Engines from Continuations

R. Kent Dybvig and Robert Hieb

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

July 21, 1988

Abstract

Engines provide the means for a computation to be run for a limited period of time, interrupted if it does not complete in that time, and later restarted from the point of interruption. Previous work on engines demonstrated that engines can be implemented by defining a new interpreter to support them. This article demonstrates that engines may be defined in terms of continuations and timer interrupts and thereby incorporated into an existing language implementation. The article extends the engine mechanism to solve several problems with nestable engines, and demonstrates that the extended mechanism can be implemented in terms of continuations as well.

Engines   Continuations   Timed Preemption

This material is based on work supported by the National Science Foundation under grant number CCR-8803432 and by Sandia National Laboratories under contract number 06-06211.

1. Introduction

Programming languages typically do not provide any means for a computation to be run for a limited period of time, interrupted if it does not complete in that time, and later restarted from the point of interruption. The means to interrupt and restart programs running as processes is provided by most operating systems, but this ability usually does not extend to the subprogram level. The *engine* mechanism introduced by Haynes and Friedman provides this ability in a clean and concise manner [5, 6, 2]. Haynes and Friedman described how unnestable engines (where only one engine can be active at a time) can be implemented by providing a new interpreter that supports engines. They also described a nestable engine mechanism that can be implemented in terms of unnestable engines.

In this article we demonstrate that engines can be implemented in terms of first-class continuations and timer interrupts and thereby incorporated into an existing language implementation. Consequently, it is not necessary to write a new interpreter or compiler to support engines as long as the host language supports first-class continuations and timer interrupts. We also show that the timer interrupt mechanism need not be primitive in the language if provisions for syntactic extensions exist. We further demonstrate that nestable engines as described by Haynes and Friedman can be implemented in a similar manner. We point out some of the problems inherent in their model and present an extended nestable engine mechanism, along with its implementation, that solves these problems.

The algorithms and examples presented in this paper are coded in Scheme. Section 2 contains a brief discussion of the features of Scheme which are essential to our presentation. This is followed, in Section 3, by a review of the engine mechanism and some examples of its use. In Section 4, we show how to construct a timer interrupt mechanism in Scheme and demonstrate how engines may be constructed from continuations with timer interrupts. We proceed with a discussion of nestable engines in Section 5 and follow this with our proposal for a new nestable engine mechanism and its implementation in Section 6. In Section 7, we conclude with a few remarks about the benefits of our method for implementing engines and some suggestions for further study.

2. Scheme

The algorithms and examples presented in this paper are coded in Scheme, a lexically-scoped dialect of Lisp that supports first-class continuations [10]. Two features of Scheme aside from continuations play an important role in this article: first-class procedures and proper treatment of tail recursion. In this section we describe first-class procedures, proper treatment of tail recursion, and continuations briefly. More detail on these subjects can be found in [2].

In Scheme, procedures are created by evaluation of **lambda** expressions. For example, the expression (**lambda** (x) x) evaluates to an identity procedure. A procedure created from a **lambda** expression may be applied directly; the expression ((**lambda** (x) x) 0) evaluates to 0. Procedures may be assigned to top-level variables using the **define** syntactic form. For example, the following definition binds the variable *identity* to the identity procedure:

```
(define identity
  (lambda (x) x))
```

After this definition, the expression (*identity* 0) also evaluates to 0.

Furthermore, it is possible to pass procedures as arguments to other procedures, return them as results, or store them indefinitely before applying them. In fact, procedures are treated like any other data object, which is why Scheme procedures are said to be *first-class*. What makes the first-class status of procedures interesting and useful is that when a procedure is applied, its body is evaluated in the lexical environment in effect when the procedure was created. In essence, a procedure created from a **lambda** expression "remembers" the lexical bindings visible where the **lambda** expression appears in the code.

For example, we can define a procedure that creates a unary operator from a binary operator and one operand:

```
(define binary->unary
  (lambda (operator operand1)
    (lambda (operand2)
      (operator operand1 operand2))))
```

This procedure, which we have named *binary->unary*, expects a procedure of two arguments (the binary operator) and a value (the first operand) as its arguments, and it returns a procedure of one argument as its value. When the procedure returned by *binary->unary* is applied to a value (the second operand), it applies the binary operator to the two operands to obtain its value. Unary minus may now be defined as (*binary->unary* $-$ 0), and the *1+* procedure may be defined as (*binary->unary* $+$ 1).

It is important to realize that the locations of the visible lexical bindings are retained by a procedure, not merely the values. This matters only if a side-effect to one of the lexically visible variables occurs. For example, the following code segment creates two procedures within the lexical scope of the variable x. The procedure *get-x* returns the current value of x, while the procedure *put-x!* gives x a new value.

```
(define get-x)
(define put-x!)
(let ([x 0])
  (set! get-x
    (lambda ()
      x))
  (set! put-x!
    (lambda (v)
      (set! x v))))
```

The **define** expressions appearing outside of the **let** establish top-level bindings for the variables *get-x* and *put-x!*; these variables are given values, using **set!**, inside the **let**. Because *get-x* and *put-x!* are defined within the **let** for *x*, they can communicate through *x*. And since nothing else appears within the **let** expression, only *get-x* and *put-x!* can reference or assign *x*. Thus, we have created two procedures that share a private local variable. First-class procedures, then, are useful for writing modular code, and they may be used to program in an object-oriented style.

