# Handling Control

Dorai Sitaram
Department of Computer Science
Rice University
Houston, TX 77251-1892

To appear in: *ACM SIGPLAN '93 Conf. on Programming Language Design & Implementation*
Albuquerque, N.M., June 21-25, 1993

## Abstract

Non-local control transfer and exception handling have a long tradition in higher-order programming languages such as Common Lisp, Scheme and ML. However, each language stops short of providing a full and complementary approach — control handling is provided *only* if the corresponding control operator is first-order. In this work, we describe handlers in a higher-order control setting. We invoke our earlier theoretical result that all denotational models of control languages invariably include capabilities that handle control. These capabilities, when incorporated into the language, form an elegant and powerful higher-order generalization of the first-order exception-handling mechanism.

## 1 Introduction

Control manipulation in applicative programming languages comes in two flavors. *First-order* control operators allow computations to abort to a dynamically enclosing control context, e.g., Common Lisp's [23, 24] **throw** and ML's [9, 17] **raise**. They are invariably accompanied by forms that delimit and handle the aborted value, e.g., **catch** in Common Lisp and **handle** in ML. In contrast, *higher-order* operators such as *call-with-current-continuation* in Scheme [27, 1] and ML [4] allow unrestricted transfers of control without regard to dynamic scope.

In pre-Common Lisp [16], the operators *error* and **errorset**, intended to respectively signal and handle errors, work equally well for exits. The form **errorset** simply returns the value of its subexpression if the latter has no calls to *error*. If the subexpression does generate a call to *error* — whether due to a miscomputation or an explicit call to *error* — there is a non-local exit or abort to **errorset**. Using lists or other records to pack the error return value and placing a dispatching wrapper around **errorset** provides a rudimentary but effective form of control handling.

We next have the **catch** and **throw** pair of Common Lisp, where non-local exits are caused ("thrown") by **throw** and delimited ("caught") by **catch**. These operators are *tagged*, i.e., a tagged **throw** can only be caught by a **catch** with an identical tag. In other words, a **throw** can pick its destination, and not restrict itself to the *closest* **catch**. (In contrast, *error* and **errorset** saddle the user with the chore of writing special dispatching routines to distinguish between different exit destinations.) Because of the explicit **throw** operator, there is no reliance on errors for obtaining jumps. In fact, it is possible to view *error* as a specially tagged **throw**, and **errorset** as its corresponding **catch**. ML's exception-handling system, where **raise** causes a first-order jump and **handle** delimits it, matches this view.

In contrast to the first-order operators described above, a higher-order control operator such as Scheme's and ML's *call-with-current-continuation*[1] can transfer control to arbitrary points in the program, not just to dynamically enclosing contexts. Like its historical forerunners **J** [15] and **escape** [19], *call/cc* provides the user with a representation of the current control context: the "rest of the program" or the "continuation". Invoking this continuation at any point in the program causes the program's current context to be replaced by the continuation's context. This ability to substitute the current program context by a previously stored snapshot of a program context is simple and powerful. It allows a wide range of programming paradigms [10, 11, 12, 13] not possible with **catch** and **throw**.

However, there is no analog to *delimiting* or *handling* a control action, as with **errorset**, or to distinguishing between different varieties of control actions, as with **catch**. Methods of handling and distinguishing control actions are left to user programs. Typically,

[1] Abbreviated *call/cc* in Scheme and *callcc* in ML.

the user stores the context where a continuation should be handled as yet another *call/cc*-continuation, so that control can be transferred to it after the jump to the first continuation has accomplished its purpose. In the presence of several continuations with their respective quasi-handlers, keeping track of the various jump-off points and avoiding clashes between them requires sophisticated bookkeeping strategies [5, 12]. It is therefore useful to explore options that tackle this problem *without* sacrificing the programming power of higher-order control.

Here we show that the historical duality of first-order throwing and handling is useful even for higher-order control. In earlier work [21, 22], we showed that *all* conventional models for non-local control include a control-handling capability. In other words, if the relationship between the model meanings and the observable behavior of language terms is to match, the language, like the model, must include handlers. In light of this theoretical result, efforts to "constrain" *call/cc* are simply attempts to simulate a handler in a handler-less language. Such attempts are not only complicated but also ultimately unsatisfactory, since the original operator has to be disabled in the process. A control handler in the language cleanly solves these issues. Indeed, it enriches higher-order control, opening the way to novel and elegant control paradigms.

