Embedding Continuations in Procedural Objects

by

Christopher T. Haynes & Daniel P. Friedman

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 213

Embedding Continuations
in Procedural Objects

by

C. T. Haynes & D. P. Friedman

January, 1987

Embedding Continuations in Procedural Objects*

Christopher T. Haynes, Daniel P. Friedman

Abstract

Continuations, when available as first-class objects, provide a general control abstraction in programming languages. They liberate the programmer from specific control structures, increasing programming language extensibility. Such continuations may be extended by embedding them in procedural objects. This technique is first used to restore a fluid environment when a continuation object is invoked. We then consider techniques for constraining the power of continuations in the interest of security and efficiency. Domain mechanisms, which create dynamic barriers for enclosing control, are implemented using fluids. Domains are then used to implement an unwind-protect facility in the presence of first-class continuations. Finally, we present two mechanisms, wind-unwind and dynamic-wind, that generalize unwind-protect.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**] Language Classifications—*extensible languages; Scheme; Lisp* D.4.3 [**Programming Languages**] Language Constructs—*control structures;*

General Terms: Languages, Security

Additional Key Words and Phrases: continuations, first-class objects, escapes, labels

---

* A preliminary version of this paper was presented at the Twelfth Annual ACM Symposium on Principles of Programming Languages (1985).

This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567, MCS 83-03325, DCS 83-03325 and DCS 85-01277.

Authors' Address: Computer Science Department, Indiana University, Lindley Hall 101, Bloomington, IN 47405; Haynes@indiana.csnet, Friedman@indiana.csnet

1. Introduction

Control objects, such as Algol 60 labels, are not new to programming languages. Such objects, however, are not first-class: though they may sometimes be passed to procedures, they may not be returned by procedures or entered in data structures. This is a consequence of the stack allocation of both environment information (parameters and local storage) and control information (return address and context). When control returns from a given context the stack is popped, destroying information which is essential if control is ever to return to the context.

Some Lisp dialects provide mechanisms similar to labels in an expression-oriented rather than statement-oriented context, using *catch* and *throw*. A catch expression marks the current *continuation* (control context) with a given tag before evaluating its body. During the evaluation of the body, it is then possible to throw an arbitrary value v to the continuation associated with the tag. This has the effect of immediately returning v as the value of the entire catch expression. For example, during evaluation of the Common Lisp [15] expression

```
(+ 1
   (catch 'k
     (* (throw 'k 2) 3)))
```

the catch subexpression evaluates to 2 without the multiplication being performed, and 3 is returned as the value of the entire expression. Catch is implemented by placing the tag on the top of the control stack before evaluating the body of the catch expression. In the example above, at the time the throw expression is evaluated the stack would be as indicated in Figure 1, and the effect of the throw would be to pop the stack down through the tag frame, leaving the value 2 for the addition application. As with the labels discussed above, once the catch expression has returned (either normally or by a throw to its tag) it is no longer possible to invoke its continuation (throw to its tag). Though catch allows a continuation to be tagged, it does not make continuations first-class objects of

computation. The principal uses of this mechanism are error exits and blind backtracking.

In this paper we are concerned with first-class continuations—control objects that may be passed to and returned by procedures and stored indefinitely in data structures. As we shall demonstrate, it is possible using such continuations to form branches in the control structure. Control may then take on a tree structure (rather than being strictly linear) and may therefore require heap (rather than stack) allocation.

Given access to first-class continuations, it seems possible to extend a language to include any sequential control abstraction. For example, coroutines may be implemented using continuations [8]. (The converse is not true.) In the context of operating systems, continuations provide a natural way to record process state prior to a preemption [18] or a trap [7]. In artificial intelligence, continuations provide a ready means to implement non-blind backtracking [4,6,16].

Continuations are thus a powerful tool for language extensibility. This is particularly true when continuations may be recorded in the local state of first-class procedures. Such procedural objects may then be used to restrict or extend the semantics of continuations. Such procedural embedding of continuations may allow more efficient implementation, aid in reasoning about programs, help enforce security requirements, or assure the integrity of other language features such as fluid bindings. In this paper we demonstrate techniques for achieving such goals.