Proper treatment of tail recursion allows iteration to be expressed as recursion, thereby eliminating the need to use special looping constructs to perform iteration. Recursion is the direct or indirect invocation of a procedure by itself. Tail recursion occurs when the recursive call appears in a context where there is nothing left to do, *i.e.*, the result of the recursive call is returned as the result of the caller.

The following two definitions for *length* both compute the length of a list; the first version is not tail-recursive while the second version is tail-recursive:

```
(define length1
  (lambda (ls)
    (if (null? ls)
      0
      (+ (length1 (cdr ls)) 1))))

(define length2
  (lambda (ls n)
    (if (null? ls)
      n
      (length2 (cdr ls) (+ n 1)))))
```

In the latter version, a second argument is needed to count the elements in the list as it is traversed, so we might define *length* to be (**lambda** (*ls*) (*length*$_2$ *ls* 0)). In the definition for *length*$_1$, the recursive call appears as an argument to + and must be evaluated before + is applied. In the definition for *length*$_2$, the recursive call appears where there is nothing left to do but return and is therefore a tail-recursive call.

Handling tail recursion properly means that tail-recursive calls cannot build up a stack of activation records, as is typically done for recursive calls. In the case of directly tail-recursive procedures, the code generated by a good compiler is equivalent to the corresponding code for iterative constructs such as while loops provided by many languages. One important benefit of the treatment of tail-recursion as iteration is that assignments are not needed to update the induction variables of a loop defined in terms of tail recursion, so it is possible for a program to cycle an indefinite number of times before terminating without performing any side-effects. We take advantage of the proper treatment of tail recursion in code given in this paper, and we also show how proper treatment of tail recursion is maintained by our implementation of engines.

A continuation is an abstract entity that represents the remainder of a computation from a given point. In Scheme, continuations are made available as procedure objects via the procedure *call/cc*. The argument to *call/cc* is itself a procedure of one argument, to which the continuation is passed. When a continuation created by *call/cc* is applied to a value, it returns control of the program back to the point where the *call/cc* occurred, as if the value of the application of *call/cc* were the value passed to the continuation. Thus, *call/cc* may be used for nonlocal exits, as in the following procedure that computes the product of the elements of a list:

```
(define product
  (lambda (ls)
    (call/cc
      (lambda (quit)
        (letrec ([f (lambda (ls)
                         (if (null? ls)
                             1
                             (if (= (car ls) 0)
                                 (quit 0)
                                 (* (car ls) (f (cdr ls))))))])
          (f ls))))))
```

The continuation bound to *quit* is used to exit immediately from *product*, without completing the loop and before performing any multiplications, if a zero element is found. If a zero element is not found, the multiplications are performed and the result is returned normally.

Because continuations in Scheme are procedures, they are first-class data objects. This means that a continuation may be applied when the code that created it is not active, *i.e.*, when the program has dynamically returned from the corresponding *call/cc* application. In this case, applying the continuation still returns control to the point where the *call/cc* occurred, and processing continues as if the program had never returned from that point. This feature may be used to implement a number of interesting control structures, including non-blind backtracking [11, 4], coroutines [8], and engines.

3. Unnestable Engines

An *engine* is an object embodying a particular computation. The progress made by the computation is determined by the amount of *fuel* provided to the engine. The engine mechanism proposed by Haynes and Friedman requires two global procedures: *make-engine* and *engine-return*. The procedure *make-engine* creates an engine from a procedure of no arguments. This procedure specifies the computation to be performed by the engine. An engine is a procedure of three arguments:

- *ticks*: a positive integer that specifies the amount of "fuel" the engine will be allowed to consume,
- *return*: a procedure that specifies what to do if the engine returns before the fuel is consumed, and
- *expire*: a procedure that specifies what to do if the fuel "runs out" before the engine computation returns.

The *return* argument is passed a value and the ticks remaining if the engine computation returns; the *expire* argument is passed a new engine capable of continuing the engine's computation if the fuel is consumed before the engine computation returns.

An engine computation can return only by invoking the procedure *engine-return*. The procedure *engine-return* takes one argument; this argument is passed as the first argument to the *return* procedure. The second argument to the *return* procedure is the number of ticks remaining when *engine-return* is invoked. It is an error for the procedure originally passed to *make-engine* to return directly, *i.e.*, without calling *engine-return*. This is not a particularly meaningful restriction, since it is straightforward to define the procedure *make-simple-engine*, which creates an engine that implicitly calls *engine-return* when the computation has finished:

```
(define make-simple-engine
  (lambda (proc)
    (make-engine (lambda () (engine-return (proc))))))
```

A *tick* of fuel refers to no particular unit of computation; however, Haynes and Friedman placed the following constraints on the amount of computation associated with a tick: "(1) a larger tick count is associated with a larger expected amount of computation (in a statistical sense) and (2) unbounded real time is associated with an unbounded number of ticks (any looping computation must consume ticks)" [6]. It is possible within these constraints for an implementation to associate a different amount of computation at different times with the same number of ticks; hence, computations involving engines may be nondeterministic. Counting procedure calls is a straightforward way to provide a tick counter in Scheme, since procedure calls provide the basis for all Scheme control structures.