Section 2 introduces the higher-order control operators *run* and *fcontrol* in a Scheme setting, with simple illustrations. Sections 3 and 4 describe two familiar but larger examples where control handlers prove useful. Section 5 summarizes the results.

## 2 Manipulating control using handlers

A control operator such as *call/cc* that captures continuations is a *control reifier*: it *reifies* the continuation of the program and provides this to the user. However, a closer look shows that *call/cc* combines *two* actions: not only does it capture the current continuation, it also invokes its argument procedure on this continuation. In other words, the *handling* of the continuation takes place at the identical site as the creation of the continuation, in contrast to **errorset**/*error* and **catch**/**throw**.

Control-handling constructs in traditional Lisp *delimit* the context that can be erased by their control operators. Extrapolating from the relationship between **errorset** and *error* or **catch** and **throw**, a control delimiter for *call/cc* would control the extent of the context captured by *call/cc* or erased by its continuations. This operator, proposed by Felleisen [8], is called the "prompt", since it annotates its subexpression as an independent program, *in so far as control actions are concerned*, much like the prompt sign in a read-eval-print loop. The *procedural* variant of the prompt is called *run*, to borrow a term used for an operator that runs programs [26]. The prompt and *run* are equivalent: either can be seen as syntactic sugar for the other. Together with higher-order control reifiers like *call/cc*, the prompt supports powerful programming idioms [7, 20]. It has several successors specially suited to various practical settings, e.g., *spawn* [14], **reset** [3], and *splitter* [18]. However, none of these constructs *handles* control objects — the corresponding control reifier continues to double as handler.

In this work, we continue the process of extrapolation identified above by adding control-handling capabilities to the delimiter. In other words, we shift the site of continuation handling from the control reifier to the control delimiter. This drastically changes the aspect of both delimiter and reifier. The new control-capturing operator is a stripped down version of *call/cc* — it needs no procedural argument to "receive" its continuation, since the delimiter takes care of control handling — and is therefore given a new name: *fcontrol*. The new delimiter takes two subexpressions: (1) a computation that runs as a control-independent program, and (2) a procedure that will handle any control actions performed by the first subexpression.[2]

There is one notable difference between the current system and the historical **errorset** that it resembles: It is a much more versatile control mechanism — the continuations manipulated are higher-order, and not just aborts. The control system is identical to the one suggested by a different, theoretical route, viz., the control-handling prompts that we showed to be implicitly present in all the traditional denotational models [21, 22].

### 2.1 *Run* and *fcontrol*

The control delimiter is called *run*. It takes two arguments, a thunk[3] and a binary procedure called the handler:

(*run* $\langle$*thunk*$\rangle$ $\langle$*handler*$\rangle$)

The procedure *run* calls the thunk as an control-independent program. If the thunk returns normally, the call to *run* returns the result of the thunk. If, on the other hand, there is a control action inside the thunk, the handler is invoked on the objects produced by the control action.

The prompt, %, is a convenient syntactic variant[4] of *run*:

[2] Bruce Duba first suggested the "prompt with a handler".

[3] I.e., a procedure of zero arguments.

[4] The symbol % is chosen for its similarity to an operating system prompt. Lisp's own prompt sign is usually >; unfortunately, that symbol is taken.

(% *exp handler*) ≡ (*run* (**lambda** () *exp*) *handler*)

A control action is caused by invoking the control reifer *fcontrol* on a single argument:

(*fcontrol* ⟨*object*⟩)

This sends a signal to the dynamically nearest surrounding *run*, much like the first-order **throw** to **catch**. The important difference is that the signal contains both the argument object *and the reified context or the continuation*. *Run* processes this signal by invoking the handler on these two values. Since we want *run* to be the sole arbiter of control handling, the continuations produced by *fcontrol* are "functional". I.e., *fcontrol*-continuations, unlike *call/cc*-continuations, will not automatically erase existing context when invoked.