The next section provides a brief overview of Scheme [3,13], a Lisp dialect with first-class continuations that is used to express subsequent examples. This is followed by an introduction to first-class continuations. We next illustrate how continuations may be enhanced by embedding them in procedural objects. Such objects are then used in several simple Scheme programs that restrict or extend continuations in various ways. As our final example, we present an implementation of the *unwind-protect* facility of some Lisp systems, and two generalizations of unwind-protect that account for the first-class nature

⟨expression⟩ ::=
    ⟨constant⟩
  | ⟨variable⟩
  | ⟨syntactic extension⟩
  | (**quote** ⟨object⟩)
  | (**begin** ⟨expression⟩$^+$)
  | (**begin0** ⟨expression⟩$^+$)
  | (**lambda** (⟨variable⟩*) ⟨expression⟩$^+$)
  | (**let** ([⟨variable⟩ ⟨value⟩]*) ⟨expression⟩$^+$)
  | (**letrec** ([⟨variable⟩ ⟨value⟩]*) ⟨expression⟩$^+$)
  | (**rec** ⟨variable⟩ ⟨expression⟩)
  | (**if** ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
  | (**when** ⟨expression⟩ ⟨expression⟩)
  | (**case** ⟨tag⟩ [⟨symbol⟩ ⟨expression⟩$^+$]$^+$)
  | (**evcase** ⟨tag⟩ [⟨value⟩ ⟨expression⟩$^+$]$^+$)
  | (**define** ⟨variable⟩ ⟨expression⟩)
  | (**set!** ⟨variable⟩ ⟨expression⟩)
  | ⟨application⟩

⟨value⟩, ⟨tag⟩, ⟨procedure⟩ ::= ⟨expression⟩

⟨syntactic extension⟩ ::= (⟨keyword⟩ ⟨object⟩*)

⟨application⟩ ::= (⟨procedure⟩ ⟨expression⟩*)

Figure 2. Syntax of a Scheme subset.

of continuations.

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. [17] as part of an effort to understand the actor model of computation [9]. Scheme is a dialect of Lisp that is applicative order, lexically scoped, and properly tail-recursive. Most importantly, Scheme—unlike most other Lisp dialects—treats procedures and continuations as first-class objects.

See Figure 2 for the syntax of a Scheme subset sufficient for the purposes of this paper. The superscript * denotes zero or more, and $^+$ denotes one or more occurrences of the preceding form. Square brackets are interchangeable with parentheses, and are

used in the indicated contexts for readability. **quote** expressions return the indicated literal object, and '⟨object⟩ is equivalent to (**quote** ⟨object⟩). **begin** (**begin0**) expressions evaluate their expressions in order and return the value of the last (first). Expression lists in **lambda**, **let**, **letrec**, **case**, and **evcase** are implicit **begins**. **lambda** expressions evaluate to first-class procedural objects that statically bind their variables when invoked. **let** makes lexical bindings and **letrec** makes recursive lexical bindings. **rec** is equivalent to (**letrec** ([⟨variable⟩ ⟨expression⟩]) ⟨variable⟩). **if** evaluates its second expression if the first is true, and the third otherwise. **when** evaluates its second expression if the first is true and returns an irrelevant value otherwise. **case** evaluates the tag expression, and then returns the value of the first expression whose corresponding symbol matches the tag. If the last symbol is **else**, it always matches. **evcase** is similar to **case**, but the ⟨value⟩ expressions are evaluated. **define** assigns to a global variable. **set!** modifies an existing lexical variable; if no lexical binding is apparent, it is assumed that a global binding exists and is assigned the given value. An application evaluates its expressions (in an unspecified order) and applies the procedural value of the first expression to the values of the remaining expressions.

Most Scheme implementations provide a syntactic preprocessor that examines the first object in each expression. If the object is not a keyword, then it is assumed that the expression is an application. If the object is a syntactic extension (macro) keyword, then the expression is replaced by an appropriately transformed expression. In this paper the symbol ≡ indicates syntactic extensions. (Ideally, the syntactic extensions should be hygienic. [10])

We require nine primitive functions. *eq?* returns true if its arguments are the same reference. *not* is logical negation. *unique* returns a unique new reference that is *eq?* to itself, but to no other object. *cons* is the conventional Lisp list structure constructor. *delq!* deletes an indicated element from a list. *reverse!* (also referred to as *nreverse* [1]) does an in-place reversal of a list by modifying the list links. *for-each* (*mapc*) applies a

given procedure to each element of a list (and discards the results). *thaw* receives a *thunk* (a nullary procedure) and invokes it (useful as an argument to *for-each*). The boolean constants true and false are denoted by #t and #f, respectively.

The procedure *call-with-current-continuation*, abbreviated *call/cc*, evaluates its argument and applies it to the current continuation, represented as a first-class procedural object of one argument. In the next section we present an informal description of the semantics of such continuations.

3. An introduction to continuations

During the evaluation of an expression the system continually keeps track of the current context of evaluation. The evaluation of each subexpression has a different context that *controls* how the value of the subexpression will be used to *continue* the computation. Thus contexts of evaluation are called *control contexts* or *continuations*. The continuation of each subexpression may be represented as a function of one argument—the value of the corresponding subexpression.