Passing an engine a single tick ensures the finest granularity and the fairest distribution of processing time. It may be desirable to provide a larger number of ticks (for efficiency) or to provide a random number of ticks in each instance. The latter allows nondeterministic behavior even in a system where ticks are discrete units of computation.

One common use for engines is to implement a multitasking operating system scheduler. The kernel protection is provided by Scheme's lexical scoping and first-class procedures, following

```
(define kernel)
(letrec
   ([ready-queue (queue)]
    [start
     (lambda (proc)
        (enqueue (make-engine (lambda () (proc trap)))
                 ready-queue))]
    [restart
     (lambda (k v)
        (enqueue (make-engine (lambda () (k v)))
                 ready-queue))]
    [trap
     (lambda (msg arg)
        (call/cc
           (lambda (k)
              (engine-return
                 (lambda ()
                    (case msg
                       [uninterruptible (restart k (arg))]
                       [start-process (start arg)
                                      (restart k #f)]
                       [stop-process #f])))))))]
    [dispatch
     (lambda ()
        (if (empty-queue? ready-queue)
            'finished
            ((dequeue ready-queue)
             (time-slice)
             (lambda (trap-handler ticks)
                (trap-handler)
                (dispatch))
             (lambda (engine)
                (enqueue engine ready-queue)
                (dispatch)))))])
   (set! kernel
      (lambda (proc)
         (start proc)
         (dispatch))))
```

Figure 1. An operating system built with engines.

Wand [12], and processes are represented by engines. A process running in an engine that requires uninterruptible access to kernel services passes a request to *engine-return*; the kernel acts on the request—with no engine running, and hence without interruption. A process that must be resumed after the request is serviced must arrange to obtain its continuation, which can then be made into an engine and later restarted. The simple operating system kernel in Figure 1 illustrates this mechanism. This operating system maintains a queue[1] of ready processes (engines) that it cycles through in a round-robin fashion, giving processor time to each process in turn. The system kernel provides three services: *start-process*, *stop-process*, and *uninterruptible.* Each process receives a *trap* argument that it must invoke with a message and an argument in order to take advantage of these services. The *trap* procedure obtains the current continuation before invoking *engine-return* to return from the engine. The procedure returned from the engine looks at the message and performs the requested service.

The procedure *time-slice* determines the number of ticks given to a process each time it runs. Depending upon the desired effect, *time-slice* may return the same number of ticks each time it is called, or it may generate a random number of ticks. For example, the following version of *time-slice* returns a positive integer less than or equal to 100:

```
(define time-slice
  (lambda ()
    (+ (random 100) 1)))
```

In the case of *uninterruptible* and *start-process*, the continuation of the *trap* is made into a new engine so that the process requesting the service can continue. The following code illustrates the use of the operating system and the services it provides:

```
(define amoeba
  (lambda (generation)
    (lambda (trap)
      (trap 'uninterruptible (lambda () (write generation) (newline)))
      (if (= (random 2) 0)
        (trap 'stop-process #f)
        (let ([split-amoeba (amoeba (+ generation 1))])
          (trap 'start-process split-amoeba)
          (split-amoeba trap))))))

(kernel (amoeba 0))
```

An "amoeba" is a process controlled by *kernel.* Each amoeba has a one in two chance of surviving and reproducing. The first amoeba is assigned "generation" 0 by the call to *kernel.* If it survives, it will be promoted to generation 1 and "split" to create two new amoebas, which in turn may each create two more amoebas assigned generation 2, and so on. Each amoeba first prints its generation number, using *uninterruptible* to ensure that the number and the ensuing newline are printed together. The numbers may or may not be printed in the sequence 0, 1, 1, *etc.*, depending on the choice of *time-slice* in the kernel.

[1] We assume the basic queue manipulation procedures *queue*, which builds an empty queue, *empty-queue?*, which tests whether a queue is empty, *enqueue*, which adds an object to the queue, and *dequeue*, which removes and returns the next object from the queue. We assume that queue operations are performed in the normal first-in, first-out order.

4. Engines from Continuations

We now turn to an implementation of engines in terms of first-class continuations. The implementation provides a realistic demonstration of the power of first-class continuations as well as demonstrating that engines need not be provided as primitive in a language that supports first-class continuations.

Any multitasking primitive must have the ability to do two things: (1) preempt a running process after a given amount of computation, and (2) obtain the control state of the process so that it can be later resumed. The first ability is provided by a primitive timer interrupt mechanism in some Lisp and Scheme systems; we show here how to construct a timer interrupt mechanism if one is not available. First-class continuations give us the second ability.

Our timer system defines the procedures *start-timer*, *stop-timer*, and *decrement-timer*, which can be described operationally as follows:

- (*start-timer ticks handler*): *start-timer* initializes the timer to *ticks* ticks and installs *handler* as the procedure to be invoked (without arguments) when the timer expires.
- (*stop-timer*): *stop-timer* resets the timer and returns the number of ticks remaining.
- (*decrement-timer*): If the timer is on (timer is nonzero) *decrement-timer* decrements the timer by one tick. If the timer has reached zero, *decrement-timer* invokes the saved handler. If the timer is not on, or if the timer has not reached zero, *decrement-timer* simply returns.

