LISP AND SYMBOLIC COMPUTATION: An International Journal, 6, 453–484, 1993

Subcontinuations*

ROBERT HIEB†

R. KENT DYBVIG (*dyb@cs.indiana.edu*)

Indiana University Computer Science Department
Bloomington, IN 47405

CLAUDE W. ANDERSON, III (*anderson@cs.rose-hulman.edu*)

Rose-Hulman Institute of Technology Computer Science Department
Terre Haute, Indiana 47803

(Received: November, 1992)

(Revised: June, 1993)

Keywords: Continuations, Control Structure, Control Delimiters, Concurrency, Engines, Scheme

Abstract. Continuations have proven to be useful for implementing a variety of control structures, including exception handling facilities and breadth-first searching algorithms. However, traditional continuations are not useful in the presence of concurrency, because the notion of the rest of the computation represented by a continuation does not in general make sense. Traditional continuations can also be difficult to use in nonconcurrent settings, since their global nature is sometimes problematic. This article presents a new type of continuation, called a *subcontinuation*. Just as a traditional continuation represents the rest of a computation from a given point in the computation, a subcontinuation represents the rest of a *subcomputation* from a given point in the subcomputation. Subcontinuations may be used to control tree-structured concurrency by allowing nonlocal exits to arbitrary points in a process tree and allowing the capture of a subtree of a computation as a composable continuation for later use. In the absence of concurrency the localized control achievable with subcontinuations makes them more useful than traditional continuations.

1. Introduction

A continuation is an abstract entity that represents the rest of the computation from a given point in the computation. A language such as Scheme [4] that provides access to continuations as first-class values need not directly

*This material is based on work supported by the National Science Foundation under grant number CCR-88-03432 and by Sandia National Laboratories under contract number 06-06211. This article is a revised and extended version of a paper presented at the 1990 ACM Conference on Principles and Practice of Parallel Programming.

†Robert Hieb died in an automobile accident in April 1992.

support many traditional imperative control structures such as loops, "gotos," exception handlers, and coroutines. This simplifies the language and allows the programmer to create new control structures not anticipated by the language designer.

Traditional continuations, however, do not work well in the presence of concurrency, since the notion of the rest of the computation represented by a continuation does not in general make sense. If we use traditional continuations in the presence of tree-structured concurrent operators, such as a parallel call operator, we must decide whether the "current continuation" includes the rest of the computation back to the root of the process tree or whether it includes only the rest of the computation of the current (leaf) process. Neither approach is adequate in all cases. Restricting continuations to use within the leaves of a process tree makes exception handling difficult, since exceptions may need to propagate all of the way to the root process. On the other hand, if control is not localized to a leaf process, it is difficult to use nonlocal exits or other continuation-based control features within the leaf process. Furthermore, neither approach allows us to consider an intermediate portion of the process tree as a single unit; that is, we cannot exit from an arbitrary subtree of the process tree, nor can we use continuations to save the state of an arbitrary subtree. We must have some way to specify how far back in a process tree a continuation extends.

Traditional continuations can also be difficult to use even in the absence of concurrency, since their global nature sometimes results in undesirable nonlocal effects. For example, it is difficult to suspend a subcomputation and later restart it in a different continuation and still ensure that control returns to the new continuation when the subcomputation completes. Doing so typically requires a seemingly redundant continuation capture and an additional continuation invocation to go along with it (and several additional lines of comments). Again, we would like to have some way to specify how much of a computation is included in a continuation.

In this article we present a new type of continuation, called a *subcontinuation*, that allows us to do exactly that. Just as a traditional continuation represents the rest of a computation from a given point in the computation, a subcontinuation represents the rest of a *subcomputation* from a given point in the subcomputation. Because of this, subcontinuations are more useful than traditional continuations both in the presence and absence of concurrency. Subcontinuations provide complete control over trees of processes, allowing nonlocal exits to arbitrary points in a process tree and allowing the capture of a subtree of a computation as a composable continuation for later use. In the absence of concurrency, subcontinuations offer the ability to localize the effects of control operations.

Not all concurrency is tree-based. A good example of the distinction between tree-based and other forms of concurrency can be found in Halstead's Multilisp [1], which supports both parallel calls (with **pcall**) and *futures* (with **future**). **pcall** introduces tree-based concurrency, since it evaluates its arguments in parallel and then applies the value of the first argument to the values of the remaining arguments as in a normal procedure call. On the other hand, **future** initiates an independent parallel process that does not "return" a value; instead, the value is requested when needed, which may not be until after the parent process has returned from the code that created the future. It is the notion of returning, with or without values, to the point of creation that distinguishes tree-based concurrency from other forms of concurrency. Other examples of tree-based concurrency are McCarthy's *amb* operator [20] and related constructs such as parallel **and** and **or** operators. Although our mechanism does not apply to nontree-structured concurrency, we do discuss how our mechanism can be used in languages that allow both forms of concurrency.

The remainder of the article is organized as follows. In Section 2, we describe traditional continuation control strategies along with a few newer continuation control mechanisms, and we discuss some of the shortcomings of these strategies that subcontinuations are designed to solve. In Sections 3 and 4, we introduce subcontinuations and show how they can be used to control subcomputations, both concurrent and sequential, in a simple, consistent manner. In Section 5, we describe *control filters*, which allow us to specify entry and exit handlers similar to those established by Common Lisp's *unwind-protect* and Scheme's *dynamic-wind* control structures. In Section 6, we describe an implementation of *engines*, which provide multitasking capability, in terms of subcontinuations and control filters. In Section 7, we present a simple operational semantics for a call-by-value variant of the λ-calculus extended with subcontinuations, assignments, and control filters. In Section 8, we describe how subcontinuations are implemented. Finally, in Section 9, we make some concluding remarks.