For example, consider the expression $(+\ 1\ (*\ 2\ 3))$. We may represent the seven distinct continuations created during its evaluation as follows:

$$
\begin{aligned}
k1 &= (\text{lambda}\ (v)\ (+\ 1\ (*\ 2\ v))) \\
k2 &= (\text{lambda}\ (v)\ (+\ 1\ (*\ v\ 3))) \\
k3 &= (\text{lambda}\ (v)\ (+\ 1\ (v\ 2\ 3))) \\
k4 &= (\text{lambda}\ (v)\ (+\ 1\ v)) \\
k5 &= (\text{lambda}\ (v)\ (+\ v\ (*\ 2\ 3))) \\
k6 &= (\text{lambda}\ (v)\ (v\ 1\ (*\ 2\ 3))) \\
k7 &= (\text{lambda}\ (v)\ v).
\end{aligned}
$$

Invocation of any of these continuation with the value of its corresponding subexpression yields the value of the entire expression; for example, $(k4\ \ (*\ 2\ 3)) = 7$. If the expression $(+\ 1\ (*\ 2\ 3))$ were being evaluated as part of some larger expression, and that part had some continuation K, we could easily modify each of the above continuations so that it represented the control context within the larger evaluation; for example,

$k4$ = (lambda (*v*) (*K* (+ 1 *v*))).

Though any programming system maintains the current continuation of each expression it evaluates, these continuations are generally not accessible to the programmer. However, in Scheme the primitive function *call/cc* causes the current continuation to be packaged as a first-class procedure of one argument and passed to a procedure provided by the programmer. This continuation represents the remainder of the computation from the *call/cc* application point. At any time this continuation may be invoked with any value, with the effect that this value is taken as the value of the *call/cc* application. (The continuation of a continuation application is discarded, unless it has been saved with another *call/cc*.) Related facilities include Landin's *J* operator [2,11] and Reynolds' *labels* and *escapes* [12,14].

The simplest use of continuations is as escape procedures. For example, the *catch* example above may be expressed as

```
(+ 1
   (call/cc
     (lambda (k)
       (* (k 2) 3))))
```

where k is bound to a continuation that is equivalent to $k4$ above. (We generally use k to represent continuations by analogy with denotational semantics, in which meta-level continuations are typically denoted by κ.) This example does not exploit the fact that Scheme continuations are first-class, and control remains linear; the effect of invoking k is simply to backtrack to a previous point in the control environment. This backtracking is blind in the sense that it is impossible to return to the point from which the backtracking was initiated, since the continuation of that point has not been saved.

To illustrate the use of first-class continuations, we present an example of non-blind backtracking. Suppose there are two approaches to solving some problem, with no *a priori* way of knowing which is best. (Perhaps the approaches are different branches of a search

tree.) If the answer has not been found after some time spent exploring the first approach, we wish to try the second approach; that is, we wish to backtrack to the original choice point and try the other choice. However, in the event that the second approach fails to arrive at an answer, we wish to return to our previous place in the computation of the first approach and continue from there. The backtracking from the first approach must have been *non-blind* for this to be possible. This behavior is achieved with the following program schema:

```
(let ([backtrack-k any]
      [non-blind-k any])
  (if (call/cc
        (lambda (k)
          (set! backtrack-k k)
          (backtrack-k #t)))
      (begin
        begin the first approach
        (when time-to-backtrack
          (call/cc
            (lambda (k)
              (set! non-blind-k k)
              (backtrack-k #f))))
        continue computing)
      (begin
        try another approach
        (if have-answer
            answer
            (non-blind-k any)))))
```

(We use *any* to indicate an irrelevant value.) Here the continuation of *backtrack-k* is to branch conditionally on the value passed to the continuation. At first this continuation is invoked with #t from the first *call/cc* expression. If the first approach does not look promising, then *backtrack-k* is invoked with #f, causing the second approach to be tried; but before invoking *backtrack-k* the current continuation is saved in *non-blind-k*. If the second approach fails, *non-blind-k* is invoked (with an arbitrary value), thereby resuming exploration of the first approach.

With first-class continuations it is possible to form branches in the control structure,

which is then in the form of a tree, rather than the traditional linear (stack) structure. By invoking continuations, it is possible to jump between any two nodes of this tree. For example, the dotted line in Figure 3 indicates the control transition that occurs when the *non-blind-k* continuation is invoked in the above example. First-class continuations require that control information be heap allocated, at least when branching actually takes place.