The following code implements the timer system:

```
(define start-timer)
(define stop-timer)
(define decrement-timer)
(let ([clock 0] [handler #f])
  (set! start-timer
    (lambda (ticks new-handler)
      (set! handler new-handler)
      (set! clock ticks)))
  (set! stop-timer
    (lambda ()
      (let ([time-left clock])
        (set! clock 0)
        time-left)))
  (set! decrement-timer
    (lambda ()
      (when (> clock 0)
        (set! clock (- clock 1))
        (when (= clock 0) (handler))))))
```

Using the timer system requires inserting calls to *decrement-timer* in appropriate places. Consuming a timer tick on entry to a procedure usually provides a sufficient level of granularity (in Scheme, procedure calls are required for all iteration and recursion). This can be accomplished by using **extend-syntax** [9, 2] to supply **timed-lambda**, a version of **lambda** that automatically invokes *decrement-timer* before executing the expressions in its body:

```
(extend-syntax (timed-lambda)
  [(timed-lambda parameters exp ...)
   (lambda parameters (decrement-timer) exp ...)])
```

In some systems it is possible to redefine **lambda** itself to be **timed-lambda**; the timer mechanism would thus be invisible to the programmer [3, 1]. In our examples we will assume that **lambda** implicitly decrements the timer. Using syntactic transformations to implement the timer means that independently compiled code, including system code for built-in library procedures, will not decrement the timer. A built-in mechanism that would not suffer from this deficiency can be modeled after the syntactic mechanism. A single instruction (increment with trap on overflow) suffices on some machines.

It may also be useful to provide other "timed" special forms. For testing operating systems or synchronization primitives, it is useful to have assignments consume ticks so that we do not inadvertently provide a safe test-and-set operation:

```
(extend-syntax (timed-set!)
  [(timed-set! id exp)
   (let ([v exp])
     (decrement-timer)
     (set! id v))])
```

We now turn to the implementation of unnestable engines in terms of continuations and the timer interrupt mechanism. We use *call/cc* in two places in the engine implementation: (1) to obtain the continuation of the computation that invokes the engine so that we can return to that continuation when the engine computation returns or the timer expires, and (2) to obtain the continuation of the engine computation when the timer expires so that we can return to this computation if the newly created engine is subsequently run.

The code for the engine system, shown in Figure 2, defines the procedures *make-engine* and *engine-return*. The state of the engine system is contained in three variables local to the engine system, *active?*, *do-return*, and *do-expire*. The *active?* variable is a flag used to catch attempts to invoke an engine from within an engine, *i.e.*, an attempt to nest engines, and to catch attempts to invoke *engine-return* when no engine is running. It is set to true when an engine is running and to false otherwise. When an engine is started, the engine assigns to *do-return* and *do-expire* procedures that, when invoked, return to the continuation of the engine's caller to invoke the *return* or *expire* procedures. The engine starts (or restarts) the computation by invoking the procedure passed as an argument to *make-engine* with the specified number of ticks. The ticks and the private procedure *timer-handler* are then used to start the timer.

Suppose that the timer expires before the engine returns. The procedure *timer-handler* is then invoked. It initiates a call to *start-timer*, but obtains the ticks by calling *call/cc* with *do-expire*.

```
(define make-engine)
(define engine-return)
(let ([active? #f] [do-return #f] [do-expire #f])
  (letrec
    ([timer-handler
      (lambda ()
        (start-timer (call/cc do-expire) timer-handler))]
     [new-engine
      (lambda (resume)
        (lambda (ticks return expire)
          (if active?
              (error 'engine "attempt to nest engines")
              (set! active? #t))
          ((call/cc
             (lambda (escape)
               (set! do-return
                 (lambda (value ticks)
                   (set! active? #f)
                   (escape (lambda () (return value ticks)))))
               (set! do-expire
                 (lambda (resume)
                   (set! active? #f)
                   (escape (lambda () (expire (new-engine resume))))))
               (resume ticks))))))])
    (set! make-engine
      (lambda (proc)
        (new-engine
          (lambda (ticks)
            (start-timer ticks timer-handler)
            (proc)
            (error 'engine "invalid completion"))))))
  (set! engine-return
    (lambda (value)
      (if active?
          (let ([ticks (stop-timer)])
            (do-return value ticks))
          (error 'engine-return "no engine running")))))
```

Figure 2. Engines implemented with continuations.

Consequently, *do-expire* is called with a continuation that, if invoked, will restart the timer and continue the interrupted computation. The *do-expire* procedure creates a new engine from this continuation and arranges for the engine's *expire* procedure to be invoked with the new engine in the correct continuation.

If, on the other hand, the engine returns (via *engine-return*) before the timer expires, *engine-return* stops the timer and passes the value and the ticks remaining to *do-return*; *do-return* arranges for the engine's *return* procedure to be invoked with the value and ticks in the correct continuation.

There are three subtle aspects to this code. The first concerns the method used to start the timer when an engine is invoked. The code would apparently be simplified by letting *new-engine* start the timer before it resumes (or initiates) the engine computation, instead of passing the ticks to the computation and letting it start the timer. However, starting the timer within the process means that ticks will not be consumed prematurely. If the engine system itself consumes fuel, then an engine provided with a small amount of fuel may not progress toward completion. (It may, in fact, make "negative progress." That is, an engine run with a small amount of fuel may produce a new engine that requires an even larger amount of fuel to finish the computation—something we discovered the hard way.) If the software timer described above is used, this problem can also be avoided by compiling the engine-making code using the untimed version of **lambda**.

