Abstracting Timed Preemption With Engines

by

Christopher T. Haynes

and

Daniel P. Friedman

Department of Computer Science
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT N0. 178

Abstracting Timed Preemption With Engines

By

Christopher T. Haynes and Daniel P. Friedman

Revised: November, 1986

Research reported herein was supported in part by the National Science Foundation under grants MCS 83-03325, MCS 83-04567, and MCS 85-01277.

Comput. Lang. Vol. 12, No. 2, pp. 109–121, 1987
Printed in Great Britain.

0096-0551/87 $3.00 + 0.00

ABSTRACTING TIMED PREEMPTION WITH ENGINES†

CHRISTOPHER T. HAYNES and DANIEL P. FRIEDMAN
Computer Science Department, Indiana University, Lindley Hall 101, Bloomington, IN 47405, U.S.A.

(*Received* 18 *July* 1986; *revision received* 19 *November* 1986)

Abstract—The need for a programming language abstraction for timed preemption is argued, and several possibilities for such an abstraction are presented. One, called *engines*, is adopted. Engines are an abstraction of bounded computation, not a process abstraction in the usual sense. However, in conjuction with first class continuations, engines allow a language to be extended with time-sharing implementations for a variety of process abstraction facilities. We present a direct implementation of hiaton streams. Engine *nesting* refers to the initiation of an engine computation by an already running engine. We consider the need for engine nesting and show how it may be accomplished in a manner that charges a parent engine for the computation of its offspring. We conclude by discussing the importance of simple and general abstractions such as engines.

Engines First class objects Preemption Continuations Hiatons

1. INTRODUCTION

In this paper we introduce an *engine* facility that abstracts the notion of *timed preemption*. In conjuction with the ability to maintain multiple control environments, engines allow a language to be extended with a variety of process abstraction facilities. Engines are represented as procedural objects that embody some computation. In this respect they resemble *thunks* (procedures of no arguments), which are sometimes called *futures*, for engines embody a computation that may be performed at some future time. However, futures are invoked with no limit on the time that may be required to complete their computation, whereas engines are run with a specified computation time limit after which control is returned to the invoker. In this sense engines may be thought of as *bounded* futures.

By designing a base language with a few general abstraction mechanisms such as engines and continuations (which abstract the control environment), a powerful basis for building programming environments is provided. In this paper we are *not* introducing yet another process abstraction; rather, we are proposing an abstraction that allows implementation of arbitrary process abstractions through multiprogramming. This assures language extensibility and suitability to a wide range of applications. Concurrency issues, such as process synchronization primitives, are not central to our concerns in this paper. Though asynchronous preemption of a process implemented by multiprogramming with engines may introduce synchronization pathologies similar to those encountered in multiprocessing, they may be remedied with traditional concurrent programming techniques.

Coroutines are a mechanism for maintaining multiple control contexts without multiprocessing, but they are fundamentally different from engines. Control is passed from one coroutine to another *synchronously*, i.e. under explicit control of the program, by means of operations such as resume and detach. An engine computation, on the other hand, may relinquish control either synchronously, by deliberately returning, or *asynchronously* by a timed preemption that involves no explicit action by the computation.

Most computing environments provide a mechanism for timed preemption, such as a real-time clock capable of generating interrupts. In conjunction with a means of saving control state, such as continuations, these mechanisms may be used to implement engines. However, such mechanisms are frequently unavailable to the applications programmer, and when available they are usually imported from the operating system. As such, they tend to be inconvenient to use and always

†This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567, MCS 83-03325 and MCS 85-01277.

implementation dependent. By abstracting timed preemption and making it a standard language feature, it is possible to give concise and portable expression to a wider range of problems.

Since the introduction of engines several years ago [1], this facility has been adopted by several Scheme systems [2–4]. Engines are particularly useful in a language, such as Scheme, that provides first class procedures and control objects.

In the next section we provide an overview of Scheme. We then define the engine mechanism and include some simple examples of its use. Next we illustrate a more elaborate use of engines by implementing a simple time-sharing process scheduler. Engines are then used to implement hiatons and amb operations. This raises the issue of nested engines, for which an implementation is given. We conclude with some remarks on formalization of engines and a discussion of the importance of simple and general abstractions, such as engines.

2. AN OVERVIEW OF SCHEME

Scheme is a dialect of Lisp that is applicative order, lexically scoped, and properly tail-recursive [5, 6]. Most importantly, Scheme treats procedures and continuations as first class objects.

See Fig. 1 for the syntax of a Scheme dialect sufficient for the purposes of this paper. The superscript * denotes zero or more, and + denotes one or more occurrences of the preceding form. Square brackets are interchangeable with parentheses, and are used in the indicated contexts for readability. Constants, such as numbers, are self-evaluating. quote expressions return the indicated literal object, and '⟨object⟩ is equivalent to (quote ⟨object⟩). begin expressions evaluate their subexpressions in order and return the value of the last. Expression lists in lambda and let are implicit begins. lambda expressions evaluate to first-class procedural objects that lexically bind their arguments when invoked. let makes lexical bindings, destructuring the values if necessary. rec evaluates its expression in an environment that binds its identifier to the value of the expression itself. (rec is similar to the label for of Lisp.) if evaluates its second expression if the first is true, and the third otherwise. case evaluates the tag expression, and then returns the value of the first expression with a corresponding symbol that matches the tag. define assigns to a global identifier. set! modifies an existing lexical identifier. An application evaluates its expressions (in an unspecified order) and applies the procedural value of the first expression to the values of the remaining expressions.