4. Enhancing Continuations

To enforce constraints, or extend the semantics of continuations, we define (as Scheme procedures) modified versions of *call/cc* which pass a newly formed procedural object, or *closure,* to its argument rather than a true continuation. In order to distinguish such a closure that contains a continuation from a plain continuation, we refer to it as a *Continuation OBject*, or *cob*. When invoked, a cob performs any additional operations that we require, and then (conditions permitting) invokes the embedded continuation (or cob). Embedded continuations must be obtained using the original *call/cc* at the time the cob is created, and be retained in the environment of the cob.

```
(define call/cc-whatever
  (lambda (f)
    (call/cc
      (lambda (k)
        (let ([cob (lambda (v)
                     whatever is needed
                     (k v))])
          (f cob))))))
```

If the cob must be invoked upon implicit as well as explicit invocation of the continuation, then (*f cob*) should be replaced by (*cob* (*f cob*)).

To illustrate this technique, consider the implementation of a fluid environment [14] using the standard functional representation of environments. The fluid environment, represented by the global *fluid-env*, is extended upon entering a **let-fluid** body, and restored when leaving the body, thereby providing dynamic extent for fluid bindings.

```
(fluid ⟨var⟩) ≡ (fluid-env '⟨var⟩)

(let-fluid ⟨var⟩ ⟨exp⟩ ⟨body⟩) ≡
(let ([own-env fluid-env]
      [v ⟨exp⟩]
      [body-thunk (lambda () ⟨body⟩)])
  (set! fluid-env
    (lambda (x)
      (if (eq? x '⟨var⟩) v (own-env x))))
  (begin0
    (body-thunk)
    (set! fluid-env own-env)))
```

Here we use the standard technique for extending functional environments. The closure formed by evaluating the (**lambda** (*x*) ...) expression records the binding of *v* in its local environment. It is assumed that *fluid-env* is initially bound to a function that issues an error message.

This extension will not work in the presence of unrestricted continuations. When a continuation is invoked, the computation should continue in the same environment in which the continuation was obtained. Thus, the current fluid environment must be recorded when a continuation is obtained, and this environment must be restored when the continuation is invoked. This may be achieved by redefining *call/cc* as follows:

```
(define call/cc-fluid
  (lambda (f)
    (call/cc
      (lambda (k)
        (f (let ([own-env fluid-env])
             (lambda (v)
               (set! fluid-env own-env)
               (k v))))))))
```

In this case we avoid the (*cob* (*f* *cob*)) construction because implicit invocation of continuations always occurs with the right fluid environment. Only explicit invocation of the continuation requires that the cob's actions be performed.

5. Constraining Continuations

We now provide a sequence of examples that illustrate approaches to constraining continuations.

One-shot continuations

First we demonstrate a version of *call/cc*, referred to as *call/cc-one-shot*, which delivers explicit continuations that may only be invoked once. One can imagine certain implementation techniques for which it would be necessary to enforce this one shot constraint. For example, it would be necessary if the heap space used to record continuation frames were automatically reclaimed upon their invocation.

This constraint may be enforced by associating a variable with each cob that records whether the continuation has been invoked yet (whether it is still alive).

```
(define call/cc-one-shot
  (lambda (f)
    (call/cc
      (lambda (k)
        (f (let ([alive #t])
             (lambda (v)
               (if alive
                   (begin
                     (set! alive #f)
                     (k v))
                   (error ...)))))))))
```

Stack-based continuations

Though *call/cc-one-shot* assures that a given continuation may only be invoked once, it is still possible for continuations that are its descendents to be invoked. (If control returns in the usual way, the previously invoked continuation will eventually be reinvoked, and the error detected. However, this detection may be too late, or may not occur at all if control jumps to another branch of the continuation tree before the previously invoked continuation is reinvoked.) We now wish to enforce the constraint that, when a continuation is invoked,

neither it nor any of its descendents may be invoked a second time. This suffices to ensure that control information may be stack rather than heap allocated.

To enforce this constraint we keep an *alive* flag in each cob, as before. However, this time when a cob is invoked, not only must its own flag be set to #f, but also that of each cob below it in the cob stack. For this, each cob maintains a reference to its child cob. (In general a cob may have multiple children, but the discipline imposed here insures that there is at most one child.) Each cob is an object that can respond to messages, and only performs as a continuation if its argument is not a recognized message. When a cob is invoked, it simply clears its *alive* flag and then sends a *kill!* message to its child (if there is one) to do the same. In this way the flags of all the descendent cobs are cleared, as required.

