LISP AND SYMBOLIC COMPUTATION: An International Journal, 3, 67–99, 1990
© 1990 Kluwer Academic Publishers – Manufactured in The Netherlands

Control Delimiters and Their Hierarchies

DORAI SITARAM (*dori@rice.edu*)
MATTHIAS FELLEISEN* (*matthias@rice.edu*)
Department of Computer Science, Rice University, Houston, TX 77251-1892

Keywords: Scheme, Control structure, Continuations, Control delimiters, Engines

Abstract. Since control operators for the *unrestricted* transfer of control are too powerful in many situations, we propose the *control delimiter* as a means for restricting control manipulations and study its use in Lisp- and Scheme-like languages. In a Common Lisp-like setting, the concept of delimiting control provides a well-suited terminology for explaining different control constructs. For higher-order languages like Scheme, the control delimiter is the means for embedding Lisp control constructs faithfully and for realizing high-level control abstractions elegantly. A deeper analysis of the examples suggests a need for an entire *control hierarchy* of such delimiters. We show how to implement such a hierarchy on top of the simple version of a control delimiter.

1 The power of control operators

Since it is impossible to anticipate all possible needs for control abstractions during the design of a programming language, the inclusion of a low-level, powerful control operator seems to provide the most appropriate solution. With such an operator, a programmer can build the high-level control abstractions that are necessary for a specific problem. However, this strategy has two problems. On one hand, the implementation of high-level control abstractions with the general control operator hardly ever achieves the same efficiency as native versions of these abstractions. On the other hand, the low-level operator is often too powerful for a simple, faithful implementation of high-level abstractions.

A typical example is the programming language Scheme [14,18] and its control operator *call-with-current-continuation*, abbreviated *call/cc*. Similar examples are ISWIM and J [11], GEDANKEN with its first-class labels [15], and GL and *state* [9]. With *call/cc*, a program can access an abstraction of the current control state, the *continuation*, at any point during the evaluation. Like all other values in Scheme, continuations have first-class status; they can thus play the role of procedure arguments and results

*The work of both authors was partially supported by the National Science Foundation and DARPA.

or can be assigned to arbitrary variables. The combination of *call/cc* and first-class continuations provides the basis for implementing many different control abstractions as simple abbreviations [18] and for creating complex systems with various levels of program control [2,6,8].

Unfortunately, as Haynes and Friedman [7] observe in their treatise on constraining control, *call/cc* and its associated continuations are too general in many cases and inappropriate use can easily destroy the integrity of an embedded control abstraction. Their solution consists of a method for restricting the power of *call/cc* and continuations in appropriate ways. The crucial idea is to redefine *call/cc* so that continuations are always embedded in constraining procedural objects. With such redefinitions, *call/cc* and constrained continuations can simulate Lisp's **catch** and **throw**, protect the dynamic scope of a routine, and confine the use of continuations to dynamic domains. Though feasible, these solutions are often complex and difficult to understand. For example, the realization of **catch** and **throw** requires a complicated communication system for the continuation objects and moreover relies on a garbage collector for eliminating inaccessible continuations. Similarly, a facility for postfixing the dynamic scope of a routine relies on a central data structure for keeping track of the system's entire control tree.

Instead of accepting these complicated mechanisms for constraining control, we believe they reflect a fundamental problem of the underlying language. Although Scheme can create a *first-class* abstraction of the control state, it does not provide a *first-class* means for determining the extent of this control state. It always takes the *entire* control state from the current point in the evaluation to the *unique* end of the evaluation, which is the prompt in an interactive system. Unlike any other object in Scheme, this delimiter for control actions is a second-class citizen. We suggest making this control delimiter a first-class facility: the first-class prompt [4].

Our suggestion generalizes Stoy and Strachey's [17] *run* subroutine and Lisp's [13] **errset** facility.[1] The operator *run* and to a lesser extent **errset** allow a program to create tasks that share lexical information but are isolated with respect to non-local transfer of control. When added to a language with a control structure based on first-class continuations, the control delimiter induces two changes. First, any control actions that eliminate ongoing evaluations can only erase control information up to the dynamically closest control delimiter. Second, a control operator that provides access to the control state can only encapsulate the piece of control information between the current point of evaluation and the closest delimiter.

[1]Mitchell Wand and Andrew Black pointed out the relationships between prompts and **errset** and prompts and *run*, respectively.

Although the control delimiter solves several problems with existing control operators, it also creates the potential of undesirable interference between its uses in overlapping dynamic extents. In order to avoid such anomalies, we suggest a simple strategy for defining a *hierarchy* of delimiters. Since the control hierarchy is defined in terms of the raw control operators, it is easy to continue building hierarchies inside hierarchies and thus get arbitrarily grainy levels of control operators. The ability to create and hide the lower levels of these operators ensures that no program fragment can unintentionally or maliciously violate the security of the system or new layers of applications written on top of the system. The particular inter-level relationship we choose posits that every level in the hierarchy have complete power over its own control operations and the ones in levels above itself. We believe that this reflects the reality of multi-layered systems, except that such systems are usually implemented combining facilities from different languages and with less flexibility. By providing simple facilities for a control hierarchy, we get a powerful yet secure language for many layers of application and systems programming.

In the second and third section, we introduce our variant of Scheme and our proposal for an alternative control structure, respectively. The fourth section presents a method for implementing our control structure through a modification of a Scheme implementation and, less efficiently, within Scheme. The fifth section illustrates the use of control delimiters in a Common Lisp-like setting; it demonstrates how the idea of control delimiters provides the proper terminology for modeling and experimenting with different versions of **catch** and **throw** and **unwind-protect**. Following this, we illustrate the primitive use of our control structure in a higher-order setting with powerful abstractions for coroutines, time-preempted computations, and stream processing routines. Finally, in Section 7, we show how the naïve use of control delimiters leads to problems and how these problems are attenuated by generalizing the control delimiter to hierarchies of control delimiters. The last section briefly summarizes our approach and puts it in perspective.

2 A brief introduction to Scheme

Scheme [14,18] is an expression-oriented language with call-by-value, lexically-scoped, first-class procedures, and has imperative extensions for lexical assignment and control manipulation. In this section, we describe the core constructs of our dialect of Scheme, including its standard control structure. We then give a brief account of how to enrich the language with syntactic extensions.

The following EBNF specifies the syntax of well-formed expressions in core Scheme:

```
exp ::= basic-constant
     |  false
     |  true
     |  id
     |  (quote exp)
     |  (lambda (id ...) exp ...)
     |  (sigma (id ...) exp ...)
     |  (if exp exp exp)
     |  (exp exp ...)
     |  apply
     |  call/cc
```

The notation *item* ... is used to denote multiple (possibly zero) occurrences of the syntactic object *item*.

Basic constants are mutable and immutable data (numbers, symbols, dotted-pair structures, etc.) and basic procedures on such data (+, *, cons, car, set-cdr!, etc.). An identifier is a placeholder for a value that is determined by the lexical context (i.e., the lexically closest **lambda**-abstraction and/or the dynamically closest side-effect). A **quote**-expression is an atomic value or a dotted-pair structure; '*exp* abbreviates (**quote** *exp*). To permit selective evaluation inside a quoted expression, the backquote is used: ‘*exp* is identical to '*exp* except that subexpressions preceded by a comma are evaluated. A **lambda**-expression evaluates to a closure, a first-class procedural object. On application, a closure *establishes* bindings between the identifiers in the parameter list and the corresponding argument values; it then continues with the sequential evaluation of the body expressions. A **sigma**-expression [3] evaluates to a **sigma**-capability, a closure-like object that, on application, *modifies* the existing bindings of its parameters. Most Scheme implementations instead provide the assignment form **set!**, which takes an identifier and a subexpression, and modifies the binding of the identifier to the value of the subexpression. Conditionals are introduced by **if**: a special constant false is Scheme's *false* value; all other values, including a special constant true, count as *true*. An application is a non-empty sequence of expressions; in our dialect, these are evaluated left to right. An alternative means for performing applications uses the procedure *apply*, which is called with two arguments: the procedure and the list of the procedure arguments. The procedure *call/cc* applies its argument to an abstraction of its control context called an *abortive continuation*. When invoked with a value, such a continuation abandons its current context and continues evaluation with the value at the context captured in the continuation.