### 2.2 Simple exits

As a simple illustration, a prompt with a handler that ignores the continuation provides an *abort*, i.e., the common paradigm for procedure and loop exits. The prompt marks the entry point; an *fcontrol*-application within the prompt's first subexpression exits to the entry point with an aborted value. E.g., the following procedure for multiplying the elements of a list exits immediately on encountering a zero element:

```
(define product
  (lambda (s)
    (% (let loop ([s s])
         (if (null? s) 1
           (let ([a (car s)])
             (if (= a 0) (fcontrol 0)
               (* a (loop (cdr s)))))))
       (lambda (r k) r))))
```

### 2.3 Tree-matching

A canonical example of the use of continuations is to find if two trees have the same *fringe*[5]. The purely functional approach flattens both trees and checks if the results match. However, this would traverse the trees once completely to flatten them, and then again till it finds non-matching elements. Furthermore, even the best flattening operations require *cons*es equal to the total number of leaves.

The Scheme solution enlists both *call/cc* and assignment to avoid needless *cons*ing. Each tree is mapped to a *generator*, a procedure with internal state that successively produces the leaves of the tree:

[5] In our example ((1 . 2) . 3) and (1 . (2 . 3)) are considered to have the same fringe, as also ((1 2) 3), (1 (2 3)) and ((1 2) (3)) — the empty list (), wherever it occurs in the tree, does not contribute any leaves.

```
(define make-generator
  (lambda (tree)
    (letrec
      ([caller '*]
       [generate-leaves
        (lambda ()
          (let loop ([tree tree])
            (cond
              [(pair? tree)
               (loop (car tree)) (loop (cdr tree))]
              [(null? tree) 'skip]
              [else
                (call/cc
                  (lambda (rest-of-tree)
                    (set! generate-leaves
                      (lambda () (rest-of-tree '*)))
                    (caller tree)))]))
          (caller '()))])
      (lambda ()
        (call/cc
          (lambda (k)
            (set! caller k) (generate-leaves)))))))
```

The generator returns the empty list (which cannot be a leaf) when all the leaves have been accounted for. A simple loop alternately calls each generator, matches the leaves thus obtained, and stops immediately upon finding a mismatch:

```
(define same-fringe?
  (lambda (tree1 tree2)
    (let ([gen1 (make-generator tree1)]
          [gen2 (make-generator tree2)])
      (let loop ()
        (let ([leaf1 (gen1)] [leaf2 (gen2)])
          (if (eqv? leaf1 leaf2)
              (if (null? leaf1) #t (loop))
              #f))))))
```

The generator procedure uses *call/cc* to keep track of two continuations: (1) the continuation of each call to the generator so the result can be returned to it, and (2) the continuation marking each break in the traversal of the tree, so that the next call to the generator can resume where the previous call left off. Assignment is used to store both continuations in the internal state of the generator.

The crucial continuation is (2), the rest of the computation in the generator. The continuation (1) merely *handles* the interface with the generator. In the *call/cc* solution, each continuation represents a different instance of the *entire* program context. In fact, continuation (1) is used to remember that point in the continuation (2) where control needs to be transferred back to the caller. In the presence of the continuation-delimiting handler, continuation (1) need not be captured at all, and furthermore, continuation (2) need only be the *partial* continuation within the generator. As a side benefit, the entire bookkeeping using assignment can be wholly avoided.

We now present the Scheme solution that uses prompt and *fcontrol* rather than *call/cc* and **set!**. Here too, the program checks the leaves alternately, using generators that successively *throw* leaves:

```
(define make-fringe
  (lambda (tree)
    (lambda (any)
      (let loop ([tree tree])
        (cond [(pair? tree)
               (loop (car tree)) (loop (cdr tree))]
              [(null? tree) '*]
              [else (fcontrol tree)]))
      (fcontrol '()))))
```

A loop *catches* leaves alternately from each fringe, and compares them: a mismatch immediately stops the process:

```
(define same-fringe?
  (lambda (tree1 tree2)
    (let loop ([fringe1 (make-fringe tree1)]
               [fringe2 (make-fringe tree2)])
      (% (fringe1 '*)
         (lambda (leaf1 rest-of-fringe1)
           (% (fringe2 '*)
              (lambda (leaf2 rest-of-fringe2)
                (if (eqv? leaf1 leaf2)
                    (if (null? leaf1) #t
                        (loop rest-of-fringe1
                              rest-of-fringe2))
                    #f))))))))
```

Each time the rest of a fringe is probed, a handler is used to collect a leaf (or the empty list signaling end of fringe) and the remaining fringe computation. If the leaves from the two fringes match, more leaves are ordered. If the leaves are different, the rest of the fringes are ignored, and the predicate returns false.