We require a few primitive procedures: =, +, − and positive? are the usual arithmetic operations. cons is the traditional Lisp binary construction operation, with associated selectors car and cdr, and mutators set-car! (rplaca) and set-cdr! (rplacd). Lists are constructed of cons cells. list constructs lists, append! concatenates them, copy recursively copies list structures, and last-pair returns the last cons cell of a list. map is the usual list mapping procedure. eq?, pair? and null? are, respectively, the equality, cons cell and empty list predicates.

As any expression is evaluated, the current context of evaluation is continually maintained by the evaluation mechanism. The context of evaluation of each subexpression controls how evaluation will *continue* when the subexpression's value has been obtained. Hence control contexts

```
⟨expression⟩::=
  |⟨identifier⟩
  |⟨constant⟩
  |(quote ⟨object⟩)
  |(begin ⟨expression⟩+)
  |(lambda (⟨identifier⟩) ⟨expression⟩+)
  |(let ([⟨id-pattern⟩ ⟨value⟩]*) ⟨expression⟩+)
  |(rec ⟨identifier⟩ ⟨expression⟩)
  |(if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
  |(case ⟨tag⟩ [(⟨symbol⟩+) ⟨expression⟩]+)
  |(define ⟨identifier⟩ ⟨expression⟩)
  |(set! ⟨identifier⟩ ⟨expression⟩)
  |⟨application⟩
⟨value⟩, ⟨tag⟩, ⟨procedure⟩::=⟨expression⟩
⟨application⟩::=(⟨procedure⟩ ⟨expression⟩*)
⟨id-pattern⟩::=⟨identifier⟩|(⟨id-pattern⟩.⟨id-pattern⟩)|(⟨id-pattern⟩*)
```

Fig. 1. Syntax of a Scheme subset.

are called *continuations*. Continuations may be represented as procedural objects of one argument. When invoked with a value, the continuation proceeds with the computation as if the given value was the value of its subexpression.

Continuations are generally inacessible to the programmer. However, in Scheme it is possible to obtain the continuation of any expression using the procedure call-with-current-continuation, abbreviated call/cc. call/cc is passed a procedure that it then calls with the current continuation (the continuation of the call/cc application).† This continuation represents the remainder of the computation from the call/cc application point. At any future time this continuation may be invoked with any value, with the effect that this value is taken as the value of the call/cc application [10, 11]. (The continuation of a continuation application is discarded unless it has been saved with another call/cc.)

For example, assume the expression (+ 1 (* 2 3)) was being evaluated with continuation K (the value 7 is ultimately to be passed to K). Then the evaluation of the subexpression (* 2 3) has a continuation, k, that may be represented as (lambda (x) (K (+ 1 x))), so that (k 6) = (K 7). Using call/cc we can obtain k as an object of computation and invoke it with some other value, say 2. For example, the expression

```
(+ 1
   (call/cc
     (lambda (k)
       (* (k 2) 3))))
```

evaluates to 3, and the multiplication is never performed.

For this simple example, k need not be a first class object. Since it was not used outside of the dynamic context that it represents, in this case the continuation invocation could have been performed by simply popping the control stack until the addition application frame was reached. Control mechanisms, such as the catch tags provided in many Lisp systems, could have been used instead. However, in this paper we make extensive use of continuations to record the control context prior to context switches. Control is then returned to a context by invoking its continuation. Such invocation is performed from within another context, and hence we make full use of the first-class nature of continuations. Catch tags and related escape procedure mechanisms, which are not first-class, could not be used for this purpose.

Though control information may still be stack allocated much of the time, the use of call/cc requires that control information be heap allocated. This reflects the need for multiple control contexts in any multiprogramming environment.

3. DEFINITION OF ENGINES

Metaphorically, an engine is run by giving it a quantity of fuel. If the engine completes its computation before running out of fuel, it returns the result of its computation and the quantity of remaining fuel. If it runs out of fuel, a *new* engine is returned that, when run, continues the computation.

More formally, engine fuel is measured in *ticks*, an unspecified unit of computation. Though in some implementations a tick may represent (as the name suggests) a fixed amount of time, this is not required by the engine abstraction. Ticks measure *computation*, not time.‡ An *engine* is a procedural object of three arguments. The procedure make-engine takes a thunk (a procedure of no arguments) and returns a new engine that, when applied to a positive integer n, a two-argument *success* procedure and a one-argument *fail* procedure, proceeds to invoke the thunk for n ticks. If the thunk invokes the procedure engine-return with some value after m ticks, for some $m \leq n$, then the success procedure is applied to this value and the number $t = n - m$ of ticks remaining.

†Using this primitive we can define catch, a version of Landin's *J* operator [7–9]: (catch *id exp*) ≡ (call/cc (lambda (*id*) *exp*)).

‡Thus an engine tick is similar to a *computon*, which has been defined as "a mythical subatomic particle that bears the unit quantity of computation" [12].