A Scheme expression is evaluated in a global environment. The global environment provides an extensible and modifiable set of semantic bindings. The form (**define** *id exp*) is used to bind *id* to the value of *exp*.

Scheme allows the user to define syntactic extensions with the form **extend-syntax** [10]. It takes a list of keywords—a *primary* keyword followed by optional *auxiliary* keywords—and a sequence of specification clauses. Each specification consists of an abbreviation pattern and a corresponding expansion pattern:

```
(extend-syntax (<keyword> ...)
  [<abbreviation> <expansion>] ...).
```

A syntactic preprocessor reduces each input expression that matches the first abbreviation pattern to the appropriate core expression. In addition, the preprocessor has the ability to process ellipsis, ..., in patterns, as well as to prevent variable capture when new variable bindings are established in the expansion [10].

As an example for the use of ellipses, consider the definition of a **let**-expression:

```
(extend-syntax (let)
  [(let ([x exp] ...) body ...) ((lambda (x ...) body ...) exp ...)]).
```

A **let**-expression specifies local bindings with initial values for use within the **let**-body. Such an expression is transformed into the application of a **lambda**-expression, where the **lambda**-parameters stand for the local variables and are bound to the initial values through immediate application. As an example for hygienic expansion, we define a conditional **or**:

```
(extend-syntax (or)
  [(or x y) (let ([v x]) (if v v y))]).
```

The form **or** introduces a lexical variable *v* for the value of its first subexpression. If this is non-false, it is returned without evaluating the second parameter, otherwise the value of the latter is the result of the **or**-expression. In a naïve expansion system, the new lexical variable *v* would bind free occurrences of *v* in the second subexpression of **or**. The *hygienic* macro expansion method [10] *automatically* avoids such unintended variable bindings without further instructions from the programmer. Sometimes an identifier introduced by an expansion is *meant* to capture bindings in the syntactic extension: such identifiers are listed as auxiliary keywords to exempt them from hygienic expansion.

```
(extend-syntax (rec)
  [(rec name exp) (let ([name 'any]) ((sigma (name) name) exp))])

(extend-syntax (letrec)
  [(letrec ([x exp] ...) body ...)
   (let ([x 'any] ...) ((sigma (x ...) body ...) exp ...))])

(extend-syntax (set!)
  [(set! ([x exp] ...) body ...) ((sigma (x ...) body ...) exp ...)]
  [(set! name exp) ((sigma (name) 'any) exp)])

(extend-syntax (iterate)
  [(iterate loop ([x exp] ...) body ...)
   ((rec loop (lambda (x ...) body ...)) exp ...)])

(extend-syntax (begin0)
  [(begin0 first-exp exp ...)
   (let ([first-val first-exp]) exp ... first-val)])

(extend-syntax (and)
  [(and) true] [(and x y ...) (if x (and y ...) false)])

(extend-syntax (or)
  [(or) false] [(or x y ...) (let ([v x]) (if v v (or y ...)))])

(extend-syntax (cond else)
  [(cond) false]
  [(cond [else else-exp ...]) (begin else-exp ...)]
  [(cond [test exp ...] clause ...)
   (let ([val test]) (if val (begin val exp ...) (cond clause ...)))])

(extend-syntax (record-case else)
  [(record-case rcd) 'any]
  [(record-case rcd [else else-body ...]) (begin rcd else-body ...)]
  [(record-case rcd [tag comp body ...] clause ...)
   (let ([r rcd])
     (if (eq? (car r) tag) (apply (lambda comp body ...) (cdr r))
         (record-case r clause ...)))])
```

Figure 1: Some commonly used syntactic extensions

Some common syntactic extensions are defined in Figure 1. The **letrec**-expression introduces local definitions just as **let** does, but these definitions are mutually recursive; **rec** is used to define a single recursive procedure. The traditional Scheme assignment form, **set!**, is defined as the application of a **sigma**-closure that modifies the binding of the **sigma**-parameter. However, in keeping with the expression-oriented nature of Scheme, we also supply an alternative expansion pattern for **set!**—one that relates to **sigma** much as **let** does to **lambda**: a **set!**-expression modifies the bindings of its lexical variables and proceeds with the evaluation of its body. The form **iterate** effects a recurring evaluation, similar to a loop. The form **begin0** evaluates a sequence of expressions, returning the value of its first subexpression. The forms **and** and **or** perform the boolean operations of conditional *and* and *or*. The form **cond** is a generalization of **if** to include many sub-clauses. The form **record-case** dispatches on a given expression: if it is a record with the tag specified in one of its clauses, the identifiers in the clause are bound to the components of the record before executing the body; otherwise, the default action specified in the **else**-clause is carried out.

3 Functional continuations and control delimiters

Our dialect differs from traditional Scheme in the choice of control operators. Instead of *call/cc* and *abortive* continuations, it has *control* and *functional* (i.e., *non-abortive*) continuations [5,9]. [2] The operator *control* takes a single argument. When invoked, *control* encodes its current evaluation context as a **lambda**-closure, the *functional continuation*, and applies its argument to this continuation in the *empty* control context.

For an illustration, let us work through some examples. Consider the expression:

(add1 (*control* (**lambda** (k) 0))).

The evaluation context of the *control*-application is (add1 $\cdots$); the corresponding procedural abstraction, viz., the functional continuation, is (**lambda** (x) (add1 x)). The argument of *control*, viz., (**lambda** (k) 0), is now applied to this abstraction in the empty context:[3]

(add1 (*control* (**lambda** (k) 0)))
$\Rightarrow$ ((**lambda** (k) 0) (**lambda** (x) (add1 x)))
$\Rightarrow$ 0.

[2]The use of *control* and functional continuations is not necessary for our development, but is advantageous in many situations.

[3]The symbol $\Rightarrow$ should be read as "evaluates to."

In other words, a vacuous abstraction as a *control*-argument aborts the program. We therefore introduce the following syntax:

```
(extend-syntax (abort)
  [(abort exp) (control (lambda (dummy) exp))]).
```

Recall that, by hygienic expansion, *dummy* does not bind any identifier in *exp*.

