedures and entered in data structures. For example, the importance of functional parameters has long been recognized, though it is only more recently that actor semantics [2] and object oriented programming have demonstrated the power of first class *functional* objects. This paper illustrates, with a novel example, the power of first class *control* objects, called *continuations.*

Control objects, such as labels in Algol 60, are not new to programming languages. However, such objects are not first class—though they may be passed to procedures, they may not be returned or entered in data structures. For example, when an Algol label is invoked it is impossible for the computation to return to the state it had when the goto was executed. This is a consequence of the stack allocation of both environment (parameters and local storage) and control (return

Research reported herein was supported in part by the National Science Foundation under grants MCS 79-04183 and MCS 83-04567.

address) information. When a label is invoked the stack is popped and the context of the computation subsequent to the transfer of control is lost.

Several Lisp dialects, including Common Lisp [7] and T [4], provide mechanisms similar to labels in an expression (rather than statement) oriented context. These mechanisms were motivated by the **catch** and **throw** expressions (for example, see [11]). The **catch** statement binds the current continuation, or control state, to an identifier and then evaluates an expression contained in the catch statement. Within that expression it is possible at any point to "throw" an arbitrary value to the continuation associated with the identifier. This value is then returned immediately as the value of the entire catch expression, from which point evaluation proceeds in the environment saved with the continuation. However, as with the labels discussed above, once the catch expression has been exited, it is no longer possible to reinvoke its continuation. This is again a consequence of the stack allocation of the continuation (though the environment is now being dynamically allocated on a heap). The principal uses of this mechanism are error exits and "blind" backtracking. Non-blind backtracking, in which it may be necessary to resume a computation at the point where a continuation is invoked, is not possible in these languages.†

In Scheme 84 [1] the call-with-current-continuation expression described by the form `(call/cc (lambda (k) `*Exp*`))` causes the expression *Exp* to be evaluated in an environment where `k` is bound to the continuation of the entire `call/cc` expression. The continuation bound to `k` 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. Such continuations may be passed to functional objects (even other continuations!), returned from procedures, and preserved indefinitely in data structures. 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 manage-

---

†See Conniver [10] for a description of an artificial intelligence language that addresses problem solving with non-blind backtracking.

ment of the control behavior of the computation that includes not only arbitrary backtracking, but also other forms of context switching such as the resumption of coroutines [1] and interrupt driven multiprocessing [12].

In the next section a problem is posed to motivate the power of first class continuations. This is followed by a brief introduction to Scheme 84 and by a meta-circular tail-recursive Scheme 84 interpreter that provides an operational semantics for continuations. A solution to the proposed problem is then presented, and other implications of first class continuations are discussed.

2 The Devils and Angels Problem

Consider a program that includes, in addition to the standard expressions, three types of special purpose expressions: *milestones*, *devils*, and *angels*. The computation described by the program has the goal of finishing despite the existence of devils. A devil sends control back to the last milestone, with the same environment it had at that milestone. The value given to the devil is passed to the continuation commencing at the milestone, as if that value were the result returned by the milestone expression. Presumably this allows the computation to take a different path, possibly avoiding the devil that is lurking somewhere ahead on the previously used path. If another devil, or maybe even the same devil, is subsequently encountered, then control passes back to the penultimate milestone, not to the one just used. In other words, each milestone can be returned to exactly once; a succession of devils pushes the computation back to earlier and earlier states.

An angel sends the computation forward to where it was when it most recently encountered a devil, with the environment it had at that time. The value passed as a parameter to the angel is given to the devil's continuation as if it were the value of the devil. A succession of angels pushes the computation forward to more and more advanced states.

A milestone is a function of one argument that acts as the identity function, as well as recording the current context for later use by devils. If a devil is

encountered before any milestone has been reached, the devil has no effect; if an angel is encountered before any devil, the angel has no effect. To recharge any milestone—to make it again a possible return point for some devil once a devil has caused control to jump back to it—the milestone must be re-evaluated.