To complete this scheme, we must provide a method for setting the child references. When a cob is created, it installs a reference to itself in its parent cob. For this purpose a reference to the parent cob is maintained as the fluid binding of *parent-cob*, and all cobs are made to respond to a *set-child!* message in order that the child reference may be recorded.

The symbols 'kill!' and 'set-child!' are not used as cob messages. They might be mistaken for values passed to the embedded continuation. Instead, we use values returned by *unique*, which cannot be confused with any others. See Figure 4. Note that *child* must be initialized with a cob that can absorb *kill!* messages sent to it, but do nothing. The initial fluid environment must include a binding such as

(lambda (*x*) (lambda (*y*) *any*))

for *parent-cob* that can absorb *set-child!* messages.

The *child* references in *call/cc-stack-based* present a potential problem with respect to garbage collection. If the child reference were maintained after invocation of the parent, the child would not be collectible even though the child can no longer be invoked and there may be no other references to it. Thus after the child is invoked, but before it returns, the

```
(let ([kill! (unique)] [set-child! (unique)])
  (set! call/cc-stack-based
    (lambda (f)
      (begin0
        (call/cc-fluid
          (lambda (k)
            (letrec
              ([cob (let ([child (lambda (x) any)]
                          [alive #t])
                      (lambda (v)
                        (evcase v
                          [kill! (set! alive #f) (child kill!)]
                          [set-child! (lambda (n) (set! child n))]
                          [else (if alive
                                    (begin (cob kill!) (k v))
                                    (error ...))]))))])
              (((fluid parent-cob) set-child!) cob)
              (let-fluid parent-cob cob (cob (f cob))))))
        (((fluid parent-cob) set-child!) (lambda (x) any))))))
```

Figure 4. call/cc-stack-based

child reference of its parent is replaced by a dummy value.

Dynamic domains

In some contexts it may be necessary to restrict the range of control jumps to some dynamic context, which we refer to as a *domain*. We accomplish this by defining the procedure *domain* that takes a thunk and thaws it with a new unique reference fluidly bound to *domain-ref*. *call/cc-domain* provides a cob that signals an error unless the fluid binding of *domain-ref* at the time of its invocation is the fluid binding of *domain-ref* at the time of its creation. *domain-ref* must be bound in the initial fluid environment.

```
(define domain
  (lambda (thunk)
    (let-fluid domain-ref (unique) (thunk))))

(define call/cc-domain
  (lambda (f)
    (call/cc-fluid
      (lambda (k)
        (f (let ([own-d (fluid domain-ref)])
             (lambda (v)
               (if (eq? (fluid domain-ref) own-d)
                   (k v)
                   (error ...)))))))))
```

Other forms of domain restriction are possible. For example, we may allow control to exit from a domain by invocation of a continuation, but still prevent control from reentering the domain. To implement such an *exit-only-domain*, we extend the above code with a *domain-env* that returns #t only if invoked with the *domain-ref* of a currently active domain. Now the initial fluid environment must also bind *domain-env* to (lambda (*x*) #f).

```
(define exit-only-domain
  (lambda (thunk)
    (let-fluid domain-ref (unique)
      (let-fluid domain-env
        (let ([own-d (fluid domain-ref)]
              [own-env (fluid domain-env)])
          (lambda (d)
            (if (eq? d own-d)
                #t
                (own-env d))))
        (thunk)))))

(define call/cc-exit-only-domain
  (lambda (f)
    (call/cc-fluid
      (lambda (k)
        (f (let ([own-d (fluid domain-ref)])
             (lambda (v)
               (if ((fluid domain-env) own-d)
                   (k v)
                   (error ...)))))))))
```

6. Unwind-Protect and Wind-Unwind

Many Lisp systems provide an unwind-protect facility, that might have the syntax:

(*unwind-protect* ⟨body⟩ ⟨postlude⟩).

Normally ⟨body⟩ is evaluated first, and then ⟨postlude⟩ is evaluated. However, if control passes out of ⟨body⟩ prematurely through invocation of an outer control context, then ⟨postlude⟩ is evaluated immediately before the invocation takes place. A typical use of unwind-protect is to assure that any files opened by ⟨body⟩ are closed whenever control leaves ⟨body⟩. Unwind-protect is particularly valuable in designing fault-tolerant systems, where the ⟨postlude⟩ may assure that the system is left in a stable state in the event that an error or other exceptional condition requires that the control context shift abruptly.