### 2.4 Tagged *run* and *fcontrol*

To avoid interference between control actions arising from logically different uses of *run/fcontrol*, we should identify matching pairs of these control operators. In an earlier approach, we suggested a hierarchically ordered set of delimiters [20]. For prompts with handlers, it is natural to continue our extrapolation from Lisp's **catch** and **throw**, giving *tagged* versions of *run* and *fcontrol*, invoked respectively as:

(*run-tagged* ⟨*tag*⟩ ⟨*thunk*⟩ ⟨*handler*⟩)

and

(*fcontrol-tagged* ⟨*tag*⟩ ⟨*object*⟩)

One tagging protocol — others are possible — is to have an *fcontrol* tagged **X** jump to the dynamically closest prompt tagged **X**. Not only are intervening prompts of other tags ignored, but the continuation thrown to the **X**-prompt will be the complete continuation extending from the **X**-prompt to the **X**-*fcontrol*-application. Different tags govern different logical uses of *run/fcontrol* without fear of interference. Furthermore, since a tag is any object, we can choose unforgeable tag values and hide their use within a textual region using lexical hiding.

We can define the tagged versions using the raw primitives and a strategy whereby *fcontrol*-tagged uses *fcontrol* to send a structure consisting of both its tag and its thrown value. However, it is preferable to avoid the data-structure overhead and provide the tagged operators as primitives. We shall henceforth usurp the name *run* and *fcontrol* for the tagged operators. The previous untagged uses can be considered as having either a default or catch-all tag, say *false*.

## 3 Nestable engines

Our first larger example involving intensive control manipulation is the *engine*. An engine [5, 11] is an abstraction of computation subject to timed preemption. It forms a tractable building block for realizing a variety of communicating concurrent processes.

An engine's underlying computation is a thunk that can be run as a preemptable process. The engine is applied to three arguments: (1) a number of time units or *ticks*, (2) a *success* procedure, and (3) a *failure* procedure. If the engine computation finishes within the allotted time, the *success* procedure is applied to the result of the computation and the remaining ticks; otherwise, the *failure* procedure is applied to a thunk that represents the rest of the interrupted computation. This thunk, when called, resumes the interrupted engine computation.[6]

Haynes and Friedman [11] distinguish two varieties of engines: *flat* (unnestable) and *nestable*. Flat engines cannot run other engines, but as the authors say, this restriction "considerably simplifies the implementation of engines", where the implementation uses Scheme-style continuations.

The more general nestable engines, or *nesters*, can be called at arbitrary sites, but are more difficult to implement in Scheme. An engine that invokes ("nests")

[6] Traditionally, the value supplied to the *failure* procedure is a *new engine* representing the remaining computation of the old engine — rather than just its underlying thunk. Our version is no less general, and further allows enhancements that directly access the engine's underlying thunk.

another engine is called its *parent*. Nesters require some user-specified notion of *fairness* governing the way time is spent among the nested invocations.[7] For instance, the nestable variety described here lets each engine use ticks only from the amount allotted to its ancestors. Otherwise, an engine could "cheat" by performing its work through its offspring.

The *call/cc* implementation of flat engines involves capture of continuations at both the starting (or resuming) and returning points of an engine. Extending it to allow nestable engines entails more than adding code for tick management, since the continuations to be captured while transferring control across the generations of engines need involved bookkeeping [5].

We show here an implementation of nestable engines using control handlers. There is a clean separation between the segment for transferring control and the segment for managing time units.[8] Indeed, modifying just the time management strategy yields different kinds of fairness, including flat engines.

### 3.1 The clock

The implementation presupposes a global clock or interruptable timer that consumes ticks while a program executes. The following describes the type of clock we shall use: it may be defined using either natively provided alarms or through syntactic extensions [5] that simulate tick consumption. The internal state of the clock contains:

1. the number of remaining ticks; and
2. an interrupt handler to be invoked when the clock runs out of ticks.

The user can perform the following clock operations:

1. (*clock* 'set-handler $\langle h \rangle$) sets the interrupt handler to $\langle h \rangle$;
2. (*clock* 'set $\langle n \rangle$) sets the ticks for countdown to $\langle n \rangle$; and
3. (*clock* 'stop) stops the clock (without setting off the interrupt handler), returning the remaining ticks.