Second, it appears that the body of *engine-return* could be written without the **let**, *i.e.*, (*do-return value* (*stop-timer*)). This would be incorrect, since if the subexpressions of this application were evaluated in a left-to-right fashion, the value of *do-return* would be obtained while the timer is running. If the timer were to expire just after the value of *do-return* is obtained, when the resulting engine runs the result would not be passed to the correct *return* procedure and continuation. The solution is to never manipulate the variables that contain the state of the engine while the timer is running. Thus the timer must be stopped before the variable *do-return* is accessed.

The third subtlety concerns the procedures created by *do-return* and *do-expire* and subsequently applied by the continuation of the *call/cc* application. It may appear that *do-return* could first invoke the engine's *return* procedure, then pass the result to the continuation (and similarly for *do-expire*) as follows:

(*escape* (*return value ticks*))

However, this would result in improper treatment of tail recursion. The problem is that the current continuation will not be replaced with the continuation stored in *escape* until the call to the *return* procedure completes. Consequently, both the continuation of the running engine and the continuation of the engine invocation can be retained for an indefinite period of time, when in fact the actual engine invocation may appear to be tail recursive. This is especially inappropriate because the engine interface encourages use of continuation-passing style and hence tail recursion. The operating system code in Figure 1 provides a good example of this, since the procedure *dispatch* invokes engines tail-recursively. Since the *return* and *expire* procedures it provides the engine contain tail-recursive calls to *dispatch*, it is crucial that the engine invocation itself be implemented tail-recursively. We do so by arranging for *do-return* and *do-expire* to escape from the continuation of the running engine before invoking the *return* or *expire* procedures. Since the continuation of the engine invocation is a procedure application, passing it a procedure of no arguments results in application of the procedure in the continuation of the engine invocation.

5. Nestable Engines

Engines can be used to simulate parallel evaluation of subexpressions, *e.g.*, in the implementation of **parallel-or**. The **parallel-or** syntactic form alternates processing time among its subexpressions and returns the value of the first expression to complete with a true (that is, in Scheme, any non-false) value. This is analogous to the Scheme **or** syntactic form, which processes its arguments in sequence rather than in parallel. **parallel-or** can be defined with engines as follows:

```
(extend-syntax (parallel-or)
  [(parallel-or e ...)
   (first-true (lambda () e) ...)])

(define first-true
  (lambda proc-list
    (letrec ([engines (queue)]
             [run (lambda ()
                    (and (not (empty-queue? engines))
                         ((dequeue engines)
                          1
                          (lambda (v t) (or v (run)))
                          (lambda (e) (enqueue e engines) (run)))))])
      (for-each (lambda (proc) (enqueue (make-simple-engine proc) engines))
                proc-list)
      (run))))
```

The syntactic definition for **parallel-or** expands into a call to *first-true* with each expression replaced by a **lambda** expression with no arguments and the expression as its body, effectively delaying its evaluation. The procedure *first-true* creates a queue of engines (using *make-simple-engine*) from the procedures it receives as arguments. It then loops, each time providing the first engine on the queue with a single tick of fuel. If the engine returns before the ticks expire and the value returned is true, this value is returned; otherwise the loop continues with the remaining engines. If the ticks expire before the engine returns, then the new engine is placed into the queue and the loop continues. The loop exits when one of the engines returns a true value or when the queue becomes empty, in which case the value returned is false.

Breadth-first tree searches are particularly easy to implement using **parallel-or**. However, breadth-first tree searches require nesting of **parallel-or** expressions; therefore it is important to be able to nest the engines upon which **parallel-or** is built.

Haynes and Friedman described an approach to nesting engines they termed *fair nesting*, in which each tick charged to an engine is charged as well to each of its ancestors. This is fair in the sense that an engine must subdivide its time among its offspring. In their description, they

```
(define make-engine)
(define engine-return)
(letrec
  ([new-engine
    (lambda (resume)
      (lambda (ticks return expire)
        ((call/cc (lambda (escape)
                    (run resume
                         (stop-timer)
                         ticks
                         (lambda (value ticks)
                           (escape (lambda () (return value ticks))))
                         (lambda (engine)
                           (escape (lambda () (expire engine))))))))))]
   [run
    (lambda (resume parent child return expire)
      (let ([ticks (if (and (active?) (< parent child)) parent child)])
        (push (- parent ticks) (- child ticks) return expire)
        (resume ticks)))]
   [go
    (lambda (ticks)
      (when (active?)
        (if (= ticks 0)
            (timer-handler)
            (start-timer ticks timer-handler))))]
   [do-return
    (lambda (value ticks)
      (pop (lambda (parent child return expire)
             (go (+ parent ticks))
             (return value (+ child ticks)))))]
   [do-expire #f])      ;EATME
  #f)      ;EATME
```

Figure 3a. A nestable engine implementation (continued in Figure 3b).

assume that the procedure *engine-return* returns from the most recently activated engine. They demonstrated that fair nesting can be implemented in terms of unnestable engines, by introducing a stack of pending computations. It is possible to use similar techniques to implement fair nesting directly.

