CONSTRAINING CONTROL

by

Daniel P. Friedman

and

Christopher T. Haynes
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 170

CONSTRAINING CONTROL

by

Daniel P. Friedman
and Christopher T. Haynes

May, 1985

Constraining Control*

Daniel P. Friedman

Christopher T. Haynes

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

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 functional 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 demonstrate 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 *1985 ACM Symposium on Principles of Programming Languages.* This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567 and MCS 83-03325.

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 be passed to procedures, they may not be returned 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).

Several Lisp dialects, including Common Lisp [15] and T [10], provide mechanisms similar to labels in an expression- rather than statement-oriented context, using *catch* and *throw*. A catch expression binds the current continuation, or control state, to an identifier and then evaluates the body of the catch expression. Within the dynamic scope of the body 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 continuations (though environments are now being heap allocated). The principal uses of this mechanism are error exits and "blind" backtracking. Non-blind backtracking [16], in which a computation may always be resumed at any point where a continuation has been obtained, is not possible in these languages.

The programming language Scheme [17] is unusual in that it provides continuations as first-class objects. In general, such continuations form a tree structure, and must be heap allocated. By invoking continuations, it is possible to jump between any two nodes of this tree.

Given access to continuations, it is possible to extend a language to include any desired sequential control abstraction. For example, coroutines may be implemented using continuations [7], but not *vice-versa*. In the context of operating systems, continuations provide

a natural way to record process state prior to a preemption [18] or a trap [6]. In artificial intelligence, continuations provide a ready means to implement non-blind backtracking [3].

Continuations are a powerful tool for language extensibility. We believe it is important to have such power in system development languages. However, in specific applications it is often desirable to place constraints on the use of continuations. Such constraints may allow more efficient implementation, aid in reasoning about programs, and help enforce security requirements.

The next section provides a brief overview of Scheme. We then illustrate how continuations may be enhanced by embedding them in continuation objects. Such objects are then used in several simple Scheme programs that enforce various types of constraints on continuations. Finally, we present an implementation of the *unwind-protect* facility of some Lisp systems, and two generalizations of unwind-protect. The interest in these mechanisms is their semantics in the presence of first-class 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 [8]. Scheme may be described as a dialect of Lisp that is applicative order, lexically scoped, and properly tail-recursive. Most importantly, Scheme—unlike most other Lisp dialects—treats functions and continuations as first-class objects.

See Figure 1 for the syntax of a Scheme 84 [4] subset sufficient for the purposes of this paper.† The superscript * denotes zero or more, and $^+$ denotes one or more occurrences of

† 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

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

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

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

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

Figure 1. Syntax of a Scheme 84 subset.

the preceding form. Square brackets are interchangable 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 **begin**s. **lambda** expressions evaluate to first-class functional objects that statically bind their identifiers when invoked. **let** makes lexical bindings and **letrec** makes recursive lexical bindings. **if** evaluates its second expression if the first is true, and the third 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

heavily parenthesized expressions are greatly reduced with some practice and intelligent tools, such as editors and pretty-printers [13]. 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.

(value) expressions are evaluated. `define!` assigns to a global identifier. `set!` modifies an existing lexical identifier. An application evaluates its expressions (in an unspecified order) and applies the functional value of the first expression to the values of the remaining expressions.

Scheme 84 provides 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.

We require only nine primitive functions. `eq?` returns true if its arguments are the same reference. `not` is 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. `thaw` receives a *thunk* (a nullary function) and invokes it. `mapc` applies a given function to each element of a list.

The function `call-with-current-continuation`, abbreviated `call/cc`, evaluates its argument and applies it to the current continuation, represented as a functional object of one argument. Informally, 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 [2,9] and Reynolds' *labels* and *escapes* [11,12].

3. Enhancing Continuations

To enforce constraints, or otherwise extend the semantics of continuations, we define modified verisons of `call/cc` which pass a closure rather than a true continuation to its ar-

gument. 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 id) ≡ (fluid-env 'id)

(let-fluid id exp body) ≡

(let ([own-env fluid-env]
      [v exp])
   (define! fluid-env
      (lambda (x)
         (if (eq? x 'id) v (own-env x))))
   (begin0
      body
      (define! fluid-env own-env)))
```

This extension will not work in the presence of unrestricted continuations. When a continuation is invoked, the computation continues 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)
                     (define! 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.

4. 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 true])
                 (lambda (v)
                    (if alive
                        (begin
                           (set! alive false)
                           (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 `false`, but also that of each cob below it in the cob stack. For this, each cob maintains a reference to its child cob. Also, 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

