TECHNICAL REPORT NO. 256

Continuations and Concurrency

by

Robert Hieb and R. Kent Dybvig

Revised: January 1990

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Continuations and Concurrency*

Robert Hieb and R. Kent Dybvig
Indiana University
Computer Science Department
Lindley Hall 101
Bloomington IN 47405

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. This paper presents a new type of continuation, called a *process continuation*, that may be used to control tree-structured concurrency. Just as a traditional continuation represents the rest of a computation from a given point in the computation, a process continuation represents the rest of a *subcomputation*, or *process*, from a given point in the *subcomputation*. Process continuations allow nonlocal exits to arbitrary points in the process tree and allow the capture of a subtree of a computation as a composable continuation for later use. Even in the absence of multiple processes, the precise control achievable with process continuations 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 [16] that provides access to continuations need not directly support many traditional imperative control structures such as loops, "gotos," and exception handlers. This simplifies the language and allows the programmer to create new control structures not anticipated by the language designer. However, traditional continuations 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. In this paper, we present a new type of continuation, called a *process continuation*, that does work well with tree-structured concurrent processing. Just as a traditional continuation represents the rest of a computation from a given point in the computation, a process continuation represents the rest of a *subcomputation*, or *process*, from a given point in the *subcomputation*. Process continuations provide complete control over process trees, 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. Even in the absence of multiple processes, the precise control achievable with process continuations makes them more useful than traditional continuations.

If we use traditional continuations in the presence of tree-structured concurrency, 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. Either approach is inadequate in many cases. Restricting continuations to use within a leaf of the 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 a 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 the process tree the continuation extends; process continuations allow us to do so.

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

To appear in *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

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 [10], which supports both **pcall** and **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 [13] and related constructs such as parallel **and** and **or** operators. Although we concern ourselves primarily with tree-structured concurrency in this paper, we do discuss how our mechanism can be adapted to languages that allow independent concurrent processing.

The programming examples in this paper are written in Scheme, a dialect of Lisp. Scheme is an appropriate choice since it already provides the ability to control continuations; although, as we shall see, the level of control provided is not adequate for controlling concurrency. We show how Scheme can be extended with process continuations in a manner that makes it more suitable for concurrent implementations.

The remainder of the paper is organized as follows. In Section 2, we provide a brief overview of the Scheme language; this section can be skipped by those already familiar with Scheme. In Section 3, we discuss traditional continuation control strategies and show how they are inadequate for controlling concurrent processes. We also discuss some recently proposed continuation control mechanisms and demonstrate that they too fail to provide adequate solutions to concurrent processing problems. In Sections 4 and 5, we introduce process continuations and show how they can be used to control processes in a simple, consistent manner. In Section 6, we present a simple operational semantics for a variant of the λ-calculus with process continuations. In Section 7, we describe how process continuations may be implemented. Finally, in Section 8, we conclude with some remarks about the usefulness of process continuations in the absence of concurrency and about the use of process continuations in the presence of concurrency that is not tree-based.

2 Scheme

The examples in this paper are expressed in Scheme, a dialect of Lisp. Scheme inherits from Lisp its use of prefix notation for all syntactic expressions, including procedure applications. Scheme also inherits many of Lisp's syntactic forms and primitive procedures, especially those used to manipulate lists and symbols. Scheme might also be considered a dialect of Algol-60 [15], as it inherits from Algol-60 its lexical scoping and block structure.

Scheme differs from traditional Lisp dialects and from Algol-60 in that it supports "first-class" procedures. A Scheme procedure can be passed as an argument to another procedure, returned as a value from another procedure, or stored indefinitely while retaining the lexical bindings in effect when the procedure was created. In short, a Scheme procedure is given status equal to that of other Scheme objects, such as numbers, symbols, strings, and lists. The following example defines a procedure, *make-cell*, that accepts any object as an argument (the initial contents of the cell) and returns a pair of procedures that may be used to retrieve or change the value in the cell:

```
(define make-cell
  (lambda (x)
    (cons (lambda () x)
          (lambda (v) (set! x v)))))
```

The **define** expression binds the identifier *make-cell* to the value of the outermost **lambda** expression. A **lambda** expression creates a new procedure with the formal parameters given in the list following the key word **lambda**. The procedure body consists of the expression or expressions following the formal parameter list. The value of the last expression in the body is returned as the value of a procedure application. In this case, the body is itself a procedure application specifying that the value of the identifier *cons* is to be applied to the values of the two inner **lambda** expressions. The **set!** expression in the last line of the program assigns the identifier x the value of the identifier v.

The procedure bound to *cons* creates a *pair*, which is one of the basic Scheme object building blocks. Thus, the result of applying *make-cell* to an object is a pair of procedures. The first procedure of the pair, when applied (to no arguments), returns the current value of the identifier x. The second procedure of the pair, when applied to a single argument, changes the value of the identifier x to this argument. The procedure *car* is used to retrieve the first half of the pair and the procedure *cdr* is used to retrieve the second half of the pair. For example, the value of

```
(let ([x (make-cell 0)])
  ((cdr x) 1)
  ((car x)))
```

is 1. Within the body of the **let** expression, x is bound to the result of applying *make-cell* to 0.