It is straightforward to implement unwind-protect in Lisp systems whose control obeys the stack discipline, but it is more complicated to define and implement an unwind-protect mechanism in the presence of full continuations. For example, when control passes from

one branch of a control tree to another, (1) should unwind-protects only be triggered on the path between the current node and the closest common ancestor of the current node and destination node, or (2) should unwind-protects be triggered only if they are ancestors of the current node and descendents of the destination node? Also, if the same unwind-protect is triggered more than once, (1) should the ⟨postlude⟩ code be executed each time, or (2) should the ⟨postlude⟩ code be executed just the first time? We believe that there are probably no general answers to questions such as these. Rather, programmers who deal directly with such tools must be aware of these issues and answer these questions in light of each application's requirements. Scheme is flexible enough to implement variations such as those above, and it is important to leave programmers with the ultimate choice. However, if one unwind-protect facility is to be provided as part of the standard language (as it probably should), a design decision must be made opting for one solution that is reasonably simple and generally applicable. More research is needed, but for the time being we choose the second option for each of the above questions.

To implement unwind-protect, we could use child references, such as were used in the *call/cc-stack-based* solution. A list of references for each continuation would have to be maintained, rather than a single reference, because in the current context the continuation tree may branch. However, this would be undesirable. Without knowing what cobs the user has maintained references to, child references can not be deliberately erased. The result is that cobs could never be reclaimed by the garbage collector.

Instead of child references, each cob maintains a list of thunks that when thawed perform the unwind-protect procedures necessary when the cob is invoked. Our unwind-protect installs its thunk, which we call *unwind*, in each ancestor cob. When thawed, each *unwind* causes the ⟨postlude⟩ associated with its unwind-protect to be evaluated and then removes itself from all the cob lists in which it was installed. The addition and deletion of *unwind*s from the cob lists is achieved by sending unique messages to the cobs, as before. The most recent cob is always accessible as the fluid binding of *parent-cob*. We assume

that *unwind-protect* receives *body* and *postlude* arguments that are thunks containing the ⟨body⟩ and ⟨postlude⟩, respectively. Unwind-protect is then implemented as in Figure 5, ignoring the code within boxes. The initial fluid environment binds *parent-cob* as in *call/cc-stack-based*.

If it were possible for arbitrary continuations to be invoked from within a ⟨postlude⟩, or for continuations obtained from within a ⟨postlude⟩ to be invoked elsewhere, then unwind-protect could be subverted. By associating a domain with the invocation of *postlude*, we obtain precisely the degree of protection required.

In a traditional Lisp system where control is linear (stack-based), it is not possible for control to reenter an unwind-protect body after control has left the body and the unwind has been performed. However, with first-class continuations, this is possible and raises a new problem. The ⟨postlude⟩ expression frequently performs operations, such as closing files, that should themselves be undone if control reenters the body. We can extend the unwind-protect mechanism so that some winding expression, say ⟨prelude⟩, is executed upon any entry of the body, as well as providing an unwinding ⟨postlude⟩ that is executed upon exit. We call such a mechanism *wind-unwind*, and use the syntax

(*wind-unwind* ⟨prelude⟩ ⟨body⟩ ⟨postlude⟩).

Wind-unwind requires only a few extensions to our previous unwind-protect code. Each wind-unwind creates a thunk. When this thunk is thawed it will, if necessary, invoke the ⟨prelude⟩ of wind-unwind, and then thaw the thunk of the previous wind-unwind. A fluidly-bound *wind* reference is maintained in much the same manner as the fluid *parent-cob* reference to record the chain of preceding wind-unwind thunks. The initial fluid environment includes a binding for *wind*, such as (lambda () *any*). Every cob records the value of the fluid *wind* variable at the time of its creation as *own-wind*. Before invoking its continuation, *own-wind* is thawed, thus initiating a chain of operations that performs any required *prelude* operations. In order that *preludes* be performed only following a

```
(let ([update (unique)] [delete (unique)])
  (set! [wind-unwind ; or] unwind-protect ; without boxed code
    (lambda ([prelude] body postlude)
      (let ([own-cob (fluid parent-cob)]
            [own-wind (fluid wind)]
            [in #t])
        (letrec ([unwind (lambda ()
                           (domain postlude)
                           (set! in #f)
                           ((own-cob delete) unwind))])
          ((own-cob update) unwind)
          (let-fluid wind (lambda ()
                            (if (not in)
                                (begin
                                  (domain prelude)
                                  (set! in #t)))
                            (own-wind))
            (begin (domain prelude)
                   (begin0 (body) (unwind))))))))
  (set! [call/cc-wind-unwind ; or] call/cc-unwind-protect ; without boxed code
    (lambda (f)
      (call/cc-domain
        (lambda (k)
          (let ([cob (let ([unwinds '()]
                           [own-cob (fluid parent-cob)]
                           [own-wind (fluid wind)])
                       (lambda (v)
                         (evcase v
                           [update (lambda (x)
                                     (set! unwinds (cons x unwinds))
                                     ((own-cob update) x))]
                           [delete (lambda (x)
                                     (set! unwinds (delq! x unwinds))
                                     ((own-cob delete) x))]
                           [else (own-wind) (for-each thaw unwinds) (k v)])))])
            (let-fluid parent-cob cob (cob (f cob)))))))))
```

Figure 5. *wind-unwind* and *unwind-protect*

corresponding *postlude* operation, a flag is set when a *postlude* is performed and cleared when the corresponding *prelude* is performed.