On applying a functional continuation to a value, the latter is placed in the context determined by the former. The computation of the functional continuation sends its result back to the context of the invocation. Thus, we have

```
(add1 (control (lambda (k) (k 0)))) ⇒ 1;
```

and, since the closure can also be applied repeatedly, we also have:

```
(add1 (control (lambda (k) (k (k 0))))) ⇒ 2.
```

The operator *control* is powerful enough to simulate the discarded operator *call/cc*. Recall that *call/cc* is a procedure that calls its unary argument with the current *abortive* continuation. The operator *control* performs a similar action. Thus, in order to define *call/cc* with *control*, we could try the following:

```
(define call/cc
  (lambda (f)
    (control (lambda (k) (f ··· k ···))))).
```

In contrast with *call/cc*, which proceeds in its evaluation context, *control* calls its argument in the *empty* context. Thus in our definition of *call/cc*, we need to invoke the functional continuation in order to reestablish the correct control context:

```
(control (lambda (k) (k (f ··· k ···)))).
```

Furthermore, the continuation provided by *call/cc* is *abortive*. Invoking an abortive continuation is a *jump*, i.e., the invoker's control context is abandoned. To transform *control*'s functional continuation to an abortive one, we modify the former into a procedure that performs the invocation of the continuation and immediately aborts:

```
(lambda (v) (abort (k v))).
```

Together, the complete definition of *call/cc* in terms of *control* reads:

```
(define call/cc
  (lambda (f)
    (control (lambda (k)
               (k (f (lambda (v) (abort (k v))))))))).
```

Continuations that merely involve control transfer but no passage of information should properly be closures of no arguments (thunks). Currently, invocations of such continuations take a dummy argument. Instead, we define *control0* so that *control0*-continuations are thunks:

```
(define control0
  (lambda (f)
    (control (lambda (k) (f (lambda () (k 'any))))))).
```

In addition to *control*, our dialect of Scheme provides the procedure *run* for delimiting the dynamic extent of control operations [4]. The procedure *run* creates a task from a thunk, which is a procedure of no arguments, and runs it as an *independent* program. The task does not inherit its creator's control context, but it does share its lexical bindings. The result of this task is *always* passed to the context in which the *run*-application occurs.

If *run*'s argument contains no control manipulation, the application of *run* is vacuous. If, however, *run*'s argument uses *control*, the two actions associated with *control* are delimited by *run*. First, the functional continuation created by *control* represents the portion of the surrounding context delimited by the dynamically closest *run*. Second, the *control*-expression erases its current context as usual, but only up to the dynamically nearest *run*.

In a sense, the procedure *run* acts as a *user-available* prompt: one can always expect a result to be returned to the context of a *run*-application, much as an interactive command always returns with a result at a command-line prompt. A convenient syntactic form is

```
(% e ...),
```

which expands as follows:

```
(extend-syntax (%)
  [(% e ...) (run (lambda () e ...))]).
```

We shall henceforth use "*run*" and "prompt" interchangeably, preferring "prompt" when we are emphasizing a program-context, and "*run*" when we are highlighting the use of the control operator as a *procedure*.

As an illustration of how *run* delimits transfer of control, consider (add1 (% (add1 (*control* (**lambda** (k) 0))))).

The %-expression is *run* as the independent program:

(add1 (*control* (**lambda** (k) 0))).

From a previous example, we know that this expression evaluates to 0. This result is sent to the %-context, viz., (add1 ⋯), and therefore, the entire expression evaluates to 1.

For a more complex example, consider the following expression, which invokes a continuation inside a prompt:

```
(let ([g (% (* 2 (control (lambda (k) k))))])
  (* 3 (% (* 5 (abort (g 7)))))).
```

The local variable g is bound to the functional continuation representing (* 2 ⋯):

(**lambda** (x) (* 2 x)).

The **let**-body contains (* 5 (**abort** (g 7))) as an independent task that shares the lexical variable g with its parent program. This evaluates as follows:

```
   (* 5 (abort (g 7)))
⇒ (g 7)
⇒ ((lambda (x) (* 2 x)) 7)
⇒ (* 2 7)
⇒ 14.
```

This result is returned to the context of the prompt-expression, (* (% ⋯) 3), yielding 42.

The prompt of Scheme's interactive loop is an implicit *run* surrounding each Scheme program. The interactive loop uses *base-run*, a variant of *run*, as a catch-all delimiter for every control manipulation in the Scheme program. Moreover, the identifier *base-run* can be redefined as a different procedure. This, we shall see, facilitates the development of powerful methods for creating control hierarchies.

4 Implementing *run* and *control*

The implementation machinery for *call/cc* packages the *entire* control stack into a continuation object for the user. A modification of this machinery into one that can package a contiguous *portion*, rather than the whole, of the control stack leads to a *native*[4] implementation of *control*. To include *run*, the implementation must also identify points on the stack that restrict *control*'s manipulations of the stack. Alternatively, the original *call/cc* can *simulate* these actions, *albeit inefficiently*, providing an *embedding* of the operators *run* and *control* in standard Scheme. We describe both strategies in the following subsections; we recommend skipping the second subsection on a first reading.

4.1 A strategy for a native implementation

A native implementation manipulates the control stack directly. At any stage, the control stack is the machine equivalent of the evaluation context of the program subexpression currently being evaluated. A Scheme program starts executing in an empty control context. The control context is represented by a stack, the empty context by the empty stack; sub-evaluations cause the stack to grow. The stack always contains enough information for the completion of the rest of the evaluation.

Conceptually, a *run*-application marks the top of the current control stack. A *control*-application provides the programmer with an abstraction of the top portion of the stack—from the top down to the closest *run*-mark. This is the functional continuation. The application simultaneously erases this portion off the stack and applies the *control*-argument to the functional continuation. Invoking a functional continuation on a value re-installs the abstracted partial stack on top of the current control stack, and then proceeds as if the value were returned from a sub-evaluation.

Two special uses of *control* lead to important optimizations. First, if the *control*-argument ignores its argument, the functional continuation need not be created. Second, if the *control*-argument immediately applies the functional continuation, the top portion of the stack need not be erased. As we shall see in the following sections, these cases occur quite frequently.

4.2 Embedding *run* and *control* in standard Scheme

The operator *call/cc* actually suffices to simulate the above strategy in a Scheme system without modifying the underlying implementation.[5] The

[4]By a "*native*" implementation of a facility we mean one where the facility is incorporated into the code generator for the system: it thus has a potential for efficiency not available to facilities built on top of the system.

[5]Guillermo Rozas was the first to claim the existence of such a solution—our solution assumes that Scheme has the procedure *eval* for evaluating textual arguments in the *global* environment. The implementation of *run* is not faithful because it is not tail-recursive: whereas (**iterate** *loop* () (*run* (**lambda** () (*loop*)))) is a tight loop in a system with native *run*/*control*, it exhausts stack space in the embedding.

continuations obtained by calling *call/cc* at appropriate points provide an explicit representation of the underlying control stack. Since the creation and manipulation of this stack representation use abortive continuations extensively, this strategy for *embedding* the operators *run* and *control* is less efficient than a native implementation.

The embedding captures the abortive continuations at each *run*- and *control*-application *and* at each *invocation point* of a functional continuation, and uses a stack to manage the transfer of control to the various *run*- and invocation-point continuations. Each *run*-application stores its abortive continuation in a new topmost frame on the stack, because a *control*-application in its dynamic extent must jump to the evaluation context of this *run*. A *control*-application provides its argument with its abortive continuation packaged into a procedure that simulates the functional continuation. In order to realize the *functional* behavior of *control*'s continuations, an invocation of such a continuation adds its invocation point to the topmost frame of the stack. The *run*-application takes care of returning program execution to each of its associated invocation points.

The stack data structure *run-stack* represents the underlying control stack. Each *frame* in the *run-stack* corresponds to a prompt, and contains the abortive continuation at the prompt as well as the sub-stack of the invocation points captured within this prompt. Initially, the *run-stack* is empty:

```
(define run-stack '()).
```

The implementation provides a thunk *reset-loop* that clears the *run-stack* and spawns a new *read-eval-print* loop. This interactive loop iteratively reads an input expression, surrounds it with the outermost prompt, *base-run*, and evaluates it:[6]

```
(define reset-loop
  (lambda ()
   (set! ([run-stack '()])
    (iterate read-eval-print ()
     (printf "~s~n" (eval '(base-run (lambda ()
                                   ,(prompt-read "% ")))))
     (read-eval-print))))).
```

The identifier *base-run* is initially bound to *run*.