2. Background

Continuations are commonly used in denotational semantics as a basis for deriving the meaning of control operations in imperative languages [25]. Many programming languages provide control operations such as jumps and exits that modify a program's continuation. Scheme makes continuations available as procedures via the procedure *call-with-current-continuation*, commonly abbreviated *call/cc* [4, 24]. The argument to *call/cc* is a procedure of one argument. The application (*call/cc* p) causes p to be applied to a procedure representing the current continuation. When a continuation created by *call/cc* is applied to a value, execution of the program continues from the point at which the call to *call/cc* occurred, with the value returned as the result of the call to *call/cc*. For example,

```
(call/cc (lambda (k) (+ (k 0) 1)))
```

evaluates to 0.

Suppose we wish to compute the product of a list of numbers, avoiding any multiplications if one or more elements of the list are zero. We can do this by traversing the list recursively, performing the multiplications only after the end of the list has been found, and exiting if we find zero before we find the end of the list:

```
(define product0
  (lambda (ls exit)
    (cond
      ((null? ls) 1)
      ((= (car ls) 0) (exit 0))
      (else (* (car ls) (product0 (cdr ls) exit))))))
```

Using *call/cc*, we can provide $product_0$ with an appropriate continuation that can be used as the value of *exit*:

```
(define product
  (lambda (ls)
    (call/cc
      (lambda (exit)
        (product0 ls exit)))))
```

In the presence of concurrent processing, the simplest uses of continuations can present difficulties. Suppose we wish to add the products of two lists:

```
(+ (product list1) (product list2))
```

The fact that *product* is defined using *call/cc* need not concern the programmer who uses it. However, in a concurrent system, it is no longer clear what is meant by a given *call/cc* or continuation application. Suppose **pcall** is used to allow the products of the lists to be computed concurrently:

```
(pcall + (product list₁) (product list₂))
```

In order for this to work properly, the effects of obtaining and invoking the current continuation within *product* must be local to the corresponding arm of the **pcall** expression.

But suppose we wish to multiply rather than sum the products of the two lists. If the product of one list is zero the combined product will be zero, so the entire calculation may as well be aborted. This can be achieved by passing a suitable escape continuation to $product_0$:

```
(call/cc
  (lambda (k)
    (* (product₀ list₁ k)
       (product₀ list₂ k))))
```

However, if we attempt to compute the product of the two lists concurrently using the same approach we find that we can no longer restrict the effects of continuations to a single branch of the process tree:

```
(call/cc
  (lambda (k)
    (pcall *
           (product₀ list₁ k)
           (product₀ list₂ k))))
```

The intent here is to abort all branches of the **pcall**, whereas before we wished to affect only a single branch. There is, however, no way to make such distinctions with *call/cc*. Continuation operations must affect either the entire process tree or single branches of the process tree; there is no way to designate subtrees.

Problems also arise when continuations are used for modeling process abstractions, such as coroutines [13] and engines [8, 6]. In such cases, continuations must be saved so that processes can be resumed. Again, it is difficult to specify how much of the process tree is to be affected, but another problem also arises. Such applications typically involve a two-part operation: first, the current continuation is captured, and second, another continuation is invoked. Once concurrency is introduced, the delay between the capture of one continuation and the invocation of the other continuation becomes significant and may result unexpectedly in repeated side effects. For example, (*call/cc* (**lambda** (k) (k e)))

may no longer be equivalent to e in all contexts. If it occurs while other processes are executing, side effects might occur between the capture of the continuation and its subsequent invocation, and these side effects might be repeated when the continuation is later invoked. Although this problem can be alleviated by introducing concurrency control operators to give a process exclusive control by suspending other processes, the use of such operators is likely to be expensive and error-prone.

Some of the problems inherent in abortive continuations can be solved by using "functional" continuations. Felleisen, *et al.* [11], introduced a new control operator, **F**, that is similar to *call/cc* in that it captures the current continuation and passes it to its argument. However, **F** differs from *call/cc* in two ways. One difference is that the captured continuation is compositional rather than abortive. When a functional continuation is invoked, it does not replace the current continuation; instead, the value of the computation originally captured by **F** is returned to the continuation in which the functional continuation was invoked. The other difference is that, although the continuation created by **F** does not abort the current continuation, **F** does. That is, the current continuation is aborted at the same time it is captured, rather than when another continuation is invoked. Consequently, none of the functionality of *call/cc* is lost.

The abortive nature of **F** solves one of the concurrency problems. Since **F**, rather than the invoked continuation itself, aborts the current continuation, we no longer have to protect against changes to the computational state during the interval between the capturing of the continuation and the installation of a new continuation. Instead, we can require that **F**, in the presence of concurrency, halt all computation before it captures the current continuation and passes it to its argument. However, since **F** always aborts the complete computation, it still suffers from some of the same problems as traditional continuations.

In a later paper Felleisen introduced the notion of a "prompt" operator (written "#") to provide finer control over **F** [9]. The prompt establishes the base of a computation for subsequent calls to **F**. The continuation captured by **F** extends only to the last prompt, and the current continuation is aborted only to the last prompt. When a value finally returns to a prompt application, it simply falls through to the continuation of the prompt application. Unfortunately, prompts replace the problem of capturing too much of a continuation with the problem of capturing too little of a continuation. Since the continuation captured and aborted by **F** extends only to the last prompt, we have control only over the subtree with the last prompt as its base. Achieving control over larger portions of a process tree requires either complete knowledge of all prompts in the process tree or complicated protocols for recognizing when a control operation arrives at the desired point in the process tree.

3. Subcontinuations

What we lack is a mechanism that allows the program to request the current continuation back to any given point. Prompts allow us to request only the continuation back to a single point, the one established by the last prompt, since all other prompts are "shadowed." It is as if we were programming in a block-structured language that restricts us to one variable name. In order to allow a program finer control over continuations, we introduce the notion of a *subcomputation*. Abstractly, a subcomputation represents a partial computation that can be controlled independently of the computation as a whole. A *subcontinuation* is simply the continuation of that subcomputation, *i.e.*, an abstract entity representing the rest of the subcomputation from a given point in the subcomputation.

