Unwind-protect in portable Scheme

Dorai Sitaram
Verizon
40 Sylvan Road
Waltham, MA 02451

Abstract

Programming languages that allow non-local control jumps also need to provide an *unwind-protect* facility. Unwind-protect associates a *postlude* with a given block *B* of code, and guarantees that the postlude will always be executed regardless of whether *B* concludes normally or is exited by a jump. This facility is routinely provided in all languages with *first*-order control operators. Unfortunately, in languages such as Scheme and ML with *higher*-order control, unwind-protect does not have a clear meaning, although the need for some form of protection continues to exist. We will explore the problem of specifying and implementing unwind-protect in the higher-order control scenario of Scheme.

1 Introduction

Unwind-protect has a straightforward semantics for programming languages where non-local control jumps are purely first-order, i.e., computations can abort to a dynamically enclosing context, but can never re-enter an already exited context. Such languages include Common Lisp, Java, C++, and even text-editor languages like Emacs and Vim; all of them provide unwind-protect. An unwind-protected block of code *B* has a postlude *P* that is guaranteed to run whenever and however *B* exits, whether normally or via a non-local exit to some enclosing dynamic context. This is a useful guarantee to have, as we can have *P* encode *clean-up* actions that we can rely upon to happen. The canonical use for unwind-protect is to ensure that file ports opened in *B* get closed when *B* is exited.

```
(let ([o #f])
  (unwind-protect
    ;protected code
    (begin
      (set! o (open-output-file "file"))
      ... <possible non-local exit> ...
      )
    ;the postlude
    (close-output-port o)))
```


When we move to higher-order control scenarios such as Scheme [8] and ML [1], it is no longer clear what unwind-protect should mean. Here, control *can* re-enter a previously exited dynamic context, opening up new questions such as:

1. Should a prelude be considered in addition to the postlude?
2. Can the postlude be evaluated more than once?
3. Should the postlude be enabled only for some exits but not for others, and if so which?

The language Scheme provides a related operator called *dynamic-wind* that attaches a prelude and a postlude to a code block, and ensures that the postlude (prelude) is always evaluated whenever control exits (enters) the block. While this may seem like a natural extension of the first-order unwind-protect to a higher-order control scenario, it does not tackle the pragmatic need that unwind-protect addresses, namely, the need to ensure that a kind of "clean-up" happens only for those jumps that *significantly exit* the block, and not for those that are *minor excursions*. The crux is identifying which of these two categories a jump falls into, and perhaps allowing the user a way to explicitly fix the category. It usually makes no sense to re-enter a block after the clean-up has been performed (as in the port-closing example above): Thus there is no need for a specific prelude syntax beyond sequencing, and postludes need happen only once. Thus we can answer questions 1 and 2 above with No, but there is no single objectively correct answer to question 3.

2 Call/cc and how to constrain it

Scheme's control operator, *call-with-current-continuation* (abbreviated *call/cc*), allows control to transfer to arbitrary points in the program, not just to dynamically enclosing contexts. It does so by providing the user with a *continuation*, i.e., a procedural representation of the current control context, or more simply, "the rest of the program". Invoking this continuation at any point in the program causes that point's current context to be replaced by the context that the continuation represents. The user sees *call/cc* as a procedure that takes a single unary procedure f as argument. f is called with the current continuation (hence the operator's name). This continuation is a procedure that takes a single argument, which it inserts into the old program context.[1]

This ability to substitute the current program context by a previously captured snapshot of a program context is simple and powerful [6, 7, 10], but too low-level to be used straightaway for user-level

[1] I will ignore the presence of Scheme's multiple values, as they add no particular illumination to the problem we are addressing.

abstractions. In addition to the difficulty of encoding user abstractions in terms of *call/cc*, one must also ensure that the abstractions so defined can function without interference from other uses of *call/cc*. To solve this problem, Friedman and Haynes [5] illustrate a technique for *constraining* raw *call/cc*. They define new *call/cc* operators that call the original *call/cc*, but instead of directly calling the *call/cc*-argument on the continuation, they call it on a *continuation object* or *cob*, which is a procedure that performs whatever additional constraining tasks are required before calling the actual continuation.