```
(let ([kill! (unique)] [set-child! (unique)])
   (define! call/cc-stack-based
      (lambda (f)
         (call/cc-fluid
            (lambda (k)
               (letrec
                  ([cob (let ([child (lambda (x) '*)]
                              [alive true])
                          (lambda (v)
                             (evcase v
                                [kill! (set! alive false) (child kill!)]
                                [set-child! (lambda (n) (set! child n))]
                                [else (if alive
                                          (begin (cob kill!) (k v))
                                          (error ...))])))])
                  (((fluid fcob) set-child!) cob)
                  (let-fluid fcob cob (cob (f cob)))))))))
```

Figure 2. `call/cc-stack-based`

a reference to the parent cob is maintained as the fluid binding of `fcob`, 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 can not be confused with any others. See Figure 2. Note that `child` must be initialized with a cob that can absorb messages sent to it, but do nothing. (We use `'*` to indicate an irrelevant value.) The initial fluid environment must include a binding such as `(lambda (x) (lambda (y) '*))` for `fcob`.

The `child` references in `call/cc-stack-based` present a potential problem with respect to garbage collection, for a cob is not collectible after the previous continuation is invoked. However, in this case there is no such difficulty. Because a sequence of continuation returns is invariably followed by extensions of the continuation stack, at which time the `child` field of the current cob will be reset, references to its old descendents will be

cut off (unless the programmer has retained other references to these dead continuations, in which case they can not be collected anyway).

Clearly `call/cc-stack-based` is weaker than `call/cc`, since it does not allow branching of the continuation tree. It is not so obvious whether `call/cc-one-shot` is equivalent in power to `call/cc`. It might seem that "one shot" continuations would be weaker, but see the Appendix for a demonstration that they are not. There we present an extraordinarily difficult program, `call/cc*`, which uses only `call/cc-one-shot`, and argue that it is equivalent to `call/cc`.

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 function `domain` that takes a thunk and thaws it with a new unique reference fluidly bound to `domain-ref`. `call/cc-domain` provides cobs that signal an error unless the fluid binding of `domain-ref` at the time of their invocation is the fluid binding of `domain-ref` at the time of their 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 true 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) false).

```
(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)
                      true
                      (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 ...)))))))))
```

5. 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* would be 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 powerful 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 performs the unwind-protect functions 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 `unwinds` 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 `fcob`. 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 3, ignoring the code within boxes. The initial fluid environment binds `fcob` 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 `fcob` reference to record the chain of preceding wind-unwind thunks. The initial fluid environment includes a binding for `wind`, such as `(lambda () '*)`. Every cob records the value of the fluid `wind` variable at the time of its creation as `own-wind`. Before invoking its continuation, `own-`

```
(let ([update (unique)] [delete (unique)])
  (define! wind-unwind ; or unwind-protect ; without boxed code
    (lambda ( prelude body postlude)
      (let ([own-cob (fluid fcob)]
            [own-wind (fluid wind)]
            [in true] )
        (letrec ([unwind (lambda ()
                           (domain postlude)
                           (set! in false)
                           ((own-cob delete) unwind))])
          ((own-cob update) unwind)
          (let-fluid wind (lambda ()
                            (if (not in)
                                (begin
                                  (domain prelude)
                                  (set! in true)))
                            (own-wind))
            (begin (domain prelude)
                   (begin0 (body) (unwind)) )) ))))
  (define! call/cc-wind-unwind ; or call/cc-unwind-protect ; without boxed code
    (lambda (f)
      (call/cc-domain
        (lambda (k)
          (let ([cob (let ([unwinds nil]
                           [own-cob (fluid fcob)]
                           [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) (mapc thaw unwinds) (k v)])))])
            (let-fluid fcob cob (cob (f cob)))))))))
```

Figure 9. wind-unwind and unwind-protect

`wind` is thawed, thus initiating a chain of operations that performs any required *prelude* operations. In order that *preludes* be performed only following a 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 *postludes* and *preludes*. For example, a coroutine resume operation involves a transfer from one node of the continuation tree to another, and it likely to be inappropriate for each resume to close and open files associated with a coroutine.

6. 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 id exp body) ≡

(let ([oldid exp])
   (let ([swap! (lambda ()
                  (let ([temp oldid])
                     (set! oldid id)
                     (set! id temp)))])
      (dynamic-wind swap! (lambda () body) swap!)))
```

Here existing lexical identifiers 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 identifier with a different fluid binding, then wind-unwind would not yield the desired semantics. The *postludes* and *preludes* 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 *postludes* or *preludes* 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 that follows, the state-space is extended with a cons cell, cdr of which is initially nil, 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 true) and it is the destination of a reroot operation (indicated by nil in the cdr of its state cell).

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

When invoked, *state-space returns a new state-space object that responds to the messages state, extend, and reroot:

```
(define! *state-space
   (lambda ()
      (let ([state (cons (lambda () '*) nil)])
         (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)
                         (mapc thaw state)
                         (set! state new-state))])))))
```

The list being reversed represents the path from the new state to the current state.