Each *run*-application captures its abortive continuation and pushes it along with a new empty sub-stack for invocation points atop the *run-stack*.

[6]In order to ensure that the implementation is not corrupted by calls to the error and interrupt handlers, we can redefine the latter to call *reset-loop*.

```
(define run
  (lambda (th)
    (let ([run-cont 'any])
      (let ([v ((call/cc (sigma (run-cont)
                   (set! ([run-stack (cons (cons run-cont '()) run-stack)])
                    th))))])
        (let ([top-frame (car run-stack)])
          (let ([top-run-cont (car top-frame)]
                [top-sub-stack (cdr top-frame)])
            (cond [(not (null? top-sub-stack))
                   (let ([k (car top-sub-stack)])
                     (set-cdr! top-frame (cdr top-sub-stack)) (k v))]
                  [(not (eq? run-cont top-run-cont))
                   (top-run-cont (lambda () v))]
                  [else (set! ([run-stack (cdr run-stack)]) v)])))))))).
```

Figure 2: Embedding *run* in Standard Scheme

If the *run*-argument returns normally, the *run-stack* is popped and the value returned. Thus, the outline of the procedure *run* is:

```
(define run
  (lambda (th)
    (let ([run-cont 'any])
      (let ([v ((call/cc (sigma (run-cont)
                   (set! ([run-stack (cons (cons run-cont '()) run-stack)])
                    th))))])
        ...
        (set! ([run-stack (cdr run-stack)]) v))))).
```

The prompt continuation stored in *run-stack* expects a thunk and thaws it: with this tactic, a specified action can be performed *after* a jump has been made to the prompt.

Each *control*-application jumps to its nearest prompt-context; it does this by clearing the topmost sub-stack of invocation points on *run-stack* and invoking the associated prompt continuation. Further, *control* simulates its functional continuation with an abortive continuation and the saved topmost sub-stack on the *run-stack*. Together, these two pieces of information effectively describe the functional continuation grabbed by *control*. The argument of *control* is now applied to an object that simulates

```
(define control
  (lambda (f)
    (call/cc (lambda (control-cont)
      (let ([control-frame (car run-stack)])
        (let ([control-run-cont (car control-frame)]
              [control-sub-stack (cdr control-frame)])
          (set-cdr! control-frame '())
          (control-run-cont
            (lambda ()
              (f (lambda (v)
                   (call/cc (lambda (invoke-cont)
                     (let ([invoke-sub-stack (cdr (car run-stack))])
                       (set-cdr! invoke-frame
                         (append control-sub-stack
                                 (cons invoke-cont invoke-sub-stack)))
                       (control-cont v))))))))))))))
```

Figure 3: Embedding *control* in Standard Scheme

the behavior of a functional continuation. Thus, the code for *control* looks approximately like

```
(define control
  (lambda (f)
    (call/cc (lambda (control-cont)
      (let ([control-frame (car run-stack)])
        (let ([control-run-cont (car control-frame)]
              [control-sub-stack (cdr control-frame)])
          (set-cdr! control-frame '())
          (control-run-cont
            (lambda () (f (lambda (v) ... (control-cont v))))))))))).
```

Since the prompt continuation expects a thunk, a *control*-application can abort to its prompt *before* calling the *control*-argument on its functional continuation. Thus, *control*-applications are *pure jumps*, e.g., the loop (**iterate** *loop* () (**abort** (*loop*))) does not run out of stack space.

The functional continuation object has access to the abortive continuation and the sub-stack of invocation points grabbed by *control*. Upon invocation, the functional continuation must first push its own current abortive continuation (invocation point) on to the topmost sub-stack on *run-stack*, so that computation can return to the current evaluation context. It then reinstalls the recorded invocation points by pushing them all atop the topmost sub-stack. Finally, the continuation object transfers control to the recorded abortive continuation:

```
(lambda (v)
  (call/cc (lambda (invoke-cont)
    (let ([invoke-top-frame (car run-stack)])
      (let ([invoke-sub-stack (cdr invoke-top-frame)])
        (set-cdr! invoke-top-frame
          (append control-sub-stack (cons invoke-cont invoke-sub-stack)))
        (control-cont v)))))).
```

An invocation of the abortive continuation associated with a *control*-application eventually reaches the prompt-context that enclosed the original call to *control*. Since this may differ from the prompt-context enclosing the *invocation* of the functional continuation, we add code in the body of the procedure *run* that checks if the *run*-argument has returned in the proper prompt-context, and if not, jumps to the topmost prompt on the *run-stack*:

```
(let ([top-frame (car run-stack)])
  (let ([top-run-cont (car top-frame)] [top-sub-stack (cdr top-frame)])
    (cond [...]
          [(not (eq? run-con top-run-cont)) (top-run-cont
                                              (lambda () v))]
          [else (set! ([run-stack (cdr run-stack)]) v)]))).
```

The prompt also needs to dispatch control back to each of the invocation points in its frame so that *control*'s continuations are *functional*. The following code in *run*'s body performs this dispatch in stack order for the entire frame before the prompt finally returns a value:

```
(cond [(not (null? top-sub-stack))
        (let ([k (car top-sub-stack)])
          (set-cdr! top-frame (cdr top-sub-stack)) (k v))]
      [(not (eq? run-cont top-run-cont)) (top-run-cont (lambda () v))]
      [else (set! ([run-stack (cdr run-stack)]) v)]).
```

Figures 2 and 3 collect the above code fragments into the final definitions for *run* and *control*.

5 First-order control abstractions with prompts

In contrast to Scheme-like languages, Common Lisp [16] (and older dialects of Lisp) provide language features for *first-order* control manipulation. First-order control operations suffice for many traditionally important uses of evaluation control such as aborting subcomputations, exiting procedures and loops, and handling basic exceptions. The crucial implementation characteristic of such first-order manipulations is that they cannot reach beyond the dynamic extent of the control expression. Consequently, first-order control operators avoid the need for *copying* a portion of the run-time stack and for *switching* stacks. They only require the ability to mark the control stack and to erase it down to a chosen mark. Therefore, such operations are also called *stack-based*, for they avoid *heap* allocations for the run-time stack or copies of it.

The more powerful higher-order operators can simulate first-order behavior [7], but such simulations usually require heap-based implementations. With the operators prompt and **abort**, simulations of first-order operators are faithful, simple and truly stack-based.

5.1 Catch and throw

Traditional Lisp systems provide first-order control manipulation with the pair of operators **catch** and **throw**. The form (**catch** *tag exp*) marks the control stack with a user-defined tag; the form (**throw** *tag exp*) erases the control stack down to the closest matching tag.

In our dialect, the same stack-based behavior can be obtained with the pair prompt and **abort**. After all, a prompt marks the stack, and an **abort**-statement deletes the stack down to this mark. What we need is a mechanism for associating a tag with a particular prompt. We choose a message-passing protocol. In such a set-up, a **throw** sends a unique **throw**-*message* containing its tag and the thrown value to the nearest **catch**:[7]

```
(extend-syntax (throw)
  [(throw tag value) (abort (list 'throw tag value))]).
```

The **catch**-operator, on receiving a **throw**-message, checks the tag of the message against its own, and then decides whether to return the thrown value or **throw** it further to some other enclosing **catch**:

[7]In a real implementation, the tag throw would have to be replaced by a unique token.

```
(extend-syntax (catch)
  [(catch tag exp)
   (let ([catch-tag tag] [result (% exp)])
     (record-case result
       ['throw (throw-tag throw-value)
               (if (eq? throw-tag catch-tag) throw-value
                   (throw throw-tag throw-value))]
       [else result]))]).
```

Otherwise, if the returned value is not a **throw**-message, the evaluation of **catch**'s sub-expression terminates normally and returns a simple value.