The operator *spawn* is used to establish the root of a subcomputation. When applied to a procedural argument, *spawn* invokes (spawns) the procedure as a subcomputation. *spawn* passes the procedure one argument, a *subcomputation controller*. If the controller is never used, the *spawn* has no effect other than to evaluate the body of the procedure. For example, the following evaluates to #t:

```
(spawn (lambda (c) #t))
```

If the controller is invoked, it captures and aborts the current continuation back to and including the root established by the *spawn* invocation. In this manner, the *spawn* invocation and subsequent controller invocation delimit the subcontinuation. If the controller is invoked but the captured continuation is not, the only effect is to abort the part of the continuation between the calls to *spawn* and the controller. For example:

```
(cons 1 (spawn (lambda (c)
                 (cons 2 (c (lambda (k)
                              (cons 3 '()))))))))
```

evaluates to the list (1 3), since the cons of 2 is aborted. The cons of 1 remains, since it is outside of the call to *spawn* and therefore not a part of the subcomputation controlled by *c*.

Invocation of a subcontinuation does not replace (abort) the current continuation; instead, the subcontinuation is composed with the current continuation, as with **F**.

Consider the following similar expression, which evaluates to the list (1 3 2):

```
(cons 1 (spawn (lambda (c)
                 (cons 2 (c (lambda (k)
                              (cons 3 (k '())))))))))
```

When the controller *c* is invoked, the current continuation includes the cons of 1, which is outside of the *spawn*, and the cons of 2, which is inside of the *spawn*. The continuation *k*, therefore, represents the cons of 2, and this much of the current continuation is aborted. Thus, the current continuation at the cons of 3 is just the cons of 1. When *k* is invoked, the current continuation includes the cons of 1 and the cons of 3. The value of the call to *k* is the list (2), which is returned to the cons of 3, whose value is returned, finally, to the cons of 1. As above, the cons of 1 is neither included in the continuation *k* nor aborted by the controller, since it is not within the subcomputation.

The root of a subcontinuation is removed either by a normal return from the subcomputation or by the application of the corresponding controller. As implied above, application of a controller is valid only when the corresponding root is in the continuation of the application. Once the root has been removed, further invocations of the controller are invalid. However, since a subcontinuation includes, at its base, the root of the suspended subcomputation, the controller is again valid when the continuation is reinstated. For instance, in the following example the controller is returned as the result of the call to *spawn* and then applied:

```
((spawn (lambda (c) c))
 (lambda (k) k))
```

Since the controller's root no longer exists, its application is invalid. The following example is also invalid, but for a different reason:

```
(spawn (lambda (c)
         (c (lambda (k)
              (c (lambda (k) k))))))
```

Here the controller is applied twice. The first application (in the second line) is valid. The second application (in the third line) is not valid, since the controller's root has removed from the current continuation by the first application. On the other hand, in the following example both controller applications are valid, since the subcontinuation, including its root, is reinstated before the outermost application occurs:

```
(spawn (lambda (c)
         (c (c (lambda (k)
                 (k (lambda (k) k)))))))
```

The result of this expression is a procedure that returns its argument, since after the second call to the controller nothing remains to be done in the continuation except to return. Several more interesting and useful examples are given in the following section.

In the presence of concurrency, the effect of a control operation must be defined in terms of the branches of a process tree. By "process tree," we mean simply a tree-structured continuation record. Since traditional continuation control operators are derived from the notion of representing continuations as stacks, it is not surprising that such operators are inadequate for controlling concurrency. The *spawn* operator, on the other hand, is designed specifically for the control of tree-structured concurrency.

Each *spawn* application establishes the root of a new subtree, logically representing a subcomputation, and each application of a concurrent operator adds two or more branches to the tree. The application of a controller is valid only if the application occurs in a subtree of the controller's root. Similarly, the continuation created (and aborted) by a controller consists of the entire subtree of its root.

Since subcontinuations can be applied more than once, more than one instance of the same root can occur in a process tree. It is even possible for the subcomputation resumed by a subcontinuation to invoke the subcontinuation itself. (This is true even in the absence of concurrency.) Consequently, we add one more rule: the continuation captured (and aborted) by a controller consists of the smallest complete subtree containing both the controller's root and the controller's application.

One can think of *spawn* as a version of # that creates a new **F** each time it is used; the new **F** recognizes only the root established by this use of #, and the new root is recognized only by the new **F**. If we had an indefinite supply of matched # and **F** operators, we could define *spawn* approximately as $(\lambda p.\#_i(p\mathbf{F}_i))$. However, this definition does not accurately reflect when application of the controller $\mathbf{F}_i$ is valid. **F** captures a continuation only up to a # application; the # application itself is left as part of the continuation of the **F** application. If, instead, **F** captured a continuation up to and including a # application, the approximate definition would be more accurate.

4. Examples

Using *spawn*, nonlocal exits can be established that do not suffer from defects inherent in the use of *call/cc* or # and **F**. Unlike *call/cc*, *spawn* can be constrained easily to ensure that the continuation used to exit from a computation cannot also be used to resume the parent computation. Furthermore, since *spawn* does not need to capture the continuation of its invocation, establishing an exit point with *spawn* does not affect concurrent processes. Also, there is no restriction to a single level of exits as there is with # and **F**. The following example shows how *spawn* can be used to provide a general-purpose nonlocal exit capability:

```
(define spawn/exit
  (lambda (proc)
    (spawn (lambda (c)
             (proc (lambda (exit-value)
                     (c (lambda (p)
                          exit-value))))))))
```

Here *proc* is spawned as a subcomputation that is not given complete access to its controller. Instead, it is given a modified controller that it can use only to abort its computation and return a value. The modified controller invokes the original controller with a procedure that throws away the subcontinuation and returns *exit-value* as the value of the spawned process. Using *spawn/exit*, a computation may exit from any level, since *spawn* operations may be nested arbitrarily. Furthermore, once a computation has returned or has been suspended, use of the exit procedure is invalid.