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

7. Conclusions

Throughout the history of programming languages, many developments have either provided more general facilities or restricted the power of existing facilities. For example, 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 versions 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 with an alternate data structure, rather than functional embedding.

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 brought bugs in the state-space code of an earlier version of this paper to our attention. We thank Mitch Wand, George Springer, Eugene Kohlbecker, John Nienart, Gary Brooks, Bruce Duba and Matthias Felleisen for detailed comments on this paper.

References

[1] Baker, Henry G., Jr., "Shallow Binding in Lisp 1.5," *CACM* **21**, July 1978, pages 565–569.

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

[3] Friedman, Daniel P., Christopher T. Haynes, and Eugene Kohlbecker, "Programming with Continuations," *Program Transformation and Programming Environments*, ed. P. Pepper, Springer-Verlag, 1984, pages 263–274.

[4] Friedman, Daniel P., Christopher T. Haynes, Eugene Kohlbecker, and Mitchell Wand. "The Scheme 84 Reference Manual," Indiana University Computer Science Department Technical Report No. 153, May, 1984.

[5] Hanson, Chris, and John Lamping, "Dynamic Binding in Scheme," unpublished manuscript, 1984.

[6] Haynes, Christopher T., and Daniel P. Friedman, "Engines build process abstractions," *Conf. Rec. of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984, pages 18–24.

[7] Haynes, Christopher T., Daniel P. Friedman, and Mitchell Wand "Continuations and Coroutines," *Conf. Rec. of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984, pages 293–298.

[8] Hewitt, Carl, "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.

[9] Landin, Peter, "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] Rees, Jonathan A., and Norman I. Adams IV, "T: A dialect of Lisp or, LAMBDA: The ultimate software tool," *Conf. Rec. of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122.

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

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

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

[14] Steele, Guy Lewis, Jr., "Macaroni is better than spaghetti," Conf. Rec. of the Symposium on Artificial Intelligence and Programming Languages, *SIGPLAN Notices 12, 8 and SIGART Newsletter 64*, August, 1977, pages 60–66.

[15] Steele, Guy L., *Common Lisp: The Language*, Digital Press, Bedford MA, 1984.

[16] Sussman, Gerald Jay, and Drew Vincent McDermott, "From PLANNER to CONNIVER—A genetic approach", *Proceedings of Joint Computer Conference 41, part II*, AFIPS Press, NJ, (1973) pages 1171–1179.

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

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

```
(define! call/cc*
   (lambda (f)
      (let ([k '*])
         (letrec ([loop (lambda (f)
                          (let ([v (call/cc-one-shot
                                     (lambda (k2)
                                        (set! k k2)
                                        (f (lambda (x) (k x)))))])
                            (call/cc-one-shot
                              (lambda (return)
                                 (loop (lambda (k) (return v)))))))])
            (loop f)))))
```

Figure 4. `call/cc*`

Appendix

Claim: `call/cc*` *is equivalent to* `call/cc`.

See Figure 4. When `call/cc*` is invoked with a function `f`, a local binding for `k` is created and `loop` is invoked with `f`. The `call/cc-one-shot` on the right-hand side of the `let` pair is then invoked, resulting in its "one shot" continuation being bound to `k2` and saved in `k`. `f` is then passed the closure `(lambda (x) (k x))`, which is the cob of `call/cc*`.

If `f` subsequently invokes the cob with a value x, continuation `k` is invoked with x. Recall that this is the continuation of the first `call/cc-one-shot` expression, the value of which is bound to `v`, so that x is bound to `v` and the second `call/cc-one-shot` expression is evaluated. The continuation of this expression is the continuation of the original invocation of `call/cc*`, and is now bound to `return`. `loop` is then invoked with a function that, when passed a cob, will ignore it and invoke `return` with `v`. The first `call/cc-one-shot` is now invoked again, obtaining a new continuation that is identical to the continuation recorded in `k` (which was invoked following the cob invocation). This new continuation is then saved in `k`, replacing the one that was consumed. The cob is then passed to the closure `(lambda (k) (return v))`, which ignores it and passes `v` to the `return` continuation. `return` is the

continuation of `call/cc*`, and `v` is passed to the cob, so the proper result has been obtained upon the first cob invocation. It is `return` that allows the `call/cc*`'s continuation to be invoked without implicit or explicit invocation of the new `k` continuation.

Each subsequent invocation of the cob results in the invocation value being bound to `v`, and the sequence of actions described above being repeated. With each cob invocation two continuations are invoked—`k` and `return`. However, with each invocation both `call/cc-one-shot` expressions are evaluated, resulting in unused continuations being recorded in `k` and `return`. Thus the `call/cc-one-shot` restriction that its continuations be invoked only once is obeyed.