A **throw** to a non-existent tag will eventually arrive at the outermost *read-eval-print* loop's prompt. When this happens, the loop should issue an appropriate error message. Given that the loop uses *base-run*, we can issue such an error message by redefining this routine as:

```
(define base-run
  (lambda (thunk)
    (let ([result (run thunk)])
      (record-case result
        ['throw (throw-tag throw-value)
                (printf "Throw to unknown tag: ~a ~a"
                        throw-tag throw-value)]
        [else result])))).
```

Since our building blocks, prompt and **abort**, only perform truly stack-based control manipulations, our version is a simple and faithful transliteration of the control manipulation provided by *native* **catch** and **throw**.

With Scheme's *call/cc*, on the other hand, a simulation of **catch** and **throw** is heap-based. For example, Haynes and Friedman [7] show how to use *call/cc* and abortive continuations to define a variant of *call/cc* called *call/cc-stack-based*. This is equivalent to **catch**, and invoking the abortive continuation obtained with *call/cc-stack-based* is equivalent to **throw**. However, *call/cc-stack-based* manipulates the control stack extensively. A **catch** in their implementation captures the current control stack, while a **throw** replaces its entire current stack by the captured one. The operator *call/cc-stack-based* produces abortive continuations using *call/cc* and packages them in *continuation objects*. These objects can identify whether their continuation lies below the current context on the runtime stack. Upon invocation, such an object sends messages to those continuation objects that should no longer be reachable and thus disables their future use. It is left to the garbage collector to reclaim these unreachable continuation objects; however, if there are still references to these objects, the collector may not identify them as garbage and retain them beyond their appropriate life-span.

5.2 Unwind-protect

As a further example of where the constraining action of the prompt is useful, we consider the **unwind-protect** facility of Lisp systems [16]. An **unwind-protect** form has two parts: a *body* and a *postlude*:

(**unwind-protect** *body postlude*).

The task of **unwind-protect** is to guarantee the execution of *postlude*, whether the evaluation of *body* terminates normally or by a non-local transfer of control through a **throw**. A *postlude* is used to specify clean-up operations: a typical *postlude* involves closing files or releasing resources used by the *body*.

As a first attempt, we could define the following:

```
(extend-syntax (unwind-protect)
  [(unwind-protect body postlude) (begin0 (% body) postlude)]).
```

This version ensures *postlude*'s execution but at a significant cost: the prompt enclosing *body* intercepts all attempts to transfer control from inside *body* to points outside of **unwind-protect**.

The problem with the above attempt is that **unwind-protect**'s prompt cannot distinguish between ordinary result values and *thrown* values. For the former, *postlude* must be executed and the value returned to the calling context. For the latter, however, the execution of *postlude* must be followed by a **throw** of the result value to the appropriate **catch**-context. The improved solution reads:

```
(extend-syntax (unwind-protect)
  [(unwind-protect body postlude)
   (let ([result (% body)])
     postlude
     (record-case result
       ['throw (tag value) (throw tag value)]
       [else result]))]).
```

It is easy to see that this implementation provides the *standard* **unwind-protect** facility with the correct behavior.

Certain situations call for an improved model. In particular, the user may want to deal differently with non-local exits occurring in *postlude*.[8]

[8] We thank one of the referees for bringing the following alternative models to our notice.

As an example, consider the expression,

```
(catch 'outer
  (iterate loop ()
    (catch 'inner
      (unwind-protect (throw 'outer true) (throw 'inner true)))
    (loop))).
```

With the current model, the above example loops forever, because the **throw** from *postlude* spawns a new iteration of the **unwind-protect**-expression before the **throw** from the body can ever take effect.

One alternative is to prohibit any non-local exits from *postlude* beyond the **unwind-protect**. To accomplish this, we simply constrain *postlude* with a prompt:

```
(extend-syntax (unwind-protect)
  [(unwind-protect body postlude)
   (let ([result (% body)])
     (% postlude)
     (record-case result
       ['throw (tag value) (throw tag value)]
       [else result]))]).
```

This guarantees that **throws** from *body* are not affected by **throws** from *postlude*. With this interpretation for **unwind-protect**, the above example terminates, returning true.

Sometimes, the neglect of exits from *postlude* in favor of exits from *body* is inappropriate. Consider, for instance,

```
(catch 'very-outer
  (catch 'outer
    (iterate loop ()
      (catch 'inner
        (unwind-protect (throw 'outer true) (throw 'very-outer false))
        (loop))))).
```

Here the expression returns true, since the **throw** from the *body* of the **unwind-protect** is chosen over the **throw** from the *postlude*. Arguably, the **throw** from *postlude* should dominate, as it reaches beyond the **throw** from *body*.

Our third model allows **throws** from both *body* and *postlude*, in such a way that the one going to the furthest enclosing **catch** prevails. To accommodate this behavior, we use unwind-messages in addition to throw-messages. An unwind-message encodes all the **throws** in the form of an association-list containing the corresponding tags and values. Each **catch**, on encountering such a message, deletes from the unwind-message's list any **throw** to its tag, and passes along the list to the next enclosing **catch**. In this manner, the unwind-message is whittled down to a list of a single tagged value, when a straightforward **throw** is effected to the corresponding **catch**.

The new definition of **catch**, which takes care of unwind-messages, is as follows:[9]

```
(extend-syntax (catch)
  [(catch tag exp)
   (let ([catch-tag tag] [result (% exp)])
     (record-case result
       ['throw (throw-tag throw-value)
               (if (eq? catch-tag throw-tag) value
                   (throw throw-tag throw-value))]
       ['unwind (tagvals)
                (let ([new-tagvals (remq (assq catch-tag tagvals) tagvals)])
                  (if (null? (cdr new-tagvals))
                      (throw (caar new-tagvals) (cdar new-tagvals))
                      (unwinder new-tagvals)))]
       [else result]))]).
```

The form **throw** remains the same. The procedure *unwinder* packages an association-list of tags and values into an unwind-message and sends it to the enclosing prompt:

```
(define unwinder (lambda (tagvals) (abort (list 'unwind tagvals)))).
```

The form **unwind-protect** constrains both *body* and *postlude* with prompts, which may thus receive either ordinary values, throw- or unwind-messages. A straightforward decision based on these results either returns a value or propagates a throw- or unwind-message:

[9]For clarity, we have used a functional method for propagating unwind messages past concentric **catch**-expressions. A more efficient implementation could have a stack of **catch**-tags and have the unwind-message jump immediately to the appropriate **catch**-expression.

```
(extend-syntax (unwind-protect)
 [(unwind-protect body postlude)
  (let ([result-body (% body)])
   (let ([result-postlude (% postlude)])
    (record-case result-body
      ['throw (tag1 val1)
             (record-case result-postlude
               ['throw (tag2 val2)
                       (unwinder (list (cons tag1 val1)
                                       (cons tag2 val2)))]
               ['unwind (tagvals2)
                       (unwinder (cons (cons tag1 val1) tagvals2))]
               [else (throw tag1 val1)])]
      ['unwind (tagvals1)
              (record-case result-postlude
                ['throw (tag2 val2)
                        (unwinder (cons (cons tag2 val2) tagvals1))]
                ['unwind (tagvals2)
                        (unwinder (append tagvals1 tagvals2))]
                [else (unwinder tagvals1)])]
      [else (record-case result-postlude
             ['throw (tag2 val2) (throw tag2 val2)]
             ['unwind (tagvals2) (unwinder tagvals2)]
             [else result-body])])))]).
```

If both *body* and *postlude* return normally, the **unwind-protect**-expression terminates with the value of *body*. A **throw** in either *body* or *postlude* arrives at the respective prompt as either a throw- or an unwind-message. If only one of either *body* or *postlude* produces such a message, it propagates unchanged to the next enclosing prompt. If both of them produce **throws**, the messages are merged into a single unwind-message to the next enclosing prompt.