Wind-unwind generally has the right semantics for such operations as opening and closing files. A *postlude* is only performed when a direct ancestor of its continuation is invoked. It is thus possible to transfer control to other continuations without the overhead of invoking *postlude*s and *prelude*s. For example, a coroutine resume operation involves a transfer from one node of the continuation tree to another, and it is probably inappropriate for each resume to close and open files associated with a coroutine.

7. Dynamic-Wind

A similar facility, called *dynamic-wind*, has the same syntax as wind-unwind, but subtly different semantics. Dynamic-wind may be used to implement a fluid environment with shallow binding.

```
(bind-fluid ⟨var⟩ ⟨exp⟩ ⟨body⟩) ≡
(let ([oldvar ⟨exp⟩])
  (let ([swap! (lambda ()
                 (let ([temp oldvar])
                   (set! oldvar ⟨var⟩)
                   (set! ⟨var⟩ temp)))])
    (dynamic-wind swap! (lambda () ⟨body⟩) swap!)))
```

Here existing lexical variables record the current fluid environment values, whereas the **let-fluid** mechanism maintains the fluid environment without side-effects to the lexical environment.

If a continuation were invoked that was not an ancestor of the current continuation and utilized the same lexical variable with a different fluid binding, then wind-unwind would not yield the desired semantics. The *postlude*s and *prelude*s that maintain the fluid environment would not be performed. The more dynamic dynamic-wind avoids this problem, but requires an associated *state-space* [1,5].

A state-space is a tree where the root is the current state. It is possible to move

from the current state to any other state, making the corresponding node the new root. (Think of picking up the tree by any node and giving it a good shake so that all paths lead to the new root.) Each dynamic-wind creates a new state that becomes the root of the state-space. When a continuation is invoked, the unique path through the state-space is traversed, with the *postlude*s or *prelude*s associated with each node being performed as they are passed. Whenever control passes out of *body*, its *postlude* is performed and it is noted that control has exited. Conversely, when control passes back in, the *prelude* is performed and it is noted that control has reentered.

In the code of Figure 6 the state-space is extended with a cons cell, cdr of which is initially null, and car of which is a thunk that will be thawed when the current state moves past the cell during a reroot operation. Note that this thunk is to do nothing in the event that control is already in its state (*in* is #t) and it is the destination of a reroot operation (indicated by a null list in the cdr of its state cell). *state-space* is an object that responds to the messages 'state', 'extend' and 'reroot'. The list being reversed represents the path from the new state to the current state.

While useful for shallow binding, dynamic-wind may cause *prelude*s and *postlude*s to be invoked too frequently for applications such as file housekeeping. Also, the current and destination continuations may use different lexical variables to record fluid bindings (as would probably be the case in a coroutine environment), so dynamic-wind is again unnecessarily performing *prelude*s and *postlude*s. This could be avoided by associating a different state-space with each set of fluidly-used lexical variables. The lexical scope of each of these state-spaces would then require its own *dynamic-wind*, *call/cc-dynamic-wind*, and bind-fluid operations.

8. Conclusions

Throughout the history of programming languages, many developments have either provided more general facilities or restricted the power of existing facilities. For example,

```
(define state-space
  (let ([state (cons (lambda () any) '())])
    (lambda (msg)
      (case msg
        [state state]
        [extend (lambda (new-state)
                  (set-cdr! state new-state)
                  (set! state new-state))]
        [reroot (lambda (new-state)
                  (reverse! new-state)
                  (for-each thaw state)
                  (set! state new-state))])))))

(define dynamic-wind
  (lambda (prelude body postlude)
    (let ([state (state-space 'state)])
      ((state-space 'extend)
       (let ([in #t])
         (rec local-state
           (cons
             (lambda ()
               (if (and in (null? (cdr local-state)))
                   'do-nothing
                   (begin
                     (domain (if in postlude prelude))
                     (set! in (not in)))))
             '())))))
      (domain prelude)
      (begin0 (body)
        ((state-space 'reroot) state)))))

(define call/cc-dynamic-wind
  (lambda (f)
    (call/cc-domain
      (lambda (k)
        (let ([state (state-space 'state)])
          (let ([cob (lambda (v)
                       ((state-space 'reroot) state)
                       (k v))])
            (f cob)))))))
```

Figure 6. *dynamic-wind* state space.

much attention has been given to control constructs that allow *goto*'s to be restricted or eliminated. Also, abstract data types provide restrictions on the scope of variables. Continuations are more general than other control facilities, and we have only begun to explore their power. However, continuations are clearly too powerful for many applications.