Figures 3a and 3b contain an implementation of nestable engines in Scheme. The interface for this version of nestable engines is identical to that of unnestable engines. Its implementation, however, is considerably more complex. It is convenient to introduce some terminology to distinguish the various engines that may be simultaneously active. Engines running at the point an engine is started are that engine's *ancestors*. Engines started while an engine is running are that engine's *descendents*. The most immediate ancestor is an engine's *parent*; the most immediate descendent is its *child*.

When an engine is started, information about its invocation must be preserved, for the same reasons as for the single engine case. The procedure *new-engine* (see Figure 3a) packages the *return* and *expire* procedures into new procedures that will call the original procedures in the continuation in which the engine was invoked. However, since the state of any currently running engines must also be preserved, these escape procedures cannot be saved in private variables, as in the single engine code. Instead, they are saved on a private stack along with timing information.

The task of saving the state and starting the engine is assigned to the procedure *run*. If no engine is active or the number of ticks supplied to the new engine is smaller than the number of ticks currently on the clock, then the timer is restarted with the ticks supplied to the new engine. Otherwise, the timer is restarted with the ticks it had when interrupted. In either case, the residual ticks for the parent engine and the child engine are saved as part of the current engine state before the process is started.

When *engine-return* is called to stop an engine, it stops the timer and calls the procedure *do-return* with the value it received and the ticks left on the clock. Since the protocol specifies that *engine-return* always stops the most recently invoked engine, the appropriate *return* procedure, along with the necessary timing information for it and its parent, is in the top frame of the engine stack. *engine-return* restarts the timer with the number of ticks remaining for the parent engine (the number of ticks returned from *stop-timer* plus the number of parent ticks saved in the stack frame), and invokes the *return* procedure with the value and the current engine's remaining ticks (the number of ticks returned from *stop-timer* plus the number of child ticks saved in the stack frame).

The situation is more complicated when an engine expires because it ran out of ticks, since the engine with the fewest ticks is not necessarily the most recently started engine. The handler (*timer-handler*—see Figure 3b) passed to *start-timer* when an engine is started calls *do-expire* with the current continuation. *do-expire* must unwind the engine stack to find the engine whose ticks have run out. If the engine is subsequently restarted, all of its descendent engines must also be restarted. Thus, *do-expire* unwinds the engine stack by calling itself recursively with procedures that can resume the interrupted engines with the saved engine parameters. These procedures can be thought of as continuations that also include the current engine state. When *do-expire* finds a stack frame in which zero ticks remain for the current engine, it restarts the timer with the number of ticks remaining for the parent engine, makes the procedure passed to *do-expire* into an engine, and passes it to the saved *expire* procedure.

Using this model for nestable engines, we can nest **parallel-or** expressions. Because **parallel-or** never uses *engine-return* except to exit from the most recently activated engine, the restriction that *engine-return* returns from the most recently activated engine is unimportant.

```
(letrec       ;EATME
  ([new-engine #f]      ;EATME
   [do-expire
    (lambda (resume)
      (pop (lambda (parent child return expire)
             (if (> child 0)
                 (do-expire (lambda (ticks) (run resume ticks child return expire)))
                 (begin (go parent)
                        (expire (new-engine resume)))))))]
   [timer-handler (lambda () (go (call/cc do-expire)))]
   [stack '()]
   [push (lambda l (set! stack (cons l stack)))]
   [pop
    (lambda (handler)
      (if (null? stack)
          (error 'engine "attempt to return from inactive engine")
          (let ([top (car stack)])
            (set! stack (cdr stack))
            (apply handler top))))]
   [active? (lambda () (not (null? stack)))])
  (set! make-engine
    (lambda (proc)
      (new-engine
        (lambda (ticks)
          (go ticks)
          (proc)
          (error 'engine "invalid completion")))))
  (set! engine-return
    (lambda (value)
      (do-return value (stop-timer)))))
```

Figure 3b. A nestable engine implementation (continued from Figure 3a).

Furthermore, we can now recursively invoke the operating system of Figure 1 to model a self-virtualizing operating system. Assuming that a process on one level uses only the services provided by the operating system that directly controls it, the *engine-return* restriction is again unimportant.

However, we cannot easily model a multilevel operating system in which processes at one level can directly access services provided at levels below. All requests must be handled by the operating system at the current level and, if necessary, passed on to the levels below. This is unfortunate, but an even more serious problem is that other features implemented independently in terms of engines (such as **parallel-or**) would interfere with the operating system implementation. A trap from within a **parallel-or** expression would result in a return from the **parallel-or** engine, not from the process engine in which it is running.

There are actually two reasons why the *engine-return* restriction must be imposed. The first and most obvious reason is that *engine-return* is a single, globally-defined procedure, and thus we must choose beforehand which engine to stop. The only sensible choice is for the global *engine-return* to stop the most recently activated engine.