The problem we are proposing is to define the three functions `milestone`, `devil`, and `angel` so that they have the described behavior. It is helpful to observe that there is no non-determinism in this problem; the control path is followed by a single process.

A helpful metaphor for these constructs is one of solving a multi-part problem. Frequently mathematics texts present problems with many parts, and the overall structure is something like "assuming the results of parts (a), (b), and (c), prove (d)". Imagine that someone trying to solve the entire problem—do all the parts—has no prior knowledge that (a), (b), and (c) are useful in solving (d). In fact, it may be that result (a) is not really what is needed for (d), but some partial result obtained by solving (a). Our problem solver begins working on (d), thinking it looks easiest. She soon gets stuck and thinks about trying to tackle another part, say (a). But, before beginning work on (a), she records where she was in solving (d); she establishes a milestone. Then, while working on (a), she discovers some partial result that is just what she needed to proceed with (d). So, she resumes work on (d) where she had left off, but with some new information, the discovery from (a). This corresponds to invoking a devil. Later, she decides to work again on (a). If she is finished with (d), she can simply return; if not, she must set up a new milestone and then return. The return to (a) is modeled by the invocation of an angel.

It is tempting at this point to think we are describing coroutines, for it is easy to set up a collection of coroutines for the above example that have behavior superficially similar to continuations. However, on closer examination, it is seen that continuations are far more general than coroutines. Each coroutine has its own unique locus of control that is constrained to remain within its textual bounds. Clearly continuations may be used to model coroutines, for this locus of control (and its associated environment) may at any time be captured as a continuation

and recorded in a data structure associated with the coroutine. But continuations are not in general constrained to such a one-to-one relation between a control object and its possible extent. Furthermore, because continuations are first class objects they may be manipulated as data by the programmer. Thus they are not limited to some particular use envisioned by the language designer. In abstracting the idea of a control object, the designers of Scheme have placed an important tool of the language implementor into the hands of the language user. A language with first class continuations does not need to include coroutines as part of its definition. Those users who want them can implement them, and those who need other sorts of control constructs can implement them as well.

3 An overview of Scheme 84

The problem we present in this paper has no direct solution in traditional programming languages. However, it does have a solution in Scheme, a programming language designed and first implemented at MIT in 1975 by Gerald Jay Sussman and Guy Lewis Steele, Jr. [6,11] as part of an effort to understand the actor model of computation. Scheme may be described as a dialect of Lisp that is applicative order, lexically scoped, and properly tail-recursive; but most importantly Scheme—unlike all other Lisp dialects—treats functions and continuations as first class objects.

A subset of the core of Scheme 84 [1], an extended version of Scheme, is composed of the following syntactic constructs.

```
<expression>::= <constant>
              | <identifier>
              | (lambda ( {<identifier>} ) <expression>)
              | (if <expression><expression><expression>)
              | (change! <identifier><expression>)
              | (call/cc <expression>)
              | (begin <expression> {<expression>} )
              | (comb <expression> {<expression>} )
```

lambda is the sole binding operator of Scheme 84. **change!** side effects an existing identifier binding or initializes a global identifier. **call/cc** is described below. **begin** provides the usual sequential evaluation of its list of expressions and returns the value of the last. **comb** evaluates its expressions (in an unspecified order) and applies the first expression to a list of arguments formed from 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 a normal function application, a *combination*, and the preprocessor inserts the core identifier **comb**.

A few important syntactic extensions follow. ($\equiv$ 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.)

(**let** ([I_1 E_1] ... [I_n E_n]) E) $\equiv$ ((**lambda** (I_1 ... I_n) E) E_1 ... E_n)

(**define** I E) $\equiv$ (**begin** (**change!** I E) (**quote** I))

(**begin0** E_0 E_1 ... E_n) $\equiv$ (**let** ([x E_0]
[y (**lambda** () (**begin** E_1 ... E_n))])
(**begin** (y) x))

let is used to make local identifier bindings. **define** changes an existing identifier binding or initializes a global identifier. It returns the identifier, rather than its new binding, which is returned by **change!**. The last macro **begin0** evaluates its expressions sequentially, as does **begin**, but returns the value of the first expression rather than the value of the last.