If the thunk does not invoke engine-return in n ticks, then the fail procedure is applied to a new engine that, when invoked, continues with the thunk's computation. Thus the types of engines are

D	denotable values
$ET = \textit{positive integers}$	engine ticks
$Succ = D \times ET \rightarrow D$	success procedures
$Fail = Eng \rightarrow D$	failure procedures
$Eng = ET \times Succ \times Fail \rightarrow D$	engines

Though we speak of an engine computation *returning* a value, it is an error for the thunk invoked by an engine to return a value directly (to the continuation of its invocation)—it must invoke engine-return for this purpose. When a success or fail procedure is invoked upon the termination of an engine, any value returned by the success or fail procedure is returned as the value of the engine invocation.

The type structure chosen for engines is rather arbitrary. For example, another alternative would be to avoid the use of success and failure procedures by returning a value that could be either an engine, indicating failure, or a (*value . ticks-remaining*) pair, indicating success. We prefer the form presented here, because it avoids such disjoint union types and the associated need to program a test for success or failure following almost every engine invocation. (This forms is analogous to handling disjoint unions with a union-case procedure [13]).

Engines, like all objects in Scheme, are first class: they may be passed to procedural objects, be returned by procedures, be stored in data structures, and have indefinite extent. Hence engines, and the environment and control information that they contain references to, may be reclaimed by a garbage collector when they are no longer accessible.

A tick might correspond to the execution of one Virtual Scheme Machine instruction, as in Scheme 84 [3] or to a number of milli-seconds of processor time, as in PC Scheme [4]. In general, the amount of computation associated with a tick is not defined, but must satisfy the following "real-time" constraints: (1) a larger tick count is associated with a larger expected amount of computation (in the statistical sense) and (2) unbounded real time is associated with an unbounded number of ticks (any looping computation must consume ticks). Thus, engine ticks might be metered by a real-time clock, with an unpredictable amount of time spent handling interrupts. A compiled language without a real-time clock could still implement engines by decrementing and testing a tick count at least once with each recursive call or iteration. For Scheme, which has no goto or other primitive iteration mechanism, it suffices to associate a tick with each user defined procedure invocation, as in some versions of Chez Scheme [2].†

We say an engine is *running* if neither its success nor fail procedure has been invoked since the last invocation of the engine. The invocation of an engine by an already running engine, which we refer to as *nesting* of engines, is disallowed by the basic engine mechanism. That is, at most one engine may be running at a given time. This simplifies the implementation of engines, and results in no loss of generality. We shall see in Section 7 that there are at least two distinct approaches of engine nesting, both of which may be implemented using unnested engines. We prefer not to commit the standard engine mechanisms to either of these nesting alternatives. Furthermore, our experience has been that when it is believed that nested engines are needed, closer examination of the problem usually reveals that nesting is not only unnecessary, but leads to unnecessarily complicated programs. Applications with genuine need for engine nesting will be discussed in Sections 6 and 7.

In the absence of traditional operations that change state, such as assignment and I/O, engines do not have state! Invoking the same engine twice runs the engine's computation from the same point. This differs from the usual interpretation of processes. To record the progress of an engine, a new engine is created. It would be possible to define an otherwise engine-like mechanism in which

†Reference [2] also includes an implementation of engines using a timer, with attention to the complexities encountered when engines coexist with other real-time facilities.

the original engine was modified upon exhaustion of the tick count, rather than returning a new engine. However, we believe that the side-effect-free approach we have taken is superior; it makes engines more suitable than traditional processes for applicative styles of programming in which side effects are disallowed or restricted.

Though engines are "procedural", they may be non-deterministic: the result of invoking the same engine twice may be different due to random differences in the rate at which ticks are consumed by its computation.

4. SIMPLE EXAMPLES

(1) The engine facility could be simplified a little further by defining *steppers* as procedures of two arguments, which perform just one tick of computation when invoked. It would then be superfluous to pass the number of ticks remaining when a stepper completes (for it would always be zero), so both the success and fail procedures would take one argument. As a simple example of the engine mechanism proposed here, we define a procedure that takes an engine and returns a stepper.

```
(define engine-to-stepper
  (lambda (eng)
    (lambda (succeed fail)
      (eng 1
        (lambda (val ticks) (succeed val))
        (lambda (new-eng)
          (fail (engine-to-stepper new-eng)))))))
```

It is also possible to implement our engine mechanism using steppers. While steppers may be more aesthetically pleasing, we have chosen to provide multi-step engines to avoid the obvious inefficiency.

(2) It is sometimes useful to run engines with an unlimited number of ticks. We abstract this behavior with a procedure complete, which invokes its succeed procedure with the value of the engine's computation and the number of ticks required for it to complete. Of course, unlike engine invocation, complete is not guaranteed to terminate. To implement complete, we repeatedly invoke a stepper to coax the engine computation along, until a value for the engine's expression is obtained.

```
(define complete
  (labmda (eng succeed)
    ((rec loop
      (lambda (stepper count)
        (stepper
          (lambda (val) (succeed val count))
          (lambda (new-eng) (loop new-eng (+1 count))))))
     (engine-to-stepper eng) 1)))
```

A good exercise is to write complete using regular engines, invoked with some arbitrary number of ticks, instead of steppers.

A procedure that just returns the number of ticks required to complete a computation may now be defined by

```
(define ticks
  (lambda (thunk)
    (complete
      (make-engine thunk)
      (lambda (value tick-count) tick-count))))
```

This might be used to compare the relative speeds of several algorithms, or to monitor the response time of code segments in a real-time system.

5. AN ENGINE PROCESS SCHEDULER

As an extended example of engine programming techniques, we use engines to implement a simple time-sharing operating system kernel. These techniques have broad applicability and may be used to implement a variety of process abstractions.