We can use *spawn/exit* with the $product_0$ procedure defined in Section 2 to add the concurrently-computed products of two lists:

```
(pcall +
       (spawn/exit (lambda (exit)
                     (product0 list1 exit)))
       (spawn/exit (lambda (exit)
                     (product0 list2 exit))))
```

By placing the *spawn/exit* outside of the **pcall**, we can also use it to compute the product of the concurrently-computed products of two lists, aborting both intermediate computations if a zero element is found in either list:

```
(spawn/exit
   (lambda (exit)
      (pcall *
             (product0 list1 exit)
             (product0 list2 exit))))
```

By the placement of *spawn*, or in this case *spawn/exit*, we specify exactly how much of the computation is aborted, avoiding the problems with traditional continuations described in Section 2.

As was the case with the inclusion of *call/cc* in Scheme, including *spawn* in a concurrent programming language reduces the number of control operators that must be supplied as primitives. We can start with a simple forking operator and use it with *spawn* to create sophisticated concurrency operators. For example, it is straightforward to derive **parallel-or** using *spawn* and **pcall**. The semantics of **parallel-or** resemble the semantics of the regular Scheme **or**. The distinction is that **or** evaluates its arguments from left to right, returning the first nonfalse value without evaluating the rest of its arguments, whereas **parallel-or** evaluates its arguments concurrently, returning the value of the first argument to complete with a nonfalse value (and abandoning evaluation of any remaining arguments).

First we define *first-true* using **pcall** and the *spawn/exit* procedure defined above. The *first-true* procedure invokes two zero-arity procedures concurrently and returns either the value of the first procedure to return with a true value, or false if both procedures return false.

```
(define first-true
   (lambda (proc1 proc2)
      (spawn/exit
         (lambda (return)
            (let ((return-if-true (lambda (x) (if x (return x)))))
               (pcall (lambda (x1 x2) #f)
                      (return-if-true (proc1))
                      (return-if-true (proc2)))))))))
```

first-true spawns a subcomputation that uses **pcall** to invoke the procedures $proc_1$ and $proc_2$ concurrently. If either procedure returns a true value, the controller is used to abort the subcomputation and return that value. Otherwise, if both procedures return, their values are discarded and #f is returned. It is now straightforward to define **parallel-or** as a syntactic extension:

```
(define-syntax parallel-or
  (syntax-rules ()
    (( e1 e2)
     (first-true (lambda () e1) (lambda () e2)))))
```

Because the examples above use the controller for nonlocal exits, the subcontinuation created by the controller has not been used. The next example shows how subcontinuations can be used to allow processes to be suspended and resumed:

```
(define parallel-search
  (lambda (tree predicate?)
    (spawn
      (lambda (c)
        (letrec ((search
                   (lambda (tree)
                     (if (not (empty? tree))
                         (pcall
                           (lambda (x y z) #f)
                           (if (predicate? (node tree))
                               (c (lambda (k)
                                    (list (node tree)
                                          (lambda () (k #f)))))
                           (search (left tree))
                           (search (right tree)))))))
          (search tree)
          #f)))))
```

The *parallel-search* procedure takes a tree and a predicate as arguments. Before initiating the search it uses *spawn* to set up a controller that it can use to suspend the search whenever a suitable node is found. **pcall** is used to allow the branches of the tree to be searched concurrently. Since the real results are returned through the controller, the procedure applied by **pcall** ignores the values of its arguments. When *predicate?* is satisfied for a node, the controller is invoked to suspend the search and return a tentative answer along with a procedure that can be used to resume the search. False is returned when there are no more nodes in the tree.

The following procedure uses *parallel-search* to return (in some order) all of the nodes of a tree that satisfy a given predicate:

```
(define find-all
  (lambda (tree predicate?)
    (letrec ((next
               (lambda (result)
                 (if result
                     (cons (car result)
                           (next ((cadr result))))
                     '()))))
      (next (parallel-search tree predicate?)))))
```

Since we have argued that *spawn* is more useful than *call/cc* even in the absence of concurrency, it is natural to wonder whether *spawn* can be used to implement *call/cc*. It can, but in order to do so, we need to have some way to wrap the evaluation of all top-level Scheme expressions with a call to *spawn* to establish the root of the entire computation. The following assumes that we can do so by redefining the top-level read-eval-print loop by altering the value of the variable *top-level-repl*:

```
(let ((old-repl top-level-repl))
  (set! top-level-repl
    (lambda ()
      (spawn (lambda (c)
               (set! call/cc (controller->call/cc c))
               (old-repl))))))

(define controller->call/cc
  (lambda (c)
    (lambda (p)
      ((c (lambda (k)
            (k (lambda ()
                 (p (lambda (v)
                      (c (lambda (kk)
                           (k (lambda () v))))))))))))))
```

Most of the complexity in this code stems from the fact that *call/cc* does not abort the current continuation, whereas *spawn* does, and from the fact that the continuations created by *call/cc* abort the current continuation, whereas those created by *spawn* do not. The call to c on the fourth line of the definition of *controller->call/cc* captures and aborts the continuation; the call to k on the following line puts it back, then calls p (the *call/cc* argument) with an aborting version of the continuation. The aborting version of the continuation uses c to abort the current continuation, then reinstates the saved continuation k. The seemingly redundant additional zero-arity **lambda** expressions and the corresponding invocation on line four are present to ensure that k is invoked to restore the root of the controller before any attempt is made to invoke the controller.

5. Control Filters

The Scheme procedure *dynamic-wind* [15, 6] may be used to perform set-up and clean-up actions on entry to or exit from a given computation, even if exit or entry occurs as the result of a continuation invocation. This procedure accepts three arguments, each of which is a zero-arity procedure: *entry*, *body*, and *exit*. In the absence of continuation operations, the effect of *dynamic-wind* is to invoke first *entry*, then *body*, then *exit*, returning the value returned by *body*. The *entry* and *exit* procedures are invoked only for the side effects they perform, since the values they return are ignored.

When a continuation is used to exit from the computation performed by *body*, *exit* is invoked; when a continuation created during the evaluation of *body* is used to reenter the computation, *entry* is invoked. Thus, whenever *body* is active, *entry* has been invoked most recently, and whenever *body* is not active, *exit* has been invoked most recently. Thus, *entry* and *exit* provide a "barrier" between the computation performed by *body* and computations performed outside *body*. This barrier may be used to set up state variables or objects external to the Scheme system that are needed only within *body*.