Let us illustrate the cob technique to solve an easily described problem, that of fluid variables. The form **let-fluid** temporarily extends the fluid environment, **fluid-env**, for the duration of its dynamic extent.

```
(define *fluid-env* '())

(define-syntax let-fluid
  (syntax-rules ()
    [(let-fluid ((x e) ...) b ...)
     (let ([old-fluid-env *fluid-env*])
       (set! *fluid-env*
         (append! (list (cons 'x e) ...) *fluid-env*))
       (let ([result (begin b ...)])
         (set! *fluid-env* old-fluid-env)
         result))])))
```

Fluid variables are accessed using the form **fluid**, defined globally using **define-fluid**, and side-effected using **set-fluid!**:

```
(define-syntax fluid
  (syntax-rules ()
    [(fluid x)
     (cond [(assq 'x *fluid-env*) => cdr]
           [else (error 'undefined-fluid 'x)])]))

(define-syntax define-fluid
  (syntax-rules ()
    [(define-fluid x e)
     (set! *fluid-env*
       (cons (cons 'x e) *fluid-env*))]))

(define-syntax set-fluid!
  (syntax-rules ()
    [(set-fluid! x e)
     (cond [(assq 'x *fluid-env*)
            => (lambda (c)
                 (set-cdr! c e))]
           [else (error 'undefined-fluid 'x)])]))
```

This definition fails in the presence of *call/cc*, because a call to a continuation does not restore the fluid environment to the value it had at the capture of the continuation. A simple cob-based rewrite of *call/cc* takes care of this:

```
(define call/cc-f
  (let ([call/cc-orig call/cc])
    (lambda (proc)
      (call/cc-orig
        (lambda (k)
          (let* ([my-fluid-env *fluid-env*]
                 [cob (lambda (v)
                        (set! *fluid-env* my-fluid-env)
                        (k v))])
            (proc cob)))))))
```

Note that once we've defined the new fluids-aware *call/cc* variant, it's a good idea to reuse the *call/cc* name to refer to the variant. In essence, we retire the original *call/cc* from further use, as it would interfere with the correct functioning of the new operator. In standard Scheme, one could do this by simply re-**set!**ing the *call/cc* name, but this has problems as programs scale. In a Scheme with a module system [4], a more robust method is to define a new module that provides the new control operator under the *call/cc* name.

3 Unwind-protect and cobs

Friedman and Haynes already use cobs to tackle the problem of defining an unwind-protect (and the corresponding *call/cc* variant) for Scheme, and observe that there is a choice of meaningful unwind-protect semantics — the choice as they see it lying in the method of identifying which postludes to perform based purely on their position relative to the continuation nodes in the control tree.

However, automatic detection of the relevant postludes does not necessarily match user expectations. Sometimes it may be more suitable to allow the user explicitly specify which continuations, or continuation calls, ought to trigger unwind-protect postludes, as Kent Pitman [9] proposes. He suggests that *call/cc* as provided in the Scheme standard may be fundamentally misdesigned as it thwarts the creation of a pragmatic unwind-protect facility, and that it be replaced by one of two *call/cc*-like operators that he describes as more unwind-protect-friendly, while still providing full continuations.

Fortunately, the cob technique can implement both the Pitman variants, as we shall show in sections 4 and 5. Thus, at least as far as unwind-protect is concerned, Scheme's design does not pose a disadvantage. Indeed, given the variety of unwind-protect styles that are possible for Scheme (it's unlikely that the Friedman–Haynes and Pitman styles exhaust this list), learning the cob technique as a reliable and flexible way to implement the styles may be a better approach than enshrining one of the styles permanently in the standard.

3.1 call/cc-e

Pitman's first *call/cc* variant, which we will call *call/cc-e*, takes an additional argument whose boolean value decides if the captured continuation should be an *escaping* or a *full* continuation. Escaping continuations cannot be used to re-enter an already exited dynamic context, whereas full continuations have no such limitation. Thus, in the expressions:

```
(call/cc-e #t M)
(call/cc-e #f N)
```

M is called with an escaping continuation, whereas *N* is called with a full continuation.

In a Scheme with *call/cc-e*, for the expression (**unwind-protect** *B P*), the postlude *P* is run only if (1) *B* exits normally, or (2) *B* calls an escaping continuation that was captured outside the **unwind-protect** expression's dynamic extent.

3.2 call/cc-l

Pitman's second *call/cc* variant, which we will call *call/cc-l*, produces continuations which take an additional argument that decides

if that continuation call is to be the *last use* of that continuation. Thus, when evaluating the following expressions:

```
(define r 'to-be-set-below)

(call/cc-l
   (lambda (k)
      (set! r k)))

(r #f 'first-use-works)
(r #f 'second-use-works)
(r #t 'third-use-works-and-is-last-use)
(r #f 'fourth-attempted-use-will-not-work)
```

the fourth attempted use of the continuation *r* will error.