6 Prompts in higher-order languages

Beyond first-order control, the new control operators provide simple and efficient macro-implementations of many high-level control paradigms. Typical examples are coroutines and engines. Higher-order control manipulation moreover requires generalizations of **unwind-protect** such as **dynamic-wind** and **wind-unwind** [7], both of which are straightforward modifications of the above code for **unwind-protect**. In addition, prompts and functional continuations give rise to new programming styles that unify such diverse directions as imperative program schemas and stream programming. We elaborate on these topics in the following subsections.

6.1 Coroutines

A coroutine [12] generalizes the concept of a procedural abstraction by including a local control state. A *call* statement invokes a coroutine. This is similar to procedure invocation in that the body of the coroutine starts executing. However, at any point inside the coroutine, a *resume* statement can transfer control to a different coroutine. The suspended coroutine stores its remaining computation in its local control state. On resuming a suspended coroutine, computation proceeds from the point saved.

Haynes, Friedman, and Wand [8] describe a succinct implementation of coroutines using *call/cc*. Each coroutine is an object with an internal control state; initially, this describes the entire coroutine computation. Upon invocation, computation proceeds according to the local control state. A *resume* instruction captures the current abortive continuation with *call/cc* and stores it in the local control state, before invoking the destination coroutine. Continuing a suspended coroutine reinstates the continuation stored in its local control state.

Using *call/cc* implies that the continuation captured by a *resume* statement is the entire control stack, whereas only the portion of the stack corresponding to the rest of the coroutine computation is needed. Using *control*, we obtain a slightly simpler solution:

```
(extend-syntax (coroutine resume)
  [(coroutine x e ...)
   (letrec ([LCS (lambda (x) e ...)]
            [resume (lambda (c v) (control (sigma (LCS) (c v))))])
     (lambda (v) (LCS v)))]).
```

The lexical variable *LCS* contains the local control state. The coroutine itself is a unary procedure that always calls its local control state. The *resume* statement uses *control* to capture and abandon the rest of the coroutine, stores the continuation in *LCS*, and continues with the resumed coroutine. The procedure *call* is

```
(define call (lambda (c v) (% (c v)))).
```

The prompt introducing the coroutine call ensures that the *control*-application in *resume* correctly identifies the rest of the coroutine.

Haynes et al. [8] go on to describe an extension of the coroutine paradigm entitled the Dahl-Hoare coroutine [1]. This coroutine has an additional facility, the *detach* statement, which transfers control back to the point where a group of coroutines was entered with a *call*. Owing to the lack of a control delimiter, the *call/cc* implementation requires each coroutine to have an additional local variable holding the caller continuation. The *call* statement grabs its continuation to provide the called coroutine with the caller continuation. Each *resume* conveys this information about the caller to the destination coroutine. Eventually, a *detach* invokes this caller continuation.

In our version, since we have already identified the caller context with a prompt, a *detach* merely **aborts** to this prompt. Thus a Dahl-Hoare coroutine is defined as:

```
(extend-syntax (coroutine resume detach)
  [(coroutine x e ...)
   (letrec ([LCS (lambda (x) e ...)]
            [resume (lambda (c v) (control (sigma (LCS) (c v))))]
            [detach (lambda (v) (control (sigma (LCS) v)))])
    (lambda (v) (LCS v)))]).
```

Here we have incorporated an update to the local control state to take place during a *detach*, so that any future invocation of the coroutine starts computation at the point left off by *detach*.

6.2 Engines

An engine performs a computation subject to timed preemption [6]. It is run with three arguments: a number of time units or *ticks*, a *success* procedure and a *failure* procedure. If the computation finishes within the given time, the *success* procedure is applied to the result of the computation and the remaining ticks; otherwise, the *failure* procedure is applied to a new engine that represents the preempted part of the computation. In either case, the procedure application happens in the call-context of the engine's invocation. Engines are useful abstractions for realizing time-sharing systems, simulating non-deterministic parallelism and distributed systems, and making time comparisons between different algorithms.

Haynes and Friedman [6] postulate a procedure for converting a thunk into an engine. An engine can be run to completion by supplying it a *failure* procedure that repeatedly runs the failed engine, and a *success* procedure that returns the value of the engine computation. Their implementation is native.

Dybvig and Hieb [2] present a simple implementation of engines with *abortive* continuations. They assume the presence of a global clock or an interruptable timer, either available primitively, or created using syntactic extensions for **lambda** and **set!** that consume ticks when evaluated. The clock's internal state holds (1) the number of remaining ticks, and (2) an interrupt handler which is invoked when the ticks run out. The user can modify both of the clock's ticks and its handler. In addition, the user can stop the clock: this returns the number of remaining ticks.

In the Dybvig-Hieb model, an engine computation captures its entry-continuation with a *call/cc*. If the engine completes within the time allotted, control passes to the entry-point of the engine, after which the *success* action is applied to the result of the engine computation and the ticks left. Should the engine fail, the interrupt handler captures the continuation at the point of failure and makes it into a new engine. Once again, control passes to the entry-point, and the *failure* action is applied to the new engine.

The entry-point continuation is held in a variable common to every engine. The currently active engine uses this variable to guarantee that either its *success* or *failure* actions takes place in its calling context. The continuation captured by a failing engine effectively represents the rest of the engine computation, since it always contains an abortive invocation of the *current* entry-point continuation. The problem with the Dybvig-Hieb engine model is the same as the one with coroutines implemented with *call/cc*: both of them use longer continuations than are needed to identify the rest of the engine or the coroutine computation. To offset this, they require yet another continuation call to truncate the computation at the appropriate point.

Again, prompts and functional continuations offer a simpler model.[10] The engine computation is an independent program embedded in a prompt. A successful engine computation returns its result to the prompt. If the engine fails, on the other hand, the interrupt handler uses a *control*-application to capture the remaining engine computation as a functional continuation and returns this procedure to the prompt as the failed engine. The value obtained at the engine's prompt is thus either the result of the engine computation or a failed engine. The *success* or *failure* actions are taken accordingly.

As an additional benefit, the use of prompt and *control* permits a clean separation between the underlying computation of the engine and its interruptability. We may thus represent an engine with a thunk denoting its computation and a preemptable version: the engine proper. The procedure *thunk→engine* produces such a representation from a thunk. The procedure *engine→thunk* retrieves the uninterruptable thunk from such a representation. Calling this thunk has the same effect as the more expensive operation of running the engine with an infinite number of ticks. The

[10]We thank Bob Hieb for pointing out a serious flaw in our earlier engine model.

procedure *run-engine* runs the engine proper.