Common Lisp [2] provides a similar form, *unwind-protect*, which allows an exit handler to be established within its body. There is no need for an entry handler since Common Lisp continuations are not first-class and can be used only for nonlocal exits.

Control filters provide similar capability in a system with subcontinuations, although with more generality since they operate at a somewhat lower level. Control filters are used to "filter" control operations. A control filter can affect controller invocation, continuation invocation, or both.

A control filter is established by the procedure *control-filter*, which takes takes two arguments. The first argument must be a procedure of one argument, representing the filter as described below. The second argument is a procedure of no arguments representing the body. The procedure *control-filter* arranges to invoke the body in a special continuation that contains the filter.

If the body procedure returns a value, the value is returned to the continuation of the call to *control-filter* and the filter is discarded. If, however, within the body an attempt is made to invoke a controller whose subcontinuation contains the filter, the filter will be invoked in the process of aborting and saving the subcontinuation. The filter receives as an argument a procedure that determines the actual continuation used to resume the suspended subcomputation. The filter must either return this argument or return a new procedure defined in terms of the argument. If no other filters appear within the subcomputation "closer" to the controller call, the argument received by the filter is the identity procedure. Otherwise, it may have been augmented by other filters closer to the controller call.

The control filter can return its argument after performing side effects to restore the program state. This is a common programming paradigm, which we formalize by defining the exit filter *on-exit* in terms of *control-filter*:

```
(define on-exit
  (lambda (exit body)
    (control-filter
      (lambda (p) (exit) p)
      body)))
```

The procedure *on-exit* expects two zero-arity procedures as arguments. The first procedure is invoked only if there is an exit during the invocation of the second procedure.

This is similar to *unwind-protect*, except that *unwind-protect* also invokes the exit handler upon normal exit from the body. It is simple to build an analogous form using *on-exit*:

```
(define spawn-unwind-protect
  (lambda (exit body)
    (on-exit
      exit
      (lambda ()
        (let ((v (body)))
          (exit)
          v)))))
```

It is also common to want to control reentry to a subcomputation. This can also be accomplished using control filters, although the mechanism is slightly more complex since the filter's return value must be modified to ensure that the entry procedure is called before the continuation is invoked. To illustrate the technique, we define an *on-entry* procedure that works much like *on-exit*, except that the first argument is a zero-arity procedure to be invoked on reentry into the body instead of on abnormal exit from the body:

```
(define on-entry
  (lambda (entry body)
    (control-filter
      (lambda (p)
        (lambda (k)
          (p (lambda (x)
               (entry)
               (k x)))))
      body)))
```

In place of the argument p, the return value is a shell wrapped around p that ensures that the entry procedure is invoked before the continuation is reinstated.

We are now ready to define *spawn-dynamic-wind*, which provides functionality similar to that of the traditional *dynamic-wind* operator. Like *dynamic-wind*, *spawn-dynamic-wind* accepts three zero-arity procedures: *entry*, *body*, and *exit*. In the simplest case, *entry* is invoked, then *body*, then *exit*. In addition, when a controller is used to create a subcontinuation that includes a *spawn-dynamic-wind* activation, the corresponding *exit* procedure is invoked, and when the subcontinuation containing the *spawn-dynamic-wind* activation is subsequently reinstated, the *entry* procedure is invoked.

The definition of *spawn-dynamic-wind* in terms of *on-entry* and *on-exit* is straightforward:

```
(define spawn-dynamic-wind
  (lambda (entry body exit)
    (entry)
    (let ((v (on-entry entry (on-exit exit body))))
      (exit)
      v)))
```

Since *on-entry* and *on-exit* do not themselves invoke the *entry* and *exit* procedures, this is done explicitly. The **let** expression is used to capture the value returned by *body* through the *on-exit* and *on-entry* calls, so that this value can be returned from *spawn-dynamic-wind* after the call to *exit*.

6. Engines

This section demonstrates how control filters and subcontinuations can be used to implement *engines*. *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 [14].

The procedure *make-engine* creates an engine from a *thunk*, a procedure of no arguments specifying the computation to be performed by the engine. The computation is run for a limited amount of time by providing the engine with a nonnegative integer representing the number of *ticks* for which the engine is permitted to run. A tick represents a small amount of computation, but is not constrained to be any particular unit. The amount of computation associated with a tick need not be consistent from one tick to the next, although on average, a larger number of ticks results in a larger amount of computation.

In addition to the ticks, an engine must be provided with *complete* and *expire* continuations, represented as procedures. The *complete* continuation must be a procedure of two arguments; it is invoked with the computation's result and the count of remaining ticks if the computation completes before the ticks expire. The *expire* continuation must be a procedure of one arguments; it is invoked with a new engine capable of continuing the computation if the ticks expire before the computation completes.

Engines may be used to implement multitasking. The following defines a version of McCarthy's *amb* operator [20] that multitasks two computations and returns the value of the first to complete:

```
(define amb
  (lambda (t0 t1)
    (let loop ((e0 (make-engine t0)) (e1 (make-engine t1)))
      (e0 1
          (lambda (value ticks) value)
          (lambda (new-e0) (loop e1 new-e0))))))
```

In an earlier article, we showed that engines may be implemented using traditional continuations[8]. The article cites several problems inherent in the interaction between engines and traditional continuations. These problems are exactly those present in the interaction between tree-structured concurrency and traditional continuations. Subcontinuations solve these problems and simplify the engine implementation, especially the implementation of nestable engines.