The basic technique is simple. The kernel dispatches a user process by running a corresponding engine with a tick count corresponding to its time slice. When the process's time is exhausted, the engine returns control to the kernel, which then dispatches the next process. We represent processes directly as engines. A ring of these processes is maintained, with the engines stored in ring entries that correspond to the PCBs (process control blocks) of traditional operating systems. The pcb-process and pcb-process! procedures extract processes from and store processes in these queue entries. The rotate! procedure rotates the ring and returns a reference to the next ring entry. The basic form of the kernel is:

```
((rec loop
  (lambda (pcb)
    (pcb-process! pcb
      ((pcb-process pcb)
       time-slice-length
       (lambda (val ticks-remaining) ???)
       (lambda (new-eng) new-eng)))
    (loop (rotate! ring))))
 (rotate! ring))
```

In practical systems it is also necessary to provide a trap mechanism that allows the user process to return control synchronously (without preemption) to the operating system with a service request. Thus we provide a procedure trap that may be invoked by a user process with a tag, which indicates the type of operating system service required and perhaps additional information specific to the service request. Invoking trap causes control to be returned to the kernel, passing the tag and other information. The kernel must then be able to resume the engine's computation.

To enable the kernel to return control to the engine at the point of the trap procedure call, the trap procedure first obtains (with call/cc) the continuation k of its invocation.† It then invokes engine-return, passing it *k* and the list containing the trap type and argument with which trap was invoked. We have:

```
(define trap
  (lambda (trap-type arg)
    (call/cc
      (lambda (k)
        (engine-return
          (list trap-type k arg))))))
```

The success procedure of the kernel's engine invocation receives control after a trap, with the list (*trap-type k arg*) as the success value. See Fig. 2 for a kernel with a simple trap handler that provides some standard process primitives. The handler dispatches on the type of the trap, obtaining some answer that is to be returned to the user. This return is accomplished by the expression (make-engine (lambda () (k ans))), which creates a new engine that invokes the user's continuation with the answer to be returned. In this scheduler the ticks-remaining is ignored, so a process loses the rest of its time-slice when it traps. As an exercise, the scheduler may be modified to charge for traps in a more consistent fashion.

---

†This technique of recording process state with continuations is also useful in the context of multiprocessing and interrupts [14].

In case the type of the trap is awaken or block, the standard ring operations are performed. fork creates a new process that evaluates the argument thunk. (This thunk should never return.) The atomic trap simply invokes its argument as part of the trap operation; this is used to execute code uninterruptibly.

```
(define kernel
  (lambda (ring)
    ((rec loop
       (lambda (pcb)
         (pcb-process! pcb
           ((pcb-process pcb)
            time-slice-length
            (lambda (trap-value ticks-remaining)
              (let ([(trap-type k arg) trap-value])
                (let ([ans (trap-handler trap-type arg ring)])
                  (make-engine (lambda () (k ans))))))
            (lambda (new-eng) new-eng)))
         (loop (rotate! ring))))
     (rotate! ring))))
(define trap-handler
  (lambda (trap-type arg ring)
    (case trap-type
      [(awaken) (insert! ring arg)]
      [(block) (delete! ring arg)]
      [(fork) (insert! ring (make-entry (make-engine arg)))]
      [(atomic) (arg)])))
```

Fig. 2. Operating System kernel.

It is possible through extensive use of procedural abstraction to abstract the trap handling procedure from the kernel and simultaneously maintain a security constraint that only the kernel be able to create and invoke engines. Furthermore, a hierarchy of trap handlers may be used, so that the user can only perform higher level trap procedures, such as semaphore operations and parbegin [1]. A version of the kernel presented here, extended to include semaphores and parbegin, has been used by our operating systems and advanced programming languages classes for several years.

6. HIATONS AND AMB

Another use of engines is to implement an asynchronous merge of two streams. The head of a stream (which may be represented by the car of a cons cell) is immediately available. An attempt to access the tail of the stream results in some computation or input operation being performed; only on completion of this operation is the tail of the stream returned. (Typically the cdr of a cons cell representing a stream refers to a thunk that is invoked when the stream tail is accessed.)

The elements of two streams may be synchronously merged to form a third stream by simply alternating from one to the other; but then the speed that the output stream is generated is limited by the slower of the two input streams. Thus if one of the input streams becomes undefined (at some point the computation of the next element does not terminate), then the merge output becomes undefined. An asynchronous merge of streams avoids these problems: it forms the output stream by selecting elements from the input streams as they become available. In particular, an asynchronous merge output becomes undefined only if both its inputs become undefined.

It is straightfoward to implement an asynchronous merge directly using engines, but here we taken an indirect approach—first implementing *hiaton-streams*, an abstraction of the hiatus, or delays, implicit in the production of stream values that has been proposed recently [15]. A hiaton-stream is simply a stream that may contain, in addition to its normal values, distinguished tokens, or *hiatons*, indicating the lapse of some time or computation in the generation of the stream.