A **let** expression may be thought of as a procedure application, since

```
(let ([x v] ...) e1 e2 ...)
```

is equivalent to

```
((lambda (x ...) e1 e2 ...) v ...)
```

In fact, **let** may be defined as a *syntactic extension* in Scheme, using **extend-syntax**:

```
(extend-syntax (let)
  [(let ([x v] ...) e1 e2 ...)
   ((lambda (x ...) e1 e2 ...) v ...)])
```

This **extend-syntax** expression specifies that any occurrence of a **let** expression following the given pattern is to be converted into the corresponding direct **lambda** application.

Several syntactic forms and procedures not described in this section are used in the examples of this paper. Some are described as they are used; the remainder should be self-explanatory. Detailed descriptions of these syntactic forms and procedures may be found in the references [16, 4]. One of the most interesting Scheme procedures, and the one that plays the most significant role in this paper, is *call/cc*, which is introduced in the following section.

3 Traditional Continuations

Continuations are commonly used in denotational semantics as a basis for deriving the meaning of control operations in imperative languages. Many programming languages provide control operations such as jumps and exits that modify a program's continuation. The programming language Scheme makes continuations available as procedures via the procedure *call-with-current-continuation*, commonly abbreviated *call/cc*. The argument to *call/cc* is itself a procedure of one argument, which is passed a procedure representing the continuation of the *call/cc* application. 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*. Either *call/cc* and continuations must affect the entire process tree or they must affect 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 [11] and engines [6, 4]. In such cases, continuations must be saved so a process 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. 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, in which case 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.* [8], 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 then 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 is still inadequate for controlling tree-structured concurrency.

In a later paper Felleisen introduced the notion of a "prompt" operator (written "#") to provide finer control over F [7]. 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 only extends 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.

4 Process Continuations

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 *process*. Abstractly, a process represents a subcomputation that can be controlled independently of the computation as a whole. A *process continuation* 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 create processes. When applied to a procedural argument, *spawn* invokes (spawns) the procedure as a process. *spawn* passes the procedure one argument, a *process controller*. When a process is spawned, a unique root is added to the process tree. When a controller is invoked, it captures and

aborts the current continuation back to the root established by the invocation of the *spawn* that created the controller. Application of a controller is valid only when its root is in the continuation of the application. The continuation returned by a controller is also a process, since, when it is invoked, its root is reinstated and subsequent applications of the controller are valid. Invocation of a process continuation does not replace (abort) the current continuation; instead, the process continuation is composed with the current continuation. The root of a process continuation is removed either by a normal return from the spawned process or by the application of the process controller. Once the root has been removed, further invocations of the controller are invalid until the process continuation has been 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 been removed from the current continuation by the first application. On the other hand, in the following example both controller applications are valid, since the process continuation, 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 creates a new process subtree with a unique root, and each application of a concurrent operator adds two or more branches to a process tree. The application of a controller is valid only if it occurs in a subtree of its root. Similarly, the continuation created (and aborted) by a controller consists of the entire subtree of its root. Since process continuations can be applied more than once, more than one instance of the same root can occur in a process tree. 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 $(\lambda p.\#_i(p\mathcal{F}_i))$. However, this definition does not accurately reflect when application of the controller $\mathcal{F}_i$ 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.

5 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 process that is not given complete access to its controller. Instead, it is given a mod-

ified controller that it can use only to abort its computation and return a value. The modified controller invokes the real controller with a procedure that throws away the process continuation 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 3 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 3.

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 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 procedure *spawn/exit* defined above. The procedure *first-true* applies two procedures concurrently and returns either the value of the first procedure to return with a true value, or false if neither procedure returns a true value.

```
(define first-true
  (lambda (proc1 proc2)
    (spawn/exit
      (lambda (return)
        (pcall (let ([v (proc1)])
                 (if v
                     (return v)
                     (lambda (v) v)))
               (let ([v (proc2)])
                 (if v
                     (return v)
                     v)))))))
```

first-true spawns a process that uses **pcall** to evaluate the procedures $proc_1$ and $proc_2$ concurrently. If either procedure returns a true value, the process controller is used to abort the process and return that value. Otherwise, an identity procedure and a false value will be returned as the arguments to **pcall**, resulting in the return of a false value from the call to *first-true*. It is now straightforward to define **parallel-or** as a syntactic extension:

```
(extend-syntax (parallel-or)
  [(parallel-or e1 e2)
   (first-true (lambda () e1) (lambda () e2))])
```

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

```
(define parallel-search
  (lambda (tree predicate?)
    (spawn
      (lambda (c)
        (define search
          (lambda (tree)
            (unless (empty? tree)
              (pcall
                (lambda (x y z) #f)
                (when (predicate? (node tree))
                  (c (lambda (k)
                       (cons (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 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 all of the nodes of a tree that satisfy a given predicate:

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

6 Semantics

To clarify the semantics of *spawn* we provide an operational semantics for the λ-calculus extended with control operators. Although such a language is unrealistically simple, a semantic specification for it can be extended naturally to more complete languages containing the *spawn* operation. To the usual λ-calculus expression types *constants, variables, abstractions* and *applications* we add *labeled expressions* and *control expressions*:

$$\begin{array}{rll} e & \rightarrow c & \text{(constants)} \\ & \mid x & \text{(variables)} \\ & \mid \lambda x\,.\,e & \text{(abstractions)} \\ & \mid e\,e & \text{(applications)} \\ & \mid l : e & \text{(labeled expressions)} \\ & \mid e \uparrow l & \text{(control expressions)} \end{array}$$

The set $v \in$ *values* consists of constants and abstractions. In the operational semantics values represent terms that cannot be further evaluated and may be passed as arguments or returned as answers. The set $l \in$ *labels* can be any countable set.