The number of ticks ranges over the natural numbers and an atom called infinity.[9] A clock with an infinite number of ticks cannot run out of time, i.e., it is quiescent or "already stopped". Stopping an already stopped clock returns infinity. Setting the clock's ticks to infinity stops the clock, i.e., (*clock* 'stop) is shorthand for (*clock* 'set infinity).

The clock's handler is set to throw an interrupt signal, say 'interrupt, to an engine prompt:

```
(clock 'set-handler
  (lambda () (fcontrol 'engine 'interrupt)))
```

### 3.2 The engine core code

The procedure *make-engine* takes a thunk and produces an engine, a procedure of three arguments: *ticks*, *success* and *failure*.

Assume for the moment that the tick management is accomplished by code segments named $\langle$*ticks-prelude*$\rangle$ and $\langle$*ticks-postlude*$\rangle$. The variable *true-ticks* — introduced in $\langle$*ticks-prelude*$\rangle$ — shows the actual number of ticks given to the current engine. This may be less than the argument *ticks*, owing to fairness considerations.

When invoked, the engine runs its thunk as an independent piece of computation, in so far as control is concerned. We therefore depict the engine computation as the engine's thunk invoked within a prompt tagged 'engine. The computation uses the flag *engine-succeeded?* to record whether the engine succeeded, and if so, the variable *ticks-left* denotes the ticks to spare. In our first outline, the prompt surrounds code that includes both the initial setting of the clock to the allotted ticks, and the stopping of the clock if the thunk returns successfully. If the engine fails — because of a clock interrupt — the handler returns a thunk representing the rest of the engine. (If the handler was invoked for some reason other than an interrupt, we simply let it pass on the value.)

After the postlude timer code $\langle$*ticks-postlude*$\rangle$ — which may modify *ticks-left* — either the *success* or *failure* action is taken, depending on the result of running the engine thunk:

```
(define make-engine;; *** first outline ***
  (lambda (thunk)
    (lambda (ticks success failure)
      ⟨ticks-prelude⟩
      (let* ([engine-succeeded? #f]
             [ticks-left 0]
             [result;; ... (I)
               (% 'engine
                  (begin (clock 'set true-ticks)
                    (let ([result (thunk)])
                      (set! ticks-left (clock 'stop))
                      ;; ... (II)
                      (set! engine-succeeded? #t)
                      result))
```

[7] Indeed, the flat engine could be considered a variant of the nester where fairness means the prohibition of children!

[8] Given a module-based Scheme, the code can be written as an engine module that abstracts over a fairness module.

[9] Some Scheme dialects provide an atom for an infinitely large number, on which the numerical procedures produce the expected results. In other dialects, any non-numerical atom may be chosen, with the procedures *min*, − and = redefined (in the lexical scope of the engine definition) to admit infinity as a possible argument.

```
                (lambda (r k)
                  (if (eq? r 'interrupt)
                      (lambda () (k #f)) r)))])
     ⟨ticks-postlude⟩
     ;; ... (III)
     (cond [engine-succeeded?
             (success result ticks-left)]
            [else (failure result)])))))
```

When the prompt returns, the variable *result* contains either the rest of the failed engine or a successful result, and the flag *engine-succeeded?* tells which of these is the case. Unfortunately, the code gives incorrect failed engines: the continuation denoting the interrupted engine includes the actions for setting the flag *engine-succeeded?* and stopping the clock. This will yield spurious results when the engine is resumed, whether as a plain thunk or as a fresh engine.

To avoid this, we use *two* prompts. The outer prompt encloses all the computation as before, including the thunk and the clock and flag operations. The new inner prompt surrounds only the setting of the clock and the call to the engine's thunk. The inner handler reacts to interrupts by *throwing* the rest of the engine to the outer prompt, thereby avoiding including the flag and clock operations in the thrown thunk. The outer handler disables interrupts that occur *after* the inner prompt has exited — this is done by resuming the interrupted computation:

```
;;; *** first modification, for (I) above ***
(let* (...
       [result
         (% 'engine
            (let ([result
                    (% 'engine
                       (begin (clock 'set true-ticks)
                         (thunk))
                       (lambda (r k)
                         (if (eq? r 'interrupt)
                             (fcontrol 'engine
                               (lambda () (k #f)))
                             r)))])
              (set! ticks-left (clock 'stop))
              ;; ... (II)
              (set! engine-succeeded? #t)
              result)
            (lambda (r k)
              (if (eq? r 'interrupt)
                  (k #f) r)))])
  ...)
```

A successful engine that finishes with no ticks to spare and suffers an interrupt between the two prompts *could* stop the clock *twice*. To avoid the second stop from setting the number of ticks left to infinity, the latter value must be coerced to zero:

```
;;; *** second modification, for (II) above ***
(set! ticks-left (infinity→0 (clock 'stop))) ...
```

where *infinity→0* is the function (**lambda** ($n$) (**if** (= $n$ infinity) 0 $n$)).