The following outlines a first attempt at defining *thunk→engine*:

```
(% (clock 'set ticks)
   (begin0 (thunk)
           (set! ticks (clock 'stop)))).
```

The procedure *thunk→engine* contains actions for setting the clock to the requisite ticks, running the thunk, and returning its value, either a success or an interrupted computation, after stopping the clock. When the clock runs out of ticks, it invokes the interrupt handler. We set the handler as a procedure that captures the rest of the engine computation, packages it as an interrupt message, and sends it to the engine's prompt:

```
(clock 'set-handler
  (lambda () (control0 (lambda (rest-of-comp)
                          (list 'engine-interrupt rest-of-comp))))).
```

A closer look reveals some problems. When the engine computation is interrupted, the functional continuation denoting the rest of the engine includes as its last action the stopping of the clock. The inclusion of the stop-action will cause problems whether used as an engine or a thunk. An engine formed from this continuation could stop the clock, and the value of the remaining ticks supplied to its *success* procedure, obtained by a second stopping of the clock, is always zero. Even worse, if the functional continuation runs as a simple thunk inside another active engine, stopping the clock implies that the rest of the engine can continue uninterrupted beyond its allotted time.

Clearly, the rest of the engine as captured by the interrupt handler should not extend to the stopping of the clock. We modify the offending expression by enclosing the setting of the clock and the thunk invocation in a second prompt:

```
(% (begin0 (% (clock 'set ticks) (thunk))
           (set! ticks (clock 'stop)))).
```

This captures the right continuation if the interrupt occurs within the inner prompt. Unfortunately, it does not solve the problem for interrupts that occur just after the inner prompt-expression has returned.

As a remedy, we modify the two prompts such that any interrupt that occurs *between* them is ignored. This differential behavior is achieved by having the outer prompt ignore all interrupts and the inner one package its interrupts as failed engines:

```
(%-atomic0 (%-failed-engine (clock 'set ticks) (thunk))
           (set! ticks (clock 'stop))).
```

Both **%-atomic0** and **%-failed-engine** are simple syntactic variants of *run*. The form **%-atomic0** runs its subexpressions inside a prompt, and tests the result of its first subexpression:

```
(extend-syntax (%-atomic0)
  [(%-atomic0 e ...)
   (let ([result (% (begin0 e ...))])
    (record-case result
      ['engine-interrupt (rest-of-eng) (rest-of-eng)]
      [else result]))]).
```

If the result is an interrupt, it is ignored by invoking the interrupted continuation; otherwise, the result is returned.

The form **%-failed-engine** runs its subexpressions inside a prompt and tests their result:

```
(extend-syntax (%-failed-engine)
  [(%-failed-engine e ...)
   (let ([result (% e ...)])
    (record-case result
      ['engine-interrupt (rest-of-eng) (list 'failed-engine rest-of-eng)]
      [else result]))]).
```

If the result is an interrupt, it is packaged as a failed engine; otherwise the result is returned unchanged.

The final version of *thunk→engine* is in Figure 4. The engine produced by *thunk→engine* is a closure that upon application to a selection procedure yields either the encapsulated thunk or the engine proper. The inverse procedure *engine→thunk* simply extracts the thunk:

```
(define engine→thunk
  (lambda (eng)
    (eng (lambda (thunk eng-prop) thunk)))).
```

Running the engine is also straightforward. Instead of the thunk, we retrieve the engine proper and apply it to its arguments:

```
(define run-engine
  (lambda (eng ticks succ fail)
    ((eng (lambda (thunk eng-prop) eng-prop)) ticks succ fail))).
```

```
(define thunk→engine
  (lambda (thunk)
    (let ([eng-prop
           (lambda (ticks succ fail)
             (let ([ans (%-atomic0 (%-failed-engine (clock 'set ticks)
                                                    (thunk))
                                   (set! ([ticks (clock 'stop)]) 'any))])
               (record-case ans
                 ['failed-engine (rest-of-eng)
                                 (fail (thunk→engine rest-of-eng))]
                 [else (succ ans ticks)])))])
      (lambda (return-either)
        (return-either thunk eng-prop))))).
```

Figure 4: The procedure *thunk→ engine*

Finally, we can implement a procedure *engine-return* that stops an engine computation. Since the computation is embedded in a prompt, the procedure is a simple **abort**-statement:

(**define** *engine-return* (**lambda** (*v*) (**abort** *v*))).

6.3 Streams

Beyond being helpful tools for the implementation of higher-level control abstractions, functional continuations and prompts also enhance the expressiveness of the underlying language. Both are well-suited tools for combining the imperative-program-schema and the stream-programming-paradigm.

Consider a multiary tree, which is either a data-leaf or a list of multiary trees. An inorder traversal of such a tree can be specified by a simple recursive algorithm. If the tree is a leaf, enumerate it; otherwise, apply the algorithm to all the elements in the list of subtrees from left to right. Abstracting over the particular enumeration procedure to be applied at each leaf, the tree walk can be written as a program schema:

```
(define enumerate
  (lambda (leaf-proc!)
    (lambda (tree)
      (iterate E ([tree tree])
        (if (leaf? tree) (leaf-proc! tree) (for-each E tree)))))).
```

The procedure for-each applies the recursive procedure E to each element of the list *tree* for the effect only. Its result is unspecified.

Based on this program schema, we can derive a variety of different tree walks by instantiating the *leaf-proc!* procedure. For example, an inorder print is simply:

```
(define inorder-print (enumerate print)).
```

Similarly, an updating procedure that alters the information of each leaf according to some *update!* procedure can be written as:

```
(define tree-update (enumerate update!)).
```

More interestingly, we can think of a tree walk, *enumerate-stream*, that returns a leaf at a time *and* a zero-ary procedure—a thunk—for enumerating the rest of the tree when appropriate. Such a pair is called a *stream*, and can be created using a lazy-cons, **stream-cons**:

```
(extend-syntax (stream-cons)
  [(stream-cons a d) (cons a (lambda () d))]).
```

Enumeration streams are useful in situations where the elements of a tree are successively fed into a different computation, or where the information in the rest of the tree may not be needed. This becomes particularly apt if the tree is large or expensive to generate.

With prompts and functional continuations, the procedure *enumerate-stream* is yet another instantiation of the program schema *enumerate*. The enumeration step should immediately return a stream consisting of a leaf and a thunk to carry on the rest of the enumeration. The rest of the enumeration is represented by the continuation of the enumeration algorithm, i.e., the portion of the control stack between the top and the call-point of *enumerate-stream*. This partial continuation can easily be captured by placing the computation in a prompt and by using *control0* to get a hold of the functional continuation of the leaf enumeration. For all future invocations, the thunk generated by *control0* must compute inside of a prompt in order to delimit further calls to *control0*. Putting all this together, we have:

```
(define enumerate-stream
  (lambda (tree)
    (% ((enumerate
          (lambda (leaf)
            (control0 (lambda (rest)
                        (stream-cons leaf (run rest))))))
        tree)))).
```

7 A hierarchy of *controls* and *runs*

Unfortunately, multiple uses of *control* and *run* have the potential of interfering with each other, making it impossible to mix *control* and *run* with high-level abstractions or high-level abstractions with each other. For a simple example, consider the expression:

(**catch** 'k (list (% (add1 (**throw** 'k 6))))).

Instead of the constant 6, our macro-implementation of **catch** and **throw** produces a list containing a throw-record, which is clearly not intended. Similarly, a **catch**-expression in the body of an engine can void the engine's clock interrupt. When the interrupt occurs in the dynamic extent of a **catch**-expression, the wrong piece of context is identified as the rest of the engine, and, even worse, the interrupted engine is aborted to the prompt of the **catch**-expression.