To implement engines in terms of subcontinuations, we assume the existence of an independent timer mechanism. New timers are created using *make-timer*, which takes no arguments and returns a new timer. A timer is started by invoking it with two arguments: *ticks* and *handler*. The handler is a thunk to be invoked after *ticks* units of computation have been performed. A timer is stopped by invoking it without arguments; in this case, it returns the number of ticks remaining. Two timers running simultaneously are independent in the sense that starting or stopping one does not affect the other, *i.e.*, the other remains stopped or continues to run. Independent timers may be implemented in terms of a single timer provided by the host operating environment. It is also straightforward to generalize the single-timer mechanism described in [8], which relies upon a syntactic extension for **lambda** that causes one tick to be consumed for each procedure call.

The following code implements nestable engines:

```
(define make-engine
  (lambda (thunk)
    (let ((timer (make-timer)) (ticks 0))
      (spawn
        (lambda (c)
          (letrec ((new-engine
                     (lambda (k)
                       (lambda (t complete expire)
                         (set! ticks t)
                         ((k #f) complete expire))))
                   (handler
                    (lambda ()
                      (c (lambda (k)
                           (lambda (complete expire)
                             (expire (new-engine k))))))))
            (c new-engine)
            (spawn-dynamic-wind
              (lambda ()
                (timer ticks handler))
              (lambda ()
                (let ((value (thunk)))
                  (lambda (complete expire)
                    (complete value ticks))))
              (lambda ()
                (set! ticks (timer))))))))))
```

Each call to *make-engine* creates a new timer, initializes a variable to hold the count of ticks remaining while the engine is idle, and spawns a new subcomputation. The controller for the subcomputation is immediately invoked to return from *make-engine* with a new engine. When this engine is invoked, *ticks* is set to the value of the first argument to the engine, and the saved continuation is used to continue from the point where the controller was invoked. *spawn-dynamic-wind* is used to start and stop the timer whenever control enters or leaves the engine. The timer handler employs the controller to abort the computation and passes a new engine created from the resulting continuation to the *expire* procedure. If the computation completes before the ticks expire, the value and remaining ticks are passed to the *complete* procedure.

Care is taken to invoke the *complete* and *expire* procedures in the proper continuation, *i.e.*, the continuation of the call to the engine. Otherwise, tail recursion within the *complete* and *expire* procedures would not be treated properly. This is done by returning to that continuation to invoke the *complete* or *expire* argument. Furthermore, the value of *ticks* for the call to *complete* is obtained within the continuation of the call to the engine rather than from within the engine itself since only then will the timer be disabled and the ticks variable set appropriately, by the "out" handler established by the call to *spawn-dynamic-wind*.

An *engine family* consists of an engine created by *make-engine* and any engines created as a result of invoking an engine in the same family with insufficient ticks for the computation to complete. In the implementation above, each engine shares a controller, timer, and ticks variable with other engines in the same family. As a result, only one engine in an engine family can be active at a time, *i.e.*, an engine cannot invoke directly or indirectly another engine in the same family. The need to nest engines in the same family arises rarely, if ever, in practice, so the added complexity required to relax this restriction does not seem warranted.

The style of nesting implemented by the code above is termed *fair nesting* [14]. With fair nesting, each tick charged to an engine is charged as well to each of its ancestors. Fair nesting results from leaving the timer of a parent engine running while the timer of a child engine is running. Other nesting styles can be implemented by altering the engine implementation or the timer mechanism.

7. Operational Semantics

To clarify the semantics of *spawn* we provide an operational semantics for a call-by-value variant of the λ-calculus extended with assignments and control operators. Although such a language is unrealistically simple, a semantic specification for it can be extended naturally to more complete languages containing *spawn* and *control-filter*. We develop the semantics by starting with a core language consisting of a set of *expressions* (e) defined over sets of *constants* (c) and *variables* (x). Expressions are constants, variable references, procedures, applications or assignments:

$$e \rightarrow c \mid x \mid \lambda x \,.\, e \mid e\, e \mid x := e$$

Since we wish to define a call-by-value variant of the λ-calculus we must distinguish those expressions that denote *values* in the language; namely, constants and procedures:

$$v \to c \mid \lambda x\,.\,e$$

We define a set of global rewrite rules over expression–store pairs. Evaluation proceeds by rewriting an expression–store pair until a value is obtained. The rewrite rules are expressed in terms of *evaluation contexts* [10]. A *context* is an expression containing a "hole," written $\square$. $C[e]$ denotes the expression formed by filling the context C with the expression e. Evaluation contexts in the core language are defined as:

$$C \to \square \mid C\,e \mid v\,C \mid x := C$$

Evaluation contexts serve to specify when a term may be evaluated. Here we have specified left-to-right evaluation of applications, because the argument is in an evaluation context only when the procedure has been reduced to a value.

To allow side effects, we include a *store* θ, which maps variables to values. We avoid introducing a distinct set of *locations* by assuming α-conversion when necessary to preserve hygiene. We also assume the existence of some map:

$$\delta : (\mathit{constants} \times \mathit{values}) \to \mathit{values}$$

to evaluate primitive function applications. Following are the rules for evaluating the core language:

$$\begin{aligned}
\langle C[x]\,,\,\theta\rangle &\Rightarrow \langle C[\theta x]\,,\,\theta\rangle && (1)\\
\langle C[(\lambda x\,.\,e)v]\,,\,\theta\rangle &\Rightarrow \langle C[e]\,,\,\theta[x \leftarrow v]\rangle, x \notin \mathrm{dom}(\theta) && (2)\\
\langle C[x := v]\,,\,\theta\rangle &\Rightarrow \langle C[v]\,,\,\theta[x \leftarrow v]\rangle, x \in \mathrm{dom}(\theta) && (3)\\
\langle C[c\,v]\,,\,\theta\rangle &\Rightarrow \langle C[\delta(c,v)]\,,\,\theta\rangle && (4)
\end{aligned}$$

In conjunction with the definition of an evaluation context, these rules define a left-to-right, applicative order semantics. The distinction between Rule 2 and Rule 3 is that in Rule 2 the store is being extended with a fresh variable (consequently, α-substitution may be necessary), whereas in Rule 3, the variable is assumed to already exist in the store and the assignment merely maps it to a new value.