Engines may be used to implement hiaton-streams, as in the following procedure that takes a stream and returns a hiaton-stream.

```
(define stream→hiaton-stream
  (lambda (stream)
    (let ([(value.thunk) stream])
      (cons value
        ((rec loop
          (lambda (eng)
            (lambda ()
              (eng number-of-ticks-per-hiaton
                (lambda (stream ticks)
                  (stream→hiaton-stream stream))
                (lambda (new-eng)
                  (cons 'H (loop new-eng))))))))
         (make-engine thunk))))))
```

The thunk created by evaluating the (lambda ()...) expression is invoked to obtain the next element of the hiaton-stream. This runs an engine that invokes the cdr of the stream. If the next stream element is produced in less than a hiaton's worth of time, the engine invocation's success procedure simply passes the stream element to stream→hiaton-stream, which makes the car of the stream immediatley available and builds a hiaton generator for the cdr of the stream. If the next stream element is not produced in the given number of ticks, the fail procedure returns a stream whose car is the symbol H (a hiaton token) and whose cdr is a hiaton generator that, when invoked, coaxes the newly generated engine for the next stream element, or hiaton.

To implement asynchronous merge, we begin by converting the streams to hiaton-streams. A synchronous merge may then be used to combine these hiaton-streams into a single hiaton-stream, and finally the hiatons may be culled from this stream, to yield the merge output stream.

Another approach to effecting nondeterminism is *amb* [16]. Operationally, amb starts two computations, represented by its arguments, and returns the result of the one that finishes first. The amb arguments could be represented as streams, the second element of which is the value associated with the argument. (The first element of the stream is immaterial for our purposes, for it is immediately available and thus cannot be used to express a pending computation.) Amb may then be implemented by simply applying the asynchronous merge described above to the argument streams. The first two elements of the resulting stream are the two immaterial values, and the third element is the amb result.

One feature of this amb implementation must be noted. Amb calls may be nested to simulate an amb of more than two computations, or because one of the computations passed to amb happens to use amb at some point. In the implementation suggested above, this results in a hiaton-stream being used by another hiaton-stream, which in turn results in an engine being invoked by a computation that is already running as an engine. Thus our implementations of amb and hiaton-streams require nested engines (though it is possible to implement nested amb in other ways without nesting engines).

7. IMPLEMENTING NESTED ENGINES

In this section we show how to eliminate the restriction that engines cannot be nested, thereby allowing engine invocation when an engine is already running. This allows any number of engines to be running at the same time. For the remainder of this section, we refer to an engine that can be nested as a *nester* to distinguish it from an engine that cannot be nested.

If e is the nester invoked most recently at the time another nester, e', is invoked, we say that e' is the child of e. All the nesters running at the time e' is invoked are said to be the ancesters of e'. Should each tick of a nester be 'charged' only to the nester itself, or to the nester and all of its ancestors? We term these alternatives *simple* and *fair* nesting, respectively.

Simple nesting is straightfoward to implement given unnested engines (using some of the techniques employed in the more involved fair implementation that follows). This may be adequate for some applications of nesters, but not all. For example, in the hiaton stream example a hiaton

should represent a fixed number of ticks of computation expended in evaluating an expression (which is achieved by using a nester to evaluate the entire expression). It should not matter whether or not the computation invokes other nesters in the course of its evaluation. However, with simple nesting it would be possible for a computation to cheat by using other nesters, run as its children, to perform much of its work. In this case the nester mechanism is failing to abstract the use of time in the intended way. We expect the nesting mechanism to be 'fair' in the sense that every tick of computation used by an engine is also charged to its parent, and by extension to all of its ancestors. Also, if nesting is 'abstract', a computation should be able to parcel out the ticks it receives among its children without being aware of whether or not it is running with one or more ancestor nesters.

To see how fair nesting may be achieved, assume a nester e_0 is given t_0 ticks while nesters $e_1, \ldots, e_n$ are running with $t_1, \ldots, t_n$ ticks remaining, respectively, where $0 \leq i < j \leq n$ implies e_j is an ancestor of e_i. The fairness requirement implies that e_0's computation can proceed for at most $t = min(t_0, t_1, \ldots, t_n)$ ticks before a preemption. (A preemption may occur in less than t ticks, for example if e_0 invokes another nester with t_i ticks when it has t_j ticks left and $t_j > t_i$.) If $t = t_k < t_0$, then the preemption of e_0 after t ticks results in the failure procedure f_k of e_k being invoked with a new nester e. (The failure procedure of e_0 should not be invoked at this time, since t_0 ticks have not been expended on e_0's computation.) When f_k is invoked, the running of $e_0, \ldots, e_{k-1}$ is suspended and their state encapsulated in e. When e is invoked it should resume running $e_0, \ldots, e_{k-1}$ with $t_0 - t, \ldots, t_{k-1} - t$ ticks remaining, respectively.

Providing fair nesting of engines is analogous to recursively virtualizing an operating system. In both cases resources at each node in the spawning tree are divided among offspring in such a way that the offspring are unaware of the division.

Applications for fair nesting include artificial intelligence problems in which engines are used to control the search strategy. When a choice point is reached, a process may be created to explore each of the alternatives. Nesters may be used to implement these processes using time-sharing. If each of the alternatives is equally likely, each of the processes should receive equal time. If some of the processes reach additional choice points and spawn additional processes, these children should share the computation resources of their parent. Thus fair, not simple, nesting is required.

A fair nester implementation must record, for each currently running nester, the success and fail procedures associated with its invocation, its parent, and the number of ticks it has remaining. This information is maintained in a record, also referred to as a nester. Thus the system must effectively maintain a list of currently running nesters, ordered from youngest (most recent child) to oldest. If each child completes its computation (indicated by an engine-return) or expires (runs out of ticks) before any of its ancestors expire, then the nester list grows and shrinks in a stack-like manner. The youngest and oldest currently running nesters are the stack top and stack bottom, respectively.