In a Scheme with *call/cc-l*, for the expression (**unwind-protect** *B P*), the postlude *P* is run only if (1) *B* exits normally, or (2) *B* calls a continuation for the (user-specified) last time, and that continuation does not represent a context that is dynamically enclosed by the **unwind-protect** expression.

In short, for *call/cc-e*, postludes are triggered only by continuations specified by the user to be escaping; and for *call/cc-l*, they are triggered only by continuation calls specified by the user to be their last use.

We will now use the cob technique to define each of these *call/cc* variants, and its corresponding unwind-protect, from *call/cc-f* (which we defined in section 2 from the primitive *call/cc*, also using a cob).

4 Unwind-protect that recognizes only escaping continuations

call/cc-e (unlike the standard *call/cc*) takes two arguments: The second argument is the procedure that is applied to the current continuation. Whether this continuation is an *escaping* or a *full* continuation depends on whether the first argument is true or false.

We implement *call/cc-e* by applying its second argument (the procedure) to a cob created using *call/cc-f*. The cobs created for *call/cc-e* #t (escaping continuations) and *call/cc-e* #f (full continuations) are different.

unwind-protect interacts with *call/cc-e* as follows: If the body is exited by a escaping continuation provided by a dynamically enclosing *call/cc-e* #t, then the postlude is performed. The postlude is *not* performed by full continuations or by escaping continuations created by a *call/cc-e* #t within the **unwind-protect**. To accomplish this, the cob generated by *call/cc-e* #t keeps a list (*my-postludes*) of all the postludes within its dynamic extent. Since a call to *call/cc-e* #t cannot know of the **unwind-protect**s that will be called in its dynamic extent, it is the job of each **unwind-protect** to update the *my-postludes* of its enclosing *call/cc-e* #ts. To allow the **unwind-protect** to access its enclosing *call/cc-e* #t, the latter records its cob in a fluid variable **curr-call/cc-cob**.

The following is the entire code for *call/cc-e* and its *unwind-protect-proc*, a procedural form of unwind-protect:

```
(define call/cc-e #f)
(define unwind-protect-proc #f)
(define-fluid ∗curr-call/cc-cob∗
   (lambda (v) (lambda (x) #f)))
(define-fluid ∗curr-u-p-alive?∗ (lambda () #t))

(let ([update (list 'update)]
      [delete (list 'delete)])

   (set! call/cc-e
      (lambda (once? proc)
         (if once?
            (call/cc-f
               (lambda (k)
                  (let∗
                     ([cob (fluid ∗curr-call/cc-cob∗)]
                      [my-postludes '()]
                      [already-used? #f]
                      [cob
                         (lambda (v)
                            (cond
                               [(eq? v update)
                                (lambda (pl)
                                   (set! my-postludes
                                      (cons pl my-postludes))
                                   ((cob update) pl))]
                               [(eq? v delete)
                                (lambda (pl)
                                   (set! my-postludes
                                      (delq! pl my-postludes))
                                   ((cob delete) pl))]
                               [already-used?
                                  (error 'dead-continuation)]
                               [else
                                  (set! already-used? #t)
                                  (for-each
                                     (lambda (pl) (pl))
                                     my-postludes)
                                  (k v)]))])
                     (let-fluid ([∗curr-call/cc-cob∗ cob])
                        (cob (proc cob))))))
               (call/cc-f
                  (lambda (k)
                     (let∗
                        ([my-u-p-alive? (fluid ∗curr-u-p-alive?∗)]
                         [cob
                            (lambda (v)
                               (if (my-u-p-alive?)
                                  (k v)
                                  (error 'dead-unwind-protect)))])
                        (cob (proc cob))))))))

   (set! unwind-protect-proc
      (lambda (body postlude)
         (let ([curr-call/cc-cob (fluid ∗curr-call/cc-cob∗)]
               [alive? #t])
            (let-fluid ([∗curr-u-p-alive?∗ (lambda () alive?)])
               (letrec ([pl (lambda ()
                              (set! alive? #f)
                              (postlude)
                              ((curr-call/cc-cob delete) pl))])
                  ((curr-call/cc-cob update) pl)
                  (let ([res (body)])
                     (pl)
                     res)))))))
```