We can solve this problem by providing each engine computation with its own *engine-return* procedure, but then we have a more subtle problem. If we use this procedure to implement operating system kernel services, we must have some way to obtain the state of the computation requesting the service. In the single-engine case this state is simply the current continuation. However, because the state of the computation in the nested engine case may include the fact that other engines are running, the current continuation is no longer sufficient. If we define the current continuation to include the state of the engine system, we capture the entire stack of running engines (including those below the engine in question); if it does not include the state of the engine system, we lose information about other engines above the engine in question. Therefore, we cannot produce a new engine from the current continuation to allow us to resume the computation unless we work from within the engine system itself. In the next section we propose a modified engine interface that allows engines to be nested in a more general fashion, avoiding the need to use continuations to implement operating system kernel services.

6. An Alternative Engine Interface

In order to nest engines while allowing a return from any running engine, our interface must provide each engine computation with its own *engine-return* procedure. We therefore require that the procedure passed to *make-engine* be a procedure of one argument; the argument supplied is the procedure capable of returning from the computation.

Furthermore, because we would like to be able to return from an engine and yet restart that engine's computation from where it left off, we must provide the engine's *return* procedure with sufficient information to do so. We must also provide the engine's handler with a means of sending information back to the interrupted computation. These two goals are accomplished by including a third argument to the *return* procedure passed to an engine, along with the usual value and ticks arguments. This argument is a procedure of one argument, which when invoked with a value will return a new engine capable of continuing the computation from where it was interrupted by the call to its *engine-return* procedure. The value passed to this engine-making procedure is provided to the continuation of the call to the *engine-return* procedure when the engine is subsequently run.

A potential ambiguity concerns which of two or more running engines that were created from the same computation, and thus share the same *engine-return* procedure, is aborted if that procedure is called. Our implementation returns control to the continuation of the most recently activated engine.

```
(define kernel)
(letrec
  ([ready-queue (queue)]
   [start
    (lambda (proc)
      (enqueue (make-engine (lambda (ret) (proc (trap ret))))
               ready-queue))]
   [restart
    (lambda (engine-maker v)
      (enqueue (engine-maker v) ready-queue))]
   [trap
    (lambda (engine-return)
      (lambda (msg arg)
        (engine-return
          (lambda (engine-maker)
            (case msg
              [uninterruptible
               (restart engine-maker (arg))]
              [start-process
               (start arg)
               (restart engine-maker #f)]
              [stop-process #f])))))]
   [dispatch
    (lambda ()
      (if (empty-queue? ready-queue)
          'finished
          ((dequeue ready-queue)
           (time-slice)
           (lambda (trap-handler ticks engine-maker)
             (trap-handler engine-maker)
             (dispatch))
           (lambda (engine)
             (enqueue engine ready-queue)
             (dispatch)))))])
  (set! kernel
    (lambda (proc)
      (start proc)
      (dispatch))))
```

Figure 4. The operating system modified to use the new engine interface.

The implementation of this new nestable-engine system (see Appendix A) is similar to the implementation of nestable engines with a global *engine-return*. The information necessary to control the currently active engines can still be kept on a stack. However, it is no longer sufficient to return to the continuation of the engine most recently activated, which would be on the top of the stack. Instead, a call to an *engine-return* procedure is much like the expiration of an engine; the stack must be unwound until the appropriate level is found so that the proper continuation can be resumed, and the unwound frames must be encapsulated in the *engine-maker* procedure passed to the *return* procedure so that a new engine can be built. In order to determine which engine to abort when an *engine-return* procedure is invoked, an identity field must be added to each frame of the engine stack. Since engines are distinguished by their *engine-return* procedures, and since in Scheme distinct procedures are distinct objects, the *engine-return* procedure itself is used to determine which engine is being aborted. The only additional complication involves arranging to pass the value provided to an *engine-maker* to the appropriate continuation when an engine created by explicit return is invoked.

This new interface is more complicated to use when implementing simple tools such as **parallel-or**, which do not require an explicit *engine-return* procedure. Fortunately, it is straightforward to define *make-simple-engine* (described in Section 2) in terms of the more complicated interface:

```
(define make-simple-engine
  (letrec ([simplify (lambda (engine)
                       (lambda (ticks return expire)
                         (engine ticks
                                 (lambda (value ticks engine-maker)
                                   (return value ticks))
                                 (lambda (engine)
                                   (expire (simplify engine))))))])
    (lambda (proc)
      (simplify (make-engine (lambda (ret) (ret (proc))))))))
```

Figure 4 contains a new version of the operating system kernel of Figure 1, suitably modified to employ the new interface. Processes run by this kernel can themselves run operating system kernels or **parallel-or** expressions, because the *trap* procedure passed to a process returns from a specific engine rather than from the innermost. This additional ability comes at no cost in terms of the complexity of the operating system kernel; the version of Figure 4 is in fact simpler because it does not directly involve continuations and because the *engine-maker* provided by the engine system simplifies the *restart* procedure.

7. Conclusion

We have demonstrated that both unnestable engines and nestable engines can be implemented on top of any language that supports first-class continuations. We have also introduced a more general engine interface that allows nestable engines to return from engines other than the most recently activated engine. The general interface also allows the computation, including any suspended engines, to be restarted from the point of return. This makes possible the direct implementation of a wide range of operating system models and the meaningful composition of language features implemented in terms of engines.

With the more general interface, it is no longer necessary or desirable to use continuations to provide access to operating system services. This makes engines more useful in languages that lack first-class continuations (assuming engines are provided primitively), and it saves the programmer from the complexity inherent in mixing continuations and engines.