The interference between multiple uses of *control* and *run* is obviously due to the spoiling of the correspondence between a particular pairing of *control* and *run*. The most natural solution calls for matching pairs of *control* and *run*. Such matching pairs should interact with each other and possibly ignore the intervention of other control operations. Given such facilities, we could match up **catch**-prompts with **throw**-*controls* and engine-prompts with engine-*controls*, avoiding the above problems. However, total independence between all pairs of *control* and *run* is not always desirable. The engine-prompts, for example, should intercept all *control*-operations as otherwise a timer would be running without an engine being active. Similarly, a **catch**-expression can also be conceived of as an independent task that either returns normally or terminates with a **throw**, but prevents other *control* actions.

We propose a *hierarchy* with different *levels* of *control* and *run* pairs such that a prompt serves as control delimiter to all *controls* of and above its level. The hierarchy is definable in terms of the original *control* and *run*. For clarity, we refer to the originals as *system-control* and *system-run* and to some pair in the hierarchy as *leveled control* and *leveled run*. Levels are indicated by non-negative integers: the lowest level is 0, and a larger integer denotes a higher level. Pairs on the same level interact as usual; pairs at different levels interact in some convenient and meaningful way.

To unravel the strands of control, we use a message-passing protocol for communicating between *leveled control* and *leveled run*. Since the prompt closest to a *control*-application is generally not of the same level and the captured functional continuation does not represent the entire relevant context, a *leveled control* switches to the nearest prompt with an appropriate package of information and leaves to the prompt what to do next. The message consists of the *control* level, the *control*-argument, and the continuation. The procedure that generates an appropriate version of *control* for a given level number is:

```
(define make-control
  (lambda (lvl#)
    (lambda (f)
      (system-control (lambda (k) (list 'control/run lvl# f k)))))).
```

In order to intercept all possible *control*-operations, a *leveled run* invokes its argument in *system-run*, guaranteeing the return of the computation to the *leveled run*-operation. If this *leveled run* receives a *control*-package, it compares its own level with that of the *control*-package and decides whether the *control*-application be performed immediately, or whether the package be forwarded to the next prompt. Given a level number, the procedure *make-run* is used to produce the corresponding *leveled run*:

```
(define make-run
  (lambda (level)
    (rec run
      (lambda (th)
        (let ([v (system-run th)])
          (record-case v
            ['control/run (control-level f k)
                          (if (>= control-level level)
                              (run (lambda () (f k)))
                              ((make-control control-level)
                               (lambda (g)
                                 (run (lambda ()
                                        (f (lambda (x) (g (k x)))))))))]
            [else v])))))).
```

If the *control*-package's level is at or above the current *run*-level, the application of the *control*-argument to the corresponding continuation takes place. If, on the other hand, the package's number is of a lower level, more of the context has to be captured before the *control*-argument can be applied. In either case, the continuation is supplied to the *control*-argument in a prompt of the current level to account for further *control* actions.

We assign level numbers to the various control operators in the abstractions for engines and **catch** and **throw**, etc., as follows. First, *base-run* is the bottommost *run* in the hierarchy, and the procedure for *resetting* the system corresponds to the bottommost *control*. Second, since an engine *control* used to interrupt a failing engine should be at a lower level than any other *run* operators, the engine's *control* and *run* are at level 1. Third, **catch** and **throw** are available at levels above the *run* used for running an engine, and are given level 2. Finally, we re-use the names *run* and *control* for the pair at level 3 in the hierarchy.

In order to make the control hierarchy safe, the procedures *make-control* and *make-run* as well as all leveled versions of *control* and *run* except the top-level ones must be hidden from the user. The availability of the user-level *control* and *run* is sufficient for building further hierarchies on top of the given one. Indeed, the solution is flexible enough to allow different *sub*-hierarchies on different levels of a given hierarchy.

Of course, our proposed scenario is not the only feasible one. We could equally well argue that different levels should not interfere with each other at all, or that the action taken according to the three different cases should be parameters of the *make-run* procedure. It would then be possible to use **catch** for exiting engines, with the provision that **catch**'s *control*-application turns off the engine's clock. The important point is not *which* scenario we choose to implement but that *every* scenario is realizable on top of the simple *control*- and *run*-operators.

8 Conclusion

In the preceding sections, we have demonstrated that the concept of a control delimiter has a variety of interesting applications. First, together with **abort**, it provides the appropriate terminology for explaining different alternatives of Lisp control constructs. If it is available within Lisp, it can be used to implement the best alternative for a particular situation. Second, and more importantly, the availability of *run* in a higher-order language with *call/cc* or *control* is the basis for embedding stack-based control structures easily and faithfully. Finally, the operator *run* facilitates the macro-implementations of improved versions of existing higher-order control abstractions in Scheme-like programming languages. In short, we believe that for systems and application languages, *run* provides an important low-level control operator.

In our analysis we ignored the connection of control delimiters to parallel and distributed programming. Our own motivation for *run* stems partly from theory [4] and partly from concern about unrestricted control action, but Stoy and Strachey [17] introduced first-order *run* as an operating system primitive for executing independent sub-processes. It is therefore natural to ask whether *run* is the appropriate control delimiter for *parallel* versions of higher-order programming languages with continuation-based control structures. We suggest this as a topic for future research.

Acknowledgment

We are grateful to Bruce Duba for a discussion on implementation strategies for control operators, to Dan Friedman for critical comments on early drafts, to Bob Hieb for suggesting several important improvements, and to the referees for their detailed readings.

References

1. O.J. Dahl and C.A.R. Hoare. Hierarchical program structure. In O.-J. Dahl, E. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, Academic Press, 1972.

2. R.K. Dybvig and R. Hieb. Engines from continuations. *Journal of Computer Languages* (Pergamon Press), 14(2), 1989.

3. M. Felleisen. A calculus for assignments in higher-order languages. *Proc. 14th ACM Symposium on Principles of Programming Languages*, 314–325, 1987.

4. M. Felleisen. The theory and practice of first-class prompts. *Proc. 15th ACM Symposium on Principles of Programming Languages*, 180–190, 1988.

5. M. Felleisen, M. Wand, D.P. Friedman, and B.F. Duba. Abstract continuations: a mathematical semantics for handling full functional jumps. *Proc. Conference on Lisp and Functional Programming*, 52–62, 1988.

6. C.T. Haynes and D.P. Friedman. Abstracting timed preemption with engines. *Journal of Computer Languages* (Pergamon Press), 12(2):109–121, 1987.

7. C.T. Haynes and D.P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):245–254, 1987.

8. 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.

9. G.F. Johnson and D. Duggan. Stores and partial continuations as first-class objects in a language and its environment. *Proc. 15th ACM Symposium on Principles of Programming Languages*, 158–168, 1988.

10. E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, 1986.

11. P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

12. C.D. Marlin. *Coroutines—A Programming Methodology, a Language Design and an Implementation.* Volume 95 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, 1980.

13. J. McCarthy et al. *Lisp 1.5 Programmer's Manual.* The MIT Press, Cambridge, 2 edition, 1965.

14. J. Rees and W. Clinger. The revised^3 report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

15. J.C. Reynolds. Gedanken—a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, 1970.

16. G.L. Steele Jr. *Common Lisp—The Language.* Digital Press, 1984.

17. J.E. Stoy and C. Strachey. OS6: an operating system for a small computer. *Comp. J.*, 15(2):117–124; 195–203, 1972.

18. G.J. Sussman and G.L. Steele Jr. *Scheme: An interpreter for the extended lambda calculus.* Memo 349, MIT AI Lab, 1975.