The interest in this implementation is provided by the possibility that the first currently running nester to expire may not be the youngest. The first nester to expire among those currently running is the one with the minimum number of ticks remaining. This nester may be determined by examining the nester stack at the time each new nester is invoked. Nesting is simulated by running each new nester as an (unnested) engine for this minimum number of ticks, with appropriately constructed success and fail procedures.

For example, Fig. 3(a) pictures the nester stack just after nester D has been invoked with 30 ticks by its parent nester C, with 20 ticks remaining. Nester C's parent and grandparent are nesters B and A, with 10 and 40 ticks, respectively, remaining at the time of D's invocation. Nester A was invoked with no other nesters running, so it has no parent (it is the bottom of the nester stack). Nester B has the least number of ticks remaining, so it expires first—10 ticks after D's invocation.

When nester B expires, control returns to its parent, nester A, by passing the fail procedure of nester B a newly formed nester. The nester stack is now broken after nester B, as pictured in Fig. 3(b). The bottom segment, containing only nester A, becomes the new system nester stack. The top segment is recorded in the new nester passed to the fail procedure. The nester record B contains no useful information, but is filled with information again whenever the new nester is invoked.

Suppose that 5 ticks later nester A invokes a new nester with 40 ticks. The resulting nester stack is pictured in Fig. 3(c). Further suppose that after another 4 ticks the nester created at the time nester B expired is invoked with 20 ticks. The nester stack segment pictured on the top of (b) is

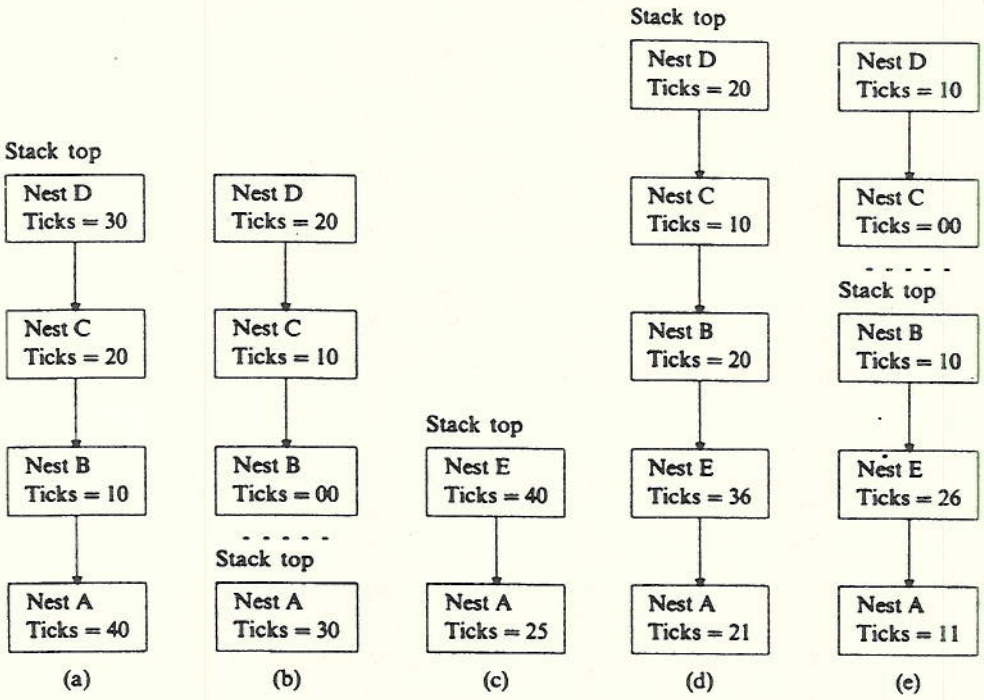

Fig. 3. Nest example.

now appended to the system nester stack to obtain the stack pictured in (d). The minimum number of ticks in this stack is 10, belonging to nester C. Thus after another 10 ticks the stack is again broken, this time between nesters B and C, with the result pictured in (e).

The append operation that forms stack (d) is of central importance. It has the effect of resuming the nesters in the appended stack segment with the number of ticks they had left at the time their ancestor (the first nester B) ran out of ticks. The append operation must copy the appended stack frames belonging to the resumed nester because this nester, like an engine, may be resumed more than once.

We now define a procedure that makes nesters using make-engine:

```
(define make-nester
  (lambda (thunk)
    (nestermaker (make-engine thunk) (list any))))
```

where nestermaker, defined in Fig. 4, embodies the behavior just illustrated, and any is an aribtrary value. nestermaker takes an engine and a nester stack that records the nesting context. nestermaker returns a nester which, like all engines, is a procedure that takes a number of ticks, a success procedure, and a fail procedure.

Nester records are represented by a data structure of the form

$$((\textit{ticks}\ \textit{success}\ \textit{fail})\ .\ \textit{parent})$$

where *ticks* is the number of ticks remaining for the nester, *success* and *fail* are the success and fail procedures of the nester's invocation, and *parent* points to the invoking nester's record. Each nester record may also be viewed as a nester stack, represented as a list linked by the *parent* pointers.