We have shown how to build a timer interrupt mechanism on top of Scheme that provides full programmer control over when the timer runs. It should also be possible to use the timer interrupt mechanisms built into some implementations of Lisp and Scheme. It is possible to provide functionality identical to **timed-lambda** with as few as one or two instructions generated at the front of each procedure body.

The relationship between continuations and engines deserves further study. Should continuations include the state of the engine system, so that invoking a continuation made outside of an engine suspends the engine, and invoking a continuation made inside of an engine restarts the engine? For some applications this may be desirable, for others it may not be.

It would also be useful to study the use of fluid (dynamic) binding in the context of engines and in the context of combined engines and continuations. A protection mechanism for engines similar to *dynamic-wind* [7] for continuations would be straightforward to implement, and with this would come the ability to support fluid binding for engines.

Acknowledgements: The authors wish to thank Dan Friedman, Chris Haynes, and an anonymous reviewer for their comments on early versions of this article. The Scheme code presented in this article was formatted by Carl Bruggeman's "Scheme Texer."

References

[1] Alan Bawden and Jonathan Rees, "Syntactic Closures," *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, July 1988, 86–95.

[2] R. Kent Dybvig, *The Scheme Programming Language*, Prentice Hall, 1987.

[3] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. "Expansion-Passing Style: A General Macro Mechanism," *Lisp and Symbolic Computation 1*, 1, June 1988, 53–75.

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

[5] Christopher T. Haynes and Daniel P. Friedman, "Engines Build Process Abstractions," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984, 18–24.

[6] Christopher T. Haynes and Daniel P. Friedman, "Abstracting Timed Preemption with Engines," *Journal of Computer Languages 12*, 2, 1987, 109–121.

[7] Christopher T. Haynes and Daniel P. Friedman, "Embedding Continuations in Procedural Objects," *ACM Transactions on Programming Languages and Systems 9*, 4, October 1987, 582–598.

[8] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand, "Obtaining Coroutines with Continuations," *Journal of Computer Languages 11*, 3/4, 1986, 143–153.

[9] Eugene E. Kohlbecker, *Syntactic Extensions in the Programming Language Lisp*, Ph.D. Thesis, Indiana University, 1986.

[10] Jonathan A. Rees and William Clinger, eds., "The Revised^3 Report on the Algorithmic Language Scheme," *SIGPLAN Notices 21*, 12, December 1986.

[11] Gerald J. Sussman and Guy L. Steele Jr., "Scheme: An Interpreter for Extended Lambda Calculus," MIT Artificial Intelligence Memo 349, December 1975.

[12] Mitchell Wand, "Continuation-Based Multiprocessing," *Conference Record of the 1980 LISP Conference*, August 1980, 19–28.

Appendix A: Nested engines with lexical return procedures.

```
(define make-engine)

(letrec
   ([new-engine
     (lambda (proc id)
        (lambda (ticks return expire)
           ((call/cc
               (lambda (k)
                  (run proc
                       (stop-timer)
                       ticks
                       (lambda (value ticks engine-maker)
                          (k (lambda () (return value ticks engine-maker))))
                       (lambda (engine)
                          (k (lambda () (expire engine))))
                       id))))))]

    [run
     (lambda (resume parent child return expire id)
        (let ([ticks (if (and (active?) (< parent child)) parent child)])
           (push (− parent ticks) (− child ticks) return expire id)
           (resume ticks)))]

    [go
     (lambda (ticks)
        (when (active?)
           (if (= ticks 0)
               (timer-handler)
               (start-timer ticks timer-handler))))]

    [do-return
     (lambda (proc value ticks id₁)
        (pop (lambda (parent child return expire id₂)
                (if (eq? id₁ id₂)
                    (begin (go (+ parent ticks))
                           (return value
                                   (+ child ticks)
                                   (lambda (value) (new-engine (proc value) id₁))))
                    (do-return
                       (lambda (value)
                          (lambda (new-ticks)
                             (run (proc value) new-ticks (+ child ticks) return expire id₂)))
                       value
                       (+ parent ticks)
                       id₁)))))]
```

```
 [do-expire
  (lambda (resume)
    (pop (lambda (parent child return expire id)
           (if (> child 0)
               (do-expire (lambda (ticks) (run resume ticks child return expire id)))
               (begin (go parent)
                      (expire (new-engine resume id)))))))]

 [timer-handler (lambda () (go (call/cc do-expire)))]

 [stack '()]

 [push (lambda l (set! stack (cons l stack)))]

 [pop
  (lambda (handler)
    (if (null? stack)
        (error 'engine "attempt to return from inactive engine")
        (let ([top (car stack)])
          (set! stack (cdr stack))
          (apply handler top))))]

 [active? (lambda () (not (null? stack)))])

(set! make-engine
  (lambda (proc)
    (letrec ([engine-return
              (lambda (value)
                (call/cc
                  (lambda (k)
                    (do-return (lambda (value)
                                 (lambda (ticks)
                                   (go ticks)
                                   (k value)))
                               value
                               (stop-timer)
                               engine-return))))])
      (new-engine (lambda (ticks)
                    (go ticks)
                    (proc engine-return)
                    (error 'engine "invalid completion"))
                  engine-return))))
```