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.

---

†Using this primitive we can define **catch**, a version of Landin's J operator [3,5,11].

We provide such a semantics with the meta-circular Scheme 84 interpreter of Figure 1.‡ The syntax of Scheme differs from the lambda calculus of traditional denotational semantics [8,9], but the same semantic techniques are evident. A significant difference between Scheme and the lambda calculus is the applicative order of evaluation in Scheme; arguments are evaluated before application. In a continuation semantics, every recursive procedure receives a continuation parameter, and is obligated to pass any results directly to this continuation, rather than simply return. Continuations, abstractly functions of one argument, represent the remainder of the computation at the point where the procedure is invoked. The value passed to the continuation is the result of the computation up to the point where it is invoked.

For example, consider `meaning` in Figure 1. The constant, identifier, and lambda expressions pass their values directly to the continuation `k`. However, the other cases are more subtle. In the `if` case, `meaning` is recursively invoked on the `test-pt` (predicate) of the expression in the current environment. Its continuation, denoted by a lambda expression, 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.

If the meanings of `lambda` expressions, *closures*, were not first class objects, then the environment could be maintained on a stack. This stack corresponds to the static chain of Algol-like languages. In addition, if continuations were not first class, then the control information could be recorded on the same stack. In this context, the closures that we use to represent continuations in `meaning` would contain the same information as a stack activation record.

4 Solution to the Devils and Angels Problem

Having introduced `call/cc` and having defined the semantics of continuations, we are now in a position to present a concise solution to our problem. First

‡Our interpreter is not compositional in its present form, but can be made so, at some expense in clarity.

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

```
(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 (mk-closure (body-pt e) (formals-pt e) r))]
            [if (meaning (test-pt e) r
                   (lambda (v) (if v (meaning (then-pt e) r k)
                                     (meaning (else-pt e) r k))))]
            [change! (meaning (val-pt e) r
                        (lambda (v)
                           (k (store! (L-lookup (id-pt e) r) v))))]
            [call/cc (meaning (fn-pt e) r
                        (lambda (v)
                           (apply-function v (cons (mk-cont k) nil) k)))]
            [begin (evaluate-all (exp-list-of e) r k)]
            [comb (meaning-of-all (comb-parts e) r
                     (lambda (vals)
                        (apply-function (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)))))))))

(define apply-function
   (lambda (fnrep args k)
      (case (type-of-fn fnrep)
            [primitive (k (apply-primitive fnrep args))]
            [closure (meaning (closure-exp fnrep)
                              (extend-env (closure-env fnrep)
                                          (closure-formals fnrep) args)
                              k)]
            [continuation ((cont-pt fnrep) (car args))])))
```

we define the stack operations **push** and **pop**, and define **past** and **future** to be references to initially empty stacks. Since the stack arguments to **push** and **pop** are to be side effected, we pass references to the stacks instead of the stacks themselves.

```
(define push
  (lambda (stk-ref val)
     (set-ref! stk-ref (cons val (deref stk-ref)))))


(define pop
   (lambda (stk-ref)
      (let ([stk (deref stk-ref)])
           (if (null? stk)
               (lambda (x) x)   ; the identity function
               (begin0 (car stk)
                  (set-ref! stk-ref (cdr stk)))))))


(define past (ref nil))
(define future (ref nil))
```

pop returns the identity function if the stack is empty. Because Scheme is untyped, objects of any type may be pushed on a stack, including continuations.

milestone grabs its continuation with **call/cc**, pushes the continuation on the **past** stack, and returns its single argument. A milestone is thus an identity function with a side effect on the **past** stack.

```
(define milestone
   (lambda (x)
      (call/cc (lambda (k)
                  (begin (push past k)
                         x)))))
```