When a nester is invoked, it first obtains the continuation of its invocation, which is bound to k. The success and fail continuations of the nester are now constructed by composing k with the success and fail procedures with which the nester was invoked. These are stored, along with the number of ticks, in the last nester of the stack. If a nester is already running, as indicated by a global flag, then an engine-return is performed. If no nester is running, then the running flag is set and the run procedure is called to invoke the engine. run calls find-minpoint to locate the nester record with the minimum number of ticks remaining. (If more than one record has the same number, any of them could be returned; but there is a small efficiency advantage to returning the

```
(define nestermaker
  (lambda (eng stk)
    (lambda (ticks succeed fail)
      (call/cc
        (lambda (k)
          (set-car! (last-pair stk)
            (list ticks
              (lambda (value ticks) (k (succeed value ticks)))
              (lambda (new-eng) (k (fail new-eng)))))
          (if running
            (engine-return (list '*return eng stk))
            (begin (define running true)
                   (run eng stk))))))))
(define run
  (lambda (eng stk)
    (let ([minpoint (find-minpoint stk)])
      (let ([((ticks succeed fail) . parent) minpoint])
        (eng ticks
          (lambda (value ticks-remaining)
            (decrement! stk (— ticks ticks-remaining))
            (if (if (pair? value)
                    (eq? (car value) '*return)
                    false)
                (let ([(rtn-eng rtn-stk) (cdr value)])
                  (run rtn-eng (append! (copy rtn-stk) stk)))
                (let ([((ticks succeed fail) . parent) stk])
                  (return-from-nester
                    parent
                    (lambda () (succeed value ticks))))))
          (lambda (new-eng)
            (decrement! stk ticks)
            (set-cdr! minpoint '())
            (return-from-nester
              parent
              (lambda () (fail (nestermaker new-eng stk))))))))))
(define return-from-nester
  (lambda (stk thunk)
    (if (null? stk)
        (begin (define running false)
          (thunk))
        (run (make-engine thunk) stk))))
```

Fig. 4. Nested engine implementation.

one closest to the bottom of the stack.) The engine passed to run is now invoked with the number of ticks of the minpoint record.

If an engine-return is performed before the ticks are exhausted, the success procedure of the engine invocation is invoked, which first calls decrement! to update the tick counts in the nest stack. If the engine-return resulted from the nestermaker operation discussed above, this was noted by a *return tag, which is accompanied by a stack segment and engine. run is then invoked with this engine and a stack formed by appending the returned stack segment to the current stack. On the other hand, if the engine-return indicated the normal termination of a nester, return-from-nester is invoked with a thunk that invokes the success procedure of the top nester with the number of ticks remaining for it and the value passed to the engine-return. return-from-nester checks if the stack is empty (i.e., the returning nester is the last one), in which case the running flag is cleared, and the thunk is invoked with no engine running. Otherwise, run is invoked with a new engine that invokes the thunk. The stack is popped when run is called to discard the nester that just returned.

If the engine invoked by run expires instead of returning, all the stack tick counts are decremented. Then the stack is broken at the minpoint by installing the empty stack as the parent of the minpoint (using set-cdr!). Finally, return-from-nester is invoked with the bottom part of the broken stack and a thunk. When invoked, this thunk calls the fail continuation with a new nester created by nestermaker from the new engine and the top part of the broken stack.

At the expense of clarity, the efficiency of this implementation can be improved in several ways.

For example, the decrement! calls could be avoided (at least until integer overflow becomes a problem) by keeping a count to be subtracted from the nester record tick values before using them.

8. CONCLUSION

We have considered several abstractions for timed preemption, including steppers, engines and nesters. We have shown these to be of equivalent expressive power, but have chosen engines for pragmatic reasons.

To illustrate the use of engines, we have implemented a sample operating system scheduler, hiaton-streams and amb. The last two examples demonstrated the need for nesting of engines. A nesting implementation was presented that charged an engine for the computation ticks ueed by its offspring.

It is natural to ask what tools might be developed for reasoning about engines. Aside from the real-time character of engines, there would still seem to be difficulties. Until recently there was no satisfactory theory for reasoning about continuations, and continuations are necessary in order to record the context of a preempted engine. However, an algebraic extension of Plotkin's λ_v-calculus [17], termed the λ_c-calculus, has been developed that allows us to reason about continuations. Starting with a tree rewriting system, a reduction system is derived that in turn yields a calculus which is Church–Rosser and has a standard reduction function [18, 19]. These tree rewriting techniques have also been used to develop a related calculus which incorporates assignment statements [20].

It seems possible to associate engine ticks with reductions in a similar reduction system that models engines. However, such an engine semantics would not be a calculus. Since the number of reductions required to reduce an expression to normal form is dependent on the order of reduction, the Church–Rosser property (among others) would not hold. Nonetheless, a reduction semantics might yield useful proof rules; we leave this to future work.

Any engine specification that precisely determines the computation quantity associated with a tick would not be entirely satisfactory. This includes reduction systems such as those discussed above and interpreters which associate ticks with meta-recursion, as in Ref. [21]. We have been careful to leave the exact nature of a tick unspecified. We require only a 'liveness' property that prohibits any engine from running forever, and a 'fairness' property that associates more ticks with more computation, on the average. Any reasonable metric of computation may be used, including real-time, formal reduction, function invocation and virtual instruction execution. Furthermore, ticks may not always represent the same quantity of computation, even when repeatedly evaluating the same expression. This generality gives implementors of the engine mechanism freedom to make efficient choices, which may be heavily constrained by the computation environment. It also discourages programmers from writing code that is timing dependent. However, this implies that engines are non-deterministic. It appears that a fully satisfactory formal semantics of engines must await the resolution of some difficult problems in the semantics of non-determinism.