We now introduce a set of *labels* and a set of operations on labels and expressions that allow us to define the semantics of subcontinuations. To describe these operations, we add to our language *labeled expressions* and *control expressions*:

$$e \to c \mid x \mid \lambda x\,.\,e \mid e\,e \mid x := e \mid l : e \mid e \uparrow l$$

We also extend the definition of evaluation contexts to include the new expressions:

$$C \rightarrow \Box \mid C\,e \mid v\,C \mid x := C \mid l : C \mid C \uparrow l$$

Based on these extensions, we add two more rewrite rules:

$$\begin{array}{rcll} \langle C[l : v]\,,\,\theta\rangle & \Rightarrow & \langle C[v]\,,\,\theta\rangle & (5)\\ \langle C_1[l : C_2[v \uparrow l]]\,,\,\theta\rangle & \Rightarrow & \langle C_1[v\,(\lambda x\,.\,l : C_2[x])]\,,\,\theta\rangle & (6)\\ & & \text{where } l \text{ does not label } C_2 & \end{array}$$

Rule 5 states that a label is removed once the labeled expression has been evaluated. Rule 6 shows how a subcontinuation is created. A control expression is reducible only if it occurs within a labeled expression with a matching label. If it does, the body of the control expression is applied to an abstraction created from the context of the control expression up to and including the matching label. The application itself occurs in a context that does not include the abstracted context. Since a control operation can occur in a context in which there is more than one matching label, the rule specifies that the innermost label determines the applicable context. We say that l labels a context C if $C = C_1[l : C_2]$ for some contexts C_1 and C_2.

Using these label operations, we can define the action of the *spawn* operator:

$$\begin{array}{rcll} \langle C[\mathit{spawn}\ v]\,,\,\theta\rangle & \Rightarrow & \langle C[l : v\,\lambda x\,.\,x \uparrow l]\,,\,\theta\rangle & (7)\\ & & \text{where } l \text{ is a fresh label} & \end{array}$$

A *spawn* operation installs a new label and invokes its argument with a controller that can be used to capture a continuation up to the point at which the subcomputation was spawned. The new label l must be distinct from all other labels in C, v, and θ to prevent inadvertent "label capturing."

We can now add *control-filter* operations to our semantics. Once again we must extend the sets of expressions and evaluation contexts allowed in our language:

$$e \rightarrow c \mid x \mid \lambda x\,.\,e \mid e\,e \mid x := e \mid l : e \mid e \uparrow l \mid \mathit{cf}\ e\ e$$

and

$$C \rightarrow \Box \mid C\,e \mid v\,C \mid x := C \mid l : C \mid C \uparrow l \mid \mathit{cf}\ C\ e \mid \mathit{cf}\ v\ C$$

The rules for rewriting *control-filter* operations are rather complex:

$$\begin{array}{rcll} \langle C[\mathit{cf}\ v_1\ v_2]\,,\,\theta\rangle & \Rightarrow & \langle C[v_2]\,,\,\theta\rangle & (8)\\ \langle C_1[\mathit{cf}\ v_1\ C_2[v_2 \uparrow l]]\,,\,\theta\rangle & \Rightarrow & & (9)\\ \multicolumn{3}{r}{\langle C_1[((v_1\,\lambda x_k\,.\,v_2\,\lambda x_v\,.\,x_k\,\lambda x_\bullet\,.\,\mathit{cf}\ v_1\ C_2[x_v]) \uparrow l)\,\bullet]\,,\,\theta\rangle} & \\ & & \text{where } l \text{ does not label } C_2 & \end{array}$$

Rule 8 is straightforward. It returns the value of the body (v_2) as the result of the *control-filter* operation. Note that, as indicated by the extended evaluation contexts, the control filter is actually evaluated before the body. It is invoked, however, only if the evaluation of the body results in a control operation that attempts to capture a context that includes the *control-filter* operation, as shown in Rule 9. The control filter v_1 is passed a procedure to be used to reinstate the continuation. The dummy parameter $\bullet$ is used to force the evaluation of the body of the corresponding abstraction, which must be delayed until the subcomputation is resumed.

8. Implementation

Continuations are usually represented as a stack of procedure activation records. In the presence of continuations, this stack is often implemented as a linked list to facilitate the capture and invocation of continuations as objects. It is also possible to employ a true stack by copying continuations that have been captured before they are modified [7, 5, 3]. With either implementation, it is possible to place a constant bound on the amount of work that must be performed by the continuation operations regardless of the size of the current continuation[17].

Subcontinuations can be implemented in a similar manner with little additional overhead. Instead of a single stack of activation records, the system maintains a stack of labeled stacks[1]. A call to *spawn* results in the addition of an empty stack to the stack of labeled stacks; this new stack is assigned a unique label associated with the controller created by the call to *spawn*. This label defines the root of the subcomputation. When a controller is invoked, all stacks down to and including the stack with the associated label are removed from the stack of labeled stacks and packaged into a subcontinuation. It is an error if no stack with the appropriate label exists.

When a subcontinuation is invoked, its saved stacks are pushed onto the current stack of labeled stacks. Because the base of the saved stacks is the stack with the label associated with the controller that created the subcontinuation, invocation of the controller is again valid. As mentioned earlier, it is possible to invoke a subcontinuation from within its own subcomputation, resulting in more than one occurrence of the associated label. In this case, the controller removes only the stacks down to and including the topmost stack with the associated label.

A concurrent implementation of subcontinuations can be accomplished

[1]For efficiency, the stack of labeled stacks should be represented as a stack of label–address pairs, where each address is a pointer to a stack segment stored elsewhere.

by using a tree instead of a stack of labeled stacks. A call to *spawn* adds an empty labeled stack to the branch of the tree corresponding to where the call occurs. When the controller is subsequently invoked, the subtree of stacks rooted at the corresponding labeled stack is pruned from the tree and packaged into a continuation. This operation may require cooperation from other processors to suspend concurrently executing branches of the subtree. Some mechanism for mutual exclusion is needed to prevent more than one processor from attempting to remove the same subtree at the same time. When a continuation is invoked, the saved subtree is grafted onto the current tree of stacks.