As we can see, the cob employed by *call/cc-e* #t (i.e., the part of the *call/cc-e* body that is active when its *once?* argument is true) is fairly involved. This is because, in addition to performing the jump, it has to respond to *update* and *delete* messages pertaining to its *my-postludes*. We have defined lexical variables *delete* and *update* so they are guaranteed to be different from any user values given to the cob. The cob also remembers its nearest enclosing cob (*prev-cob*), so that the *update* and *delete* messages can be propagated outward. (This is because *any* of the escaping continuations enclosing an **unwind-protect** can trigger the latter's postlude.) When the cob is called with a non-message, it performs all of its *my-postludes*, before calling its embedded continuation. It also remembers to set a local flag *already-used?*, because it is an error to call an escaping continuation more than once.[2]

The *call/cc-e* #f part, the one that produces full continuations, is fairly simple. Its cob simply remembers if its enclosing **unwind-protect** is alive, via the fluid variable **curr-u-p-alive?**. This is to prevent entry into an **unwind-protect** body that is known to have exited.

The corresponding *unwind-protect-proc* notes its nearest enclosing *call/cc-e* #t's cob, to let it know of its postlude. It also adds wrapper code to the postlude so that the latter can delete itself when it is done, and flag the **unwind-protect** as no longer alive. The body and the wrapped postlude are performed in sequence, with the body's result being returned.

The macro **unwind-protect** is defined as follows:

```
(define-syntax unwind-protect
  (syntax-rules ()
    [(unwind-protect body postlude)
     (unwind-protect-proc
       (lambda () body) (lambda () postlude))]))
```

The helper procedure *delq!* is used to delete a postlude from a list:

```
(define delq!
  (lambda (x s)
    (let loop ([s s])
      (cond [(null? s) s]
            [(eq? (car s) x) (loop (cdr s))]
            [else (set-cdr! s (loop (cdr s)))
                  s]))))
```

5 Unwind-protect that recognizes only last-use continuations

call/cc-l takes a single procedure argument (just like the standard *call/cc*), but the continuation it captures takes two arguments: The first argument, if true, marks that call as the *last use* of the continuation. The second argument is the usual transfer value of the continuation.

[2] Pitman's text calls the escaping continuations *single-use*, counting as a use the implicit use of the continuation (i.e., when the *call/cc-e* expression exits normally without explicitly calling its continuation). There are some design choices on what effect the use of such a continuation has on the use count of other continuations captured within its dynamic extent, whether they be single- or multi-use. For now, I assume there is no effect. If there were, such could be programmed by having the cob propagate kill messages to its nearest enclosed (not enclosing!) cob using fluid variables.

The corresponding unwind-protect's postlude is triggered by a continuation only on its last use.

call/cc-l, like *call/cc-e*, is implemented with a cob. (Unlike *call/cc-e*, *call/cc-l* does not create two types of continuations, so it doesn't need two types of cobs.) The *call/cc-l* cob looks very much like the union of the cobs for *call/cc-e*, except of course that whereas the *call/cc-e* triggers postludes for escaping continuations, the *call/cc-l* cob triggers them for continuations on their last use. Another difference is that the *call/cc-l* cob takes *two* arguments, like the user continuation it stands for. We use the cob's first argument for the message, which can be *update* and *delete* for manipulating the postludes, #f for marking non-last use, and any other value for last use.

As in the *call/cc-e* case, the cob is available as the fluid variable **curr-call/cc-cob** to an enclosed **unwind-protect**; and **unwind-protect** has a fluid variable **curr-u-p-alive?** so continuations can check it to avoid re-entering an exited **unwind-protect**. But we also associate another fluid variable with **unwind-protect**, viz., **curr-u-p-local-conts** — this is to keep track of continuations that were captured within the **unwind-protect**, for we view the call of a continuation whose capture and invocation are both local to the **unwind-protect** as non-exiting, and thus not worthy of triggering the postlude, even if it happens to be last-use. Each *call/cc-l* updates its enclosing **curr-u-p-local-conts**, and its cob's last call checks its current **curr-u-p-local-conts** before triggering postludes.

```
(define call/cc-l #f)

(define-fluid ∗curr-call/cc-cob∗ (lambda (b v) #f))
(define-fluid ∗curr-u-p-local-conts∗ '())
```