The engine currently run may be a child engine, in which case care is needed when invoking the *failure* operations. If the child has no ticks left, the parent may resume with the *failure* action on the rest of the child. If the child does have some ticks left, the child's failure was not because the ticks supplied by the user were insufficient, but because the fairness strategy curtailed its ticks. In the latter case, the parent must resume the child when the parent runs again:

```
;;; *** third modification, for (III) above ***
(cond [engine-succeeded? (success result ticks-left)]
      [(= ticks-left 0) (failure result)]
      [else ((make-engine result)
              ticks-left success failure)]) ...
```

Engines can be forced to stop immediately, either with a success value or as a failure. For a successful exit, use *fcontrol* tagged 'engine to transfer control and a success value to the engine prompt:

```
(define engine-return
  (lambda (v) (fcontrol 'engine v)))
```

To block an engine, i.e., compel it to fail, use *fcontrol* to force an interrupt:

```
(define engine-block
  (lambda () (fcontrol 'engine 'interrupt)))
```

### 3.3 The code for managing ticks

A flat engine needs very little tick management. The variable *true-ticks*, introduced in ⟨*ticks-prelude*⟩, is set to exactly the *ticks* argument supplied to the engine, since there are no parent engines. Some error-checking to ensure that there is no engine already running may be added:

```
;;; *** ⟨ticks-prelude⟩ for flat engines ***
(if (not (= (clock 'stop) infinity))
    (error 'engine "Trying to nest engines!"))
(let ([true-ticks ticks])
  ...)
```

The ⟨*ticks-postlude*⟩ for flat engines is empty.

For nestable engines, both the prelude and postlude codes are more elaborate. The algorithm first stops the currently active parent engine, if any, before running the new child engine. This yields the ticks left for the parent — infinity if there is no parent engine. For fair nesting, the child cannot be run beyond the parent's remaining ticks, regardless of the ticks allotted to the child in the program. Thus the child should be run for a number of ticks, *true-ticks*, that is the minimum of the parent's remaining ticks and the child's specified ticks. The variable *child-ticks-left* is that part of the child's ticks not accounted for by *true-ticks*, and should be remembered should the child be continued at some later time. Further, the time taken by the child is also counted against the parent — thus, *parent-ticks-left* is the parent's ticks less the child's true ticks.

```
;;; *** ⟨ticks-prelude⟩ for nestable engines ***
(let ([parent-ticks (clock 'stop)]
      [true-ticks (min parent-ticks ticks)]
      [parent-ticks-left
        (− parent-ticks true-ticks)]
      [child-ticks-left (− ticks true-ticks)])
  ...)
```

In the postlude, both the parent's and the child's remaining ticks are updated to include *ticks-left*, a nonzero number if the child finished successfully before *true-ticks* ran out. The clock is reset to *parent-ticks-left*, thereby restarting the parent engine computation:

```
;;; *** ⟨ticks-postlude⟩ for nestable engines ***
(set! parent-ticks-left (+ parent-ticks-left ticks-left))
(set! ticks-left (+ child-ticks-left ticks-left))
(clock 'set parent-ticks-left)
...
```

## 4 Backtracking through handling

Control handling provides an accessible approach to Prolog-style backtracking [2, 25]. Backtracking solves a problem or *goal* by trying to solve its *subgoals*. If the goal is a simple or *atomic* goal, it is solved by matching it with statements or *facts* in a database. A goal that is solved is said to succeed.

Given a *query* goal that is a *conjunction* of subgoals, the backtracker checks if each subgoal succeeds. If the query is a *disjunction*, the backtracker checks if at least one of the subgoals succeeds, keeping track of the rest of the subgoals with a *backtrack point*. Should a subgoal fail, the backtracker goes back to the dynamically closest backtrack point to try the next subgoal in *that* disjunction. If all such retries fail, the query as a whole fails.