In spite of the semantic difficulties posed by engines (many of which are shared with other non-deterministic facilities), we believe there are practical advantages to abstracting mechanisms for timed preemption in any multiprogramming environment. Many process abstraction facilities have been proposed, with a wide variety of design goals. A language designer who chooses one of these facilities, or opts to invent yet another, risks rapid design obsolescence. The resulting langauge would also be ill-suited to a variety of applications that do not fall within the language's design goals. Attempts to avoid the latter problem by introducing intricate facilities result in a complex design that is optimally suited to few applications. Unfortunately, any multiprocessing implementation must be designed to support one or more specific process abstractions. However, when multiprogramming is used, there is an alternative to committing a language design to a specific process abstraction facility: an abstraction of timed preemption, such as engines, may be used instead to provide time-sharing implementations of process abstractions.

In a more general context, independent of the notion of process, we feel that languages should provide a means for bounding computation. Some form of timed preemption is implicit in any mechanism for asynchronously imposing computation bounds. Engines abstract this mechanism for bounding computation.

Acknowledgements—We gratefully acknowldge Eugene Kohlbecker's help in the development of an earlier engine operating system, and his suggestion of the term *engine*. We thank Bruce Duba, Kent Dybvig, Edward Robertson, Mitchell Wand and anonymous referees for their comments on earlier drafts of this paper.

REFERENCES

1. Haynes C. T. and Friedman D. P., Engines build process abstractions. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 18–24 (August 1984).
2. Dybvig R. K., *The Scheme Programming Language*, Prentice-Hall (1987).
3. Friedman D. P., Haynes C. T., Kohlbecker E. and Wand M., *The Scheme 84 Interim Reference Manual*. Indiana University Computer Science Department Technical Report No. 153 (June 1985).
4. *TI Scheme Language Reference Manual*, Computer Science Laboratory, Texas Instruments Inc., Dallas, Tex. (1986).
5. Sussman G. J. and Steele G. L., Scheme: an interpreter for the extended lambda calculus. MIT Artificial Intelligence Memo 349 (December, 1975).
6. Rees J. and Clinger W. C., (Eds), The revised[3] report on the algorithmic language scheme. *SIGPLAN Not.* 21(12), 37–79 (December, 1986).
7. Landin P., A correspondence between ALGOL 60 and Church's lambda notation. *Commun. ACM* 8(2–3), 89–101 and 158–165 (1965).
8. Reynolds J. C., Definitional interpreters for higher-order programming languages. *Proceedings of the 25th ACM National Conference*, pp. 717–740 (1972).
9. Felleisen M., Reflections on Landin's *J*-operator: a partly historical note. *Comput. Lang.* In press.
10. Friedman D. P., Haynes C. T. and Kohlbecker E., Programming with continuations. *Program Transformation and Programming Environments* (Edited by Pepper P.), pp. 263–274. Springer, Berlin (1984).
11. Haynes C. T. and Friedman D. P., Embedding continuations in procedural objects. *ACM Trans. Progr. Lang. Sys.* In press.
12. Steele G., Woods D., Finkel R., Crispin M., Stallman R. and Goodfellow G., *The Hacker's Dictionary*. Harper & Row, New York (1983).
13. Reynolds J. C., User-defined types and procedural data structures as complementary approaches to data abstraction. *Conference on New Directions in Algorithmic Languages*, IFIP Working Group 2.1, Munich (August 1975).
14. Wand M., Continuation-based multiprocessing. *Conference Record of the 1980 Lisp Conference*, pp. 19–28 (1980).
15. Wadge W. and Ashcroft E., *Lucid, The Dataflow Programming Language*. Academic Press, New York (1985).
16. McCarthy J., A basis for a mathematical theory of computation. *Computer Programing and Formal Systems* (Edited by Braffort P. and Hirschberg D.), pp. 33–70. North-Holland, Amsterdam (1963).
17. Plotkin G., Call-by-name, call-by-value, and the λ-calculus. *Theoret. Comput. Sci.* 1, 125–159 (1975).
18. Felleisen M. and Friedman D. P., Control operators, the SECD-machine, and the λ-calculus. *Proceedings of the Conference on Formal Description of Programming Concepts*, Part III, Denmark, August 1986. North-Holland, Amsterdam. In press.
19. Felleisen M., Friedman D. P., Kohlbecker E. and Duba B., A syntactic theory of sequential control. *Theoret. Comput. Sci.* In press.
20. Felleisen M. and Friedman D. P., A calculus for assignment in higher-order languages. *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 314–325 (1987).
21. Haynes C. T. and Friedman D. P., An abstraction of timed preemption. Computer Science Department Technical Report No. 178, Indiana University, Bloomington, Ind. (1985).

About the Author—CHRISTOPHER T. HAYNES received the Ph.D. degree from the University of Iowa in 1982. His interests include control abstraction, data typing, programming environments and logic programming.

About the Author—DANIEL P. FRIEDMAN received the Ph.D. degree from the University of Texas at Austin in 1973. His field is Programming Languages.