The **devil** procedure also grabs its continuation and saves it by pushing it on the **future** stack. It then pops the **past** stack, obtaining the state of control at the last milestone, and invokes this continuation with the value of its single argument. The computation continues as if the last milestone procedure had returned this value.

```
(define devil
   (lambda (x)
      (call/cc (lambda (k)
                  (begin (push future k)
                         ((pop past) x))))))
```

Finally, the **angel** procedure pops the **future** continuation stack, obtaining the state of control at the time the last devil was invoked, and passes the devil's continuation the value of the angel's single argument. This resumes the computation as if the devil had returned this value.

```
(define angel
   (lambda (x)
      ((pop future) x)))
```

If the **future** stack is empty, the **angel** procedure acts as an identity function, as does the **devil** procedure when the **past** stack is empty.

5 Significance

Beyond their entertainment value, devils and angels may be used to implement break packages and other systems in which both forward and backward transfers of control are desired. These mechanisms are more powerful than the traditional break packages of Lisp programming environments that are stack based. Such systems allow backing out of nested break levels, but not the direct resumption of a previous break state. Continuations are the necessary tool to effect this behavior. This implies that languages lacking continuations as first class objects cannot begin to address these concerns in a coherent way.

In solving our problem, we recorded continuations on a pair of stacks. However, storing continuations in other data structures is possible. Coroutines may be implemented by recording continuations in *own* variables. Artificial intelligence applications with extensive backtracking may utilize far more sophisticated data structures. Prolog implementations contain a variety of constructs for backtracking,

but lack the capabilities to resume a prior state or manipulate continuations as data.

A naive view of continuations is found in many imperative languages that provide labels and gotos. There is no concept of a continuation within an expression; it is impossible to suspend and then later resume the computation of an expression with a given value. Furthermore, the extent of continuations is limited by the traditional stack discipline. We feel such constraints are too restrictive; they imply that if a designer wishes to provide such facilities as coroutines, he must find other ways. If he instead allowed for a more generalized means of transferring control—continuations as first class objects—he could leave this decision to the user. The designer need not make any particular choices which might render the language unsuitable for some applications.

We have demonstrated the power of first class continuations and have argued that languages that rely upon stack allocation of their control information do not provide equivalent power. So we must ask: is the efficiency of stack allocation still worth the sacrifice?

References

1. Daniel P. Friedman, Christopher T. Haynes, Eugene Kohlbecker, and Mitchell Wand, "Scheme 84 Reference Manual: Preliminary Version", November 1983.
2. Carl Hewitt, "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.
3. Peter J. Landin, "A correspondence between ALGOL 60 and Church's Lambda Notation", *CACM* **8**, 2–3, February and March 1965, pages 89–101 and 158–165.
4. Jonathan A. Rees and Norman I. Adams IV, "T: A dialect of Lisp or, LAMBDA: The ultimate software tool", Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, pages 114–122.
5. John C. Reynolds, "Definitional interpreters for higher order programming languages", *ACM Conference Proceedings 1972*, pages 717–740.
6. Guy Lewis Steele Jr. and Gerald Jay Sussman, "The revised report on Scheme: a dialect of Lisp", MIT Artificial Intelligence Memo 452, January 1978.

7. Guy Steele, "Common Lisp Reference Manual", Carnegie-Mellon University Department of Computer Science Spice Project, November 1982. To appear, Digital Press, Bedford MA.

8. Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, MIT Press, Cambridge MA, 1977.

9. Christopher Strachey and Christopher P. Wadsworth, "Continuations—a Mathematical semantics for handling full jumps", Technical Monograph PRG-11, Programming Research Group, University of Oxford (1974).

10. Gerald Jay Sussman and Drew Vincent McDermott, "From PLANNER to CONNIVER—A genetic approach", Joint Computer Conference Proceedings 41, part II, AFIPS Press, NJ, pages 1171-1179.

11. Gerald Jay Sussman and Guy Lewis Steele Jr., "Scheme: an interpreter for extended lambda calculus", MIT Artificial Intelligence Memo 349, December 1975.

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