Implementing backtracking in Scheme provides an apt use of continuations. While "purely functional" solutions with goals returning boolean values are possible, such methods require that goals explicitly call success and failure procedures to allow resumption of subgoals at backtrack points. In contrast, Scheme approaches [6, 10] aim for more concise and readable code using *call/cc*-continuations to identify and jump to backtrack points. Control handlers continue this tradition by simply using prompts to mark subgoals.

### 4.1 Unification and logic variables

An *atomic* goal is simply a predicate on terms, where terms are structured objects built from logic variables, numbers, lists and other datatypes. An atomic goal is solved by unifying the term structures composing the goal against facts in the database. (The unification process itself is a predicate: thus, the unification of two terms is an example of an atomic goal.) In this treatment, since our purpose is to study the backtracking capabilities provided by control handlers, we will not go into the details of implementing logic variables and unification in Scheme (refer [6, 10]).

### 4.2 Goals

In this treatment, a goal is a Scheme expression that *throws* (instead of just returning) the boolean *false* if it fails and a true value if it succeeds. In addition, in the latter case, the continuation of the throw represents a backtrack point if the goal is to be retried for an alternate solution. Thus, the "fail" goal is simply (*fcontrol* 'goal #f). The "true" goal is not (*fcontrol* 'goal #t) but (**begin** (*fcontrol* 'goal #t) (*fcontrol* 'goal #f)), since it should fail when retried.

A goal is evaluated by running it in a prompt: the handler handles the thrown continuation depending on whether the goal succeeded or failed. The thrown continuation is exactly the rest of the computation of the goal, in other words a representation of the backtrack point in the goal.

A user query is evaluated like any other goal, viz., inside a prompt: if it succeeds, its logic variables can be examined to see how the query was solved.

### 4.3 Disjunction and conjunction of goals

We now define[10] disjunctions (**or!**) and conjunctions (**and!**) as syntactic extensions that take an arbitrary sequence of goals as subexpressions. First, the disjunction:

```
(or! g ...) ≡
  (% 'goal
     (begin
       (% 'goal g
          (rec h
            (lambda (r k)
              (if r (begin
```

[10] The syntax **rec** helps define recursive functions: (**rec** $f$ $x$) $\equiv$ (**let** ([$f$ '*]) (**set!** $f$ $x$) $f$).

```
                                  (fcontrol 'goal #t)
                                  (% 'goal (k '*) h))))))
      ...
      (fcontrol 'goal #f))
    (rec h
      (lambda (r k)
        (if r (begin
                (fcontrol 'goal #t)
                (% 'goal (k '*) h))
              (fcontrol 'goal #f)))))
```

Each subgoal $g$ is tried successively in a separate prompt. If $g$ fails, its successor is tried, and so on. If, on the other hand, $g$ succeeds, its handler sends a signal of success to the caller of the disjunctive goal. However, $g$'s handler notes that the disjunction should backtrack at $g$'s own backtrack point before trying $g$'s successors. If all the subgoals fail, the disjunction itself fails. This is accomplished by throwing *false* after trying all the goals.

Conjunctions follow a related outline:

```
(and!) ≡
  (begin (fcontrol 'goal #t) (fcontrol 'goal #f))

(and! g g2 ...) ≡
  (% 'goal g
    (rec h
      (lambda (r k)
        (if r (% 'goal (and! g2 ...)
                (rec h2
                  (lambda (r2 k2)
                    (if r2 (begin
                             (fcontrol 'goal #t)
                             (% 'goal (k2 '*) h2)))
                    (% 'goal (k '*) h))))
            (fcontrol 'goal #f)))))
```

The first clause of the definition of **and!** shows that a vacuous conjunction is synonymous with a true goal. If subgoals are present, all of them should succeed for the conjunction to succeed. Each subgoal decides whether the subgoals following it should be tried or not. If a subgoal $g$ succeeds, its *handler* tries the conjunction of the remaining goals, $g2$, etc., but after noting that if these fail, $g$'s own backtrack point should be retried. If $g$ fails, its handler should signal overall failure, without trying $g$'s successors.

### 4.4 The cut

The above implements "pure" Prolog. Often, either for efficiency or a procedural style, we need to prune the backtracking possibilities: Prolog's method is the *cut* ("!"). The *cut* is a goal that succeeds but has the side-effect of committing all the goal choices made from a certain "cut entry" point to the point of the cut. In Prolog, the cut entry is always the immediately enclosing disjunction, but we can relax this restriction here. The syntax **or!!** stands for disjunctions with a cut entry point.