We have illustrated some techniques for constraining this power. In this process we have emphasized semantics, while ignoring some security and efficiency issues. For security, one should redefine *call/cc*, rather than creating variations with new names. Subversion of *call/cc* by tampering with the fluid environment could be prevented by appropriate use of scope control. For efficiency, the fluid environment could be represented as a data structure, rather than as a procedure.

In recent years attention has shifted from problems of sequential control to those of parallel control. Presumably it was felt that sequential control was well understood. However, we are convinced by our experience with continuations that there is still much to be learned about sequential control. Research on sequential control has a special urgency as we plunge into the complexities of parallel control.

Acknowledgements: Dynamic-wind was originally suggested by Richard Stallman. Gerald Sussman clarified how *dynamic-wind* could be used to implement **bind-fluid**. Kent Dybvig suggested that fluids could be implemented by redefining *call/cc*. Comments by Don Oxley motivated the *domain* restriction. Amitabh Srivastava suggested improvements in the state-space code of an earlier version of this paper. We thank Mitch Wand, George Springer, Eugene Kohlbecker, Gary Brooks, Bruce Duba, Matthias Felleisen, and an anonymous referee for detailed comments on this paper.

References

[1] Baker, H. G., Jr., Shallow Binding in Lisp 1.5. *Commun ACM 21* (July 1978), 565–569.

[2] Burge, W. H., *Recursive Programming Techniques.* Addison-Wesley, Reading, Mass., 1975.

[3] Dybvig, R. K., *The Scheme Programming Language.* Prentice-Hall, 1987.

[4] Friedman, D. P., Haynes, C. T., and Kohlbecker, E., Programming with Continuations. *Program Transformation and Programming Environments,* ed. P. Pepper, Springer-Verlag, 1984, 263–274.

[5] Hanson, C., and Lamping, J., Dynamic binding in scheme. unpublished manuscript, 1984.

[6] Haynes, C. T., Logic continuations. *The Journal of Logic Programming,* to appear, and Indiana University Computer Science Technical Report No. 183, March, 1986.

[7] Haynes, C. T., and Friedman, D. P., Abstracting timed preemption with Engines. *Computer Languages,* to appear. Earlier version appeared as: Engines build process abstractions. *Conf. Rec. of the 1984 ACM Symposium on Lisp and Functional Programming* (1984), 18–24.

[8] Haynes, C. T., Friedman, D. P., and Wand, M., Obtaining coroutines with continuations. *Computer Languages,* to appear. Earlier version appeared as: Continuations and coroutines. *Conf. Rec. of the 1984 ACM Symposium on Lisp and Functional Programming* (1984), 293–298.

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

[10] Kohlbecker, E., Friedman, D.P., Felleisen, M., and Duba, B., Hygienic macro expansion. *Conf. Rec. of the 1986 ACM Symposium on Lisp and Functional Programming* (1986), 151–161.

[11] Landin, P., A correspondence between ALGOL 60 and Church's Lambda Notation. *Commun ACM 8,* 2–3 (Feb. and Mar. 1965), 89–101 and 158–165.

[12] Reynolds, J., GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept. *Commun ACM 13* (May 1970), 308–319.

[13] Rees, J. and Clinger, W., (eds.), Revised^3 Report on the Algorithmic Language Scheme, *SIGPLAN Notices 21,* 37–79 (Dec. 1986)

[14] Reynolds, J., Definitional interpreters for higher order programming languages. *Proceedings ACM Conference 1972,* 717–740.

[15] Steele, G. L., Jr., *Common Lisp: The Language.* Digital Press, Bedford, Mass., 1984.

[16] Sussman, G. J., and McDermott, D. V., From PLANNER to CONNIVER—A genetic approach. *Proceedings of Joint Computer Conference 41, part II,* AFIPS Press, N.J. (1973), 1171–1179.

[17] Sussman, G. J., and Steele, G. L., Jr., Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349 (Dec. 1975).

[18] Wand, M., Continuation-based multiprocessing. *Conf. Record of the 1980 Lisp Conference* (Aug. 1980), 19–28.