Because subcontinuations are represented as stacks or trees of stacks, operations involving controllers and subcontinuations are linear with respect to the number of control points (labels and forks) within the subcontinuation rather than with respect to the total number of frames or words within the subcontinuation.

Control filters are implemented straightforwardly using a mechanism similar to that described by Haynes and Friedman for *dynamic-wind* [15]. A list of currently active filters is maintained by the system, and each time a controller is created or reestablished, a pointer to the topmost element of the list of filters is saved with the controller. When a controller is invoked, the elements of the current list above the saved pointer are invoked, topmost element first. Recall that filters receive and return a procedure that may be used to modify the subcontinuation representing the rest of the subcomputation. The first filter is passed the identity procedure; each subsequent filter is passed the procedure returned by the preceding filter. The resulting procedure is passed a representation of the subcontinuation, and the resulting, possibly modified, continuation is used to resume the subcomputation after its root is reestablished. The sublist of filters above the controller's saved pointer is also saved, so that these filters can be reestablished when the root is reestablished.

9. Conclusions

This article has presented subcontinuations, which are based upon a notion of subcomputations, and has demonstrated their usefulness in both concurrent and nonconcurrent settings.

We have limited our discussions of concurrency to tree-structured concurrency, where concurrent subcomputations eventually return (if they terminate) to the parent process that initiates them. The *spawn* operator provides a program with precise control over the tree-structured continuations that result from programming with concurrent operators similar to **pcall**. Using *spawn*, a program is able to achieve nonlocal exits without interfering unnecessarily with concurrent computations, and is also able to suspend and resume selected subtrees of the program's continuation. Some programming languages also provide operations to create independent parallel processes, *i.e.*, processes that do not return to a parent process. Since both tree-structured and other forms of concurrency may coexist in the same language, it is reasonable to define the meaning of *spawn* operations in such situations. One possibility is to treat such combinations of dependent and independent processes as a forest of trees, in which control operations affect only the tree in which they occur.

Subcontinuations were first introduced by the authors in an earlier paper, which referred to subcontinuations as process continuations [16]. A similar mechanism, developed independently, was described in a later paper by Queinnec and Serpette [22]. Our work is based on work by Felleisen, *et al.* [11, 9, 12]. Johnson and Duggan [18] have developed a notion of partial continuations that also extends traditional continuation control in a similar manner. In a related work, Sitaram and Felleisen [23] introduce techniques to constrain the effects of prompts and functional continuations. They do so, however, by developing complicated protocols on top of primitive control structures, and they do not address concurrency issues. Miller [21] does address the issue of using continuations in a parallel Scheme implementation. In his implementation, concurrency is based on *placeholders,* which are similar to Halstead's *futures,* and thus he does not treat the problems inherent in using continuations to control tree-based concurrency. Katz and Weise [19] also address the relationship between continuations and futures, but do not address control of tree-based concurrency.

Acknowledgements: The authors would like to thank the anonymous reviewers for their comments on an earlier version of this article.

References

1. Halstead, Jr., Robert H. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7, 4 (October 1985) 501–538.

2. Steele Jr., Guy L. *Common Lisp, the Language*. Digital Press, second edition (1990).

3. Bartley, David H. and Jensen, John C. The implementation of PC Scheme. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (August 1986) 86–93.

4. Clinger, William, Rees, Jonathan A., *et al.* The revised^4 report on the

algorithmic language Scheme. *LISP Pointers*, 4, 3 (1991).

5. Clinger, William D. and Ost, Eric M. Implementation strategies for continuations. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming* (July 1988) 124–131.

6. Dybvig, R. Kent. *The Scheme Programming Language*. Prentice Hall (1987).

7. Dybvig, R. Kent. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina, Chapel Hill (April 1987).

8. Dybvig, R. Kent and Hieb, Robert. Engines from continuations. *Computer Languages*, 14, 2 (1989) 109–123.

9. Felleisen, Matthias. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (January 1988) 180–190.

10. Felleisen, Matthias and Hieb, Robert. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103 (1992) 235–271.

11. Felleisen, Matthias, Friedman, Daniel P., Duba, Bruce, and Merrill, John. *Beyond Continuations*. Technical Report 216, Indiana University Computer Science Department (1987).

12. Felleisen, Matthias, Wand, Mitchell, Friedman, Daniel P., and Duba, Bruce F. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming* (July 1988) 52–62.

13. Friedman, Daniel P., Haynes, Christopher T., and Wand, Mitchell. Obtaining coroutines with continuations. *Computer Languages*, 11, 3/4 (1986) 143–153.

14. Haynes, Christopher T. and Friedman, Daniel P. Abstracting timed preemption with engines. *Computer Languages*, 12, 2 (1987) 109–121.

15. Haynes, Christopher T. and Friedman, Daniel P. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9, 4 (1987) 582–598.

16. Hieb, Robert and Dybvig, R. Kent. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (March 1990) 128–136.

17. Hieb, Robert, Dybvig, R. Kent, and Bruggeman, Carl. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (June 1990) 66–77.

18. Johnson, Gregory F. and Duggan, Dominic. Stores and partial continuations as first-class objects in a language and its environment. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (January 1988) 158–168.

19. Katz, Morry and Weise, Daniel. Continuing into the future: on the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (June 1990) 176–184.

20. McCarthy, John. A basis for a mathematical theory of computation. In Braffort, P. and Hirschberg, D., editors, *Computer Programming and Formal Systems*, North Holland (1963) 33–70.

21. Miller, James S. *MultiScheme: A Parallel Processing System Based on MIT Scheme*. PhD thesis, Massachusetts Institute of Technology (September 1987).

22. Queinnec, Christian and Serpette, Bernard. A dynamic extent control operator for partial continuations. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages* (January 1991) 174–184.

23. Sitaram, Dorai and Felleisen, Matthias. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3, 1 (January 1990) 67–99.

24. Springer, George and Friedman, Daniel P. *Scheme and the Art of Computer Programming*. MIT Press and McGraw-Hill (1989).

25. Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press (1977).