In our implementation, we simply add a handler tagged 'cut at the cut entry point. The cut itself is a goal that succeeds at first, but on backtracking, jumps to the cut entry point with a failure signal.

```
(or!! g ...) ≡
  (let ([cut (lambda ()
               (fcontrol 'goal #t) (fcontrol 'cut '*))])
    (% 'cut (or! g ...)
      (lambda (r k)
        (fcontrol 'goal #f))))
```

## 5 Conclusion

We have described a versatile control mechanism for programming languages that manipulate higher-order control. Control handling has been traditionally successful in first-order control arenas. When extrapolated *appropriately* to languages with higher-order control, it is an important programming tool, affording clean and easy solutions for a wide range of control tasks. Thus, this work bolsters our conclusion from studying denotational models that control handling is an indispensable addition to any programming language with control operators.

**Acknowledgment.** I thank Matthias Felleisen and Bruce Duba for helpful discussions.

## References


[1] W. Clinger, J. Rees, et al. Revised$^4$ Report on the Algorithmic Language Scheme, November 1991.

[2] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[3] O. Danvy and A. Filinski. Abstracting control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.

[4] B.F. Duba, R. Harper, and D. MacQueen. Typing first-class continuations in ML. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 163–173, 1991.

[5] R.K. Dybvig and R. Hieb. Engines from Continuations. *Journal of Computer Languages* (Pergamon Press), 14(2):109–124, 1989.

[6] M. Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University Computer Science Department, 1985.

[7] M. Felleisen. $\lambda$-v-CS: An Extended $\lambda$-Calculus for Scheme. In *Proc. 1988 Conference on Lisp and Functional Programming*, pages 72–84, 1988.

[8] M. Felleisen. The Theory and Practice of First-Class Prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[9] R. Harper. Introduction to Standard ML. LFCS Report Series ECS-LFCS-86-14, University of Edinburgh, 1986.

[10] C.T. Haynes. Logic Continuations. *J. Logic Program.*, 4:157–176, 1987. Preliminary version: In *Proc. of the Third International Conference on Logic Programming*, July 1985, London, England, *Lecture Notes in Computer Science*, Vol. 225, Springer-Verlag, Berlin, 671–685.

[11] C.T. Haynes and D.P. Friedman. Abstracting Timed Preemption with Engines. *Journal of Computer Languages* (Pergamon Press), 12(2):109–121, 1987. Preliminary version: *Engines Build Process Abstractions.* In *Proc. Conference on Lisp and Functional Programming*, 1985, 18–24.

[12] C.T. Haynes and D.P. Friedman. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems*, 9(4):245–254, 1987.

[13] C.T. Haynes, D.P. Friedman, and M. Wand. Obtaining Coroutines from Continuations. *Journal of Computer Languages* (Pergamon Press), 11(3/4):109–121, 1986.

[14] R. Hieb and R.K. Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 128–136, 1990.

[15] P.J. Landin. A Correspondence between Algol 60 and Church's Lambda Notation. *Communications of the ACM*, 8(2):89–101; 158–165, 1965.

[16] J. McCarthy et al. *Lisp 1.5 Programmer's Manual.* The MIT Press, 2nd edition, 1965.

[17] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

[18] C. Queinnec and B. Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 174–184, 1991.

[19] J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM Conference*, pages 717–740, 1972.

[20] D. Sitaram and M. Felleisen. Control Delimiters and Their Hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

[21] D. Sitaram and M. Felleisen. Reasoning with Continuations II: How to Get Full Abstraction for Models of Control. In *Proc. 1990 Conference on Lisp and Functional Programming*, pages 161–175, 1990.

[22] D. Sitaram and M. Felleisen. Modeling Continuations without Continuations. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 185–196, 1991.

[23] G.L. Steele Jr. *Common Lisp: the Language.* Digital Press, 1984.

[24] G.L. Steele Jr. *Common Lisp: the Language.* Digital Press, 2nd edition, 1990.

[25] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[26] J.E. Stoy and C. Strachey. OS6: An Operating System for a Small Computer. *Comp. J.*, 15(2):117–124, 195–203, 1972.

[27] G.J. Sussman and G.L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.