The following replaces (**set!** *call/cc-e* ...) in the code in section 4:

```
(set! call/cc-l
  (lambda (proc)
    (call/cc-f
      (lambda (k)
        (set-fluid! ∗curr-u-p-local-conts∗
          (cons k (fluid ∗curr-u-p-local-conts∗)))
        (let∗
          ([prev-cob (fluid ∗curr-call/cc-cob∗)]
           [my-u-p-alive? (fluid ∗curr-u-p-alive?∗)]
           [my-postludes '()]
           [already-used? #f]
           [cob
             (lambda (msg v)
               (cond
                 [(eq? msg update)
                  (set! my-postludes (cons v my-postludes))
                  (prev-cob update v)]
                 [(eq? msg delete)
                  (set! my-postludes (delq! v my-postludes))
                  (prev-cob delete v)]
                 [already-used?
                   (error 'dead-continuation)]
                 [(not (my-u-p-alive?))
                  (error 'dead-unwind-protect)]
                 [msg
                   (set! already-used? #t)
                   (if (not
                         (memq
                           k
                           (fluid ∗curr-u-p-local-conts∗)))
```

```
                    (for-each (lambda (pl) (pl))
                              my-postludes))
                (k v)]
              [else (k v)])])
        (let-fluid ([*curr-call/cc-cob* cob])
          (cob #f (proc cob))))))))
```

The following replaces (**set!** *unwind-protect-proc* ...) in the code in section 4:

```
(set! unwind-protect-proc
  (lambda (body postlude)
    (let ([curr-call/cc-cob (fluid *curr-call/cc-cob*)]
          [alive? #t])
      (let-fluid ([*curr-u-p-alive?* (lambda () alive?)]
                  [*curr-u-p-local-conts* '()])
        (letrec ([pl (lambda ()
                       (set! alive? #f)
                       (postlude)
                       (curr-call/cc-cob delete pl))])
          (curr-call/cc-cob update pl)
          (let ([res (body)])
            (pl)
            res))))))
```

The only significant difference between this *unwind-protect-proc* and the one in section 4 is that it initializes the fluid variable **curr-u-p-local-conts**, which dynamically enclosed calls to *call/cc-l* can update.

6 Conclusion

We see that the Friedman-Haynes cob technique for deriving constrained forms of *call/cc* is a reliable way to implement various forms of unwind-protect — both the Pitman-style ones that rely on explicit user annotation, and the Friedman-Haynes ones that depend on the relative positions of unwind-protects and continuations on the control tree.

The cob technique is not the only way to derive unwind-protect — for an ingenious way to derive *call/cc-e* using Scheme's built-in *dynamic-wind*, see Clinger [2]. However, the cob approach remains a predictable workhorse for systematically experimenting with new styles and modifying existing styles. This kind of flexibility is especially valuable for unwind-protect since the latter cannot have a canonical, once-and-for-all specification in Scheme, making it important to allow for multiple library solutions.

7 Acknowledgments

I thank Matthias Felleisen and Robert Bruce Findler for helpful discussions.

8 References

[1] Bruce F. Duba, Robert Harper, and Dave MacQueen, "Typing First-Class Continuations in ML", in *18th ACM Symp. on Principles of Programming Languages*, pp. 163–173, 1991.

[2] William Clinger, `http://www.ccs.neu.edu/home/will/Temp/uwesc.sch`.

[3] Matthias Felleisen, "On the Expressive Power of Programming Languages", in *European Symp. on Programming*, 1990, pp. 134–151.

[4] Matthew Flatt, "Composable and Compilable Macros: You Want It *When?*", in *International Conf. on Functional Programming*, 2002.

[5] Daniel P. Friedman and Christopher T. Haynes, "Constraining Control", in *12th ACM Symp. on Principles of Programming Languages*, 1985, pp. 245–254.

[6] Christopher T. Haynes and Daniel P. Friedman, "Engines Build Process Abstractions", in *ACM Symp. on Lisp and Functional Programming*, 1984, pp. 18–24.

[7] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand, "Continuations and Coroutines", in *ACM Symp. Lisp and Functional Programming*, 1984, pp. 293–298.

[8] Richard Kelsey and William Clinger (eds.), *Revised^5 Report on the Algorithmic Language Scheme ("R5RS")*, `http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs.html`, 1998.

[9] Kent Pitman, *UNWIND-PROTECT vs Continuations*, `http://www.nhplace.com/kent/PFAQ/unwind-protect-vs-continuatins.html`.

[10] Dorai Sitaram, *Teach Yourself Scheme in Fixnum Days*, `http://www.ccs.neu.edu/~dorai/t-y-scheme/t-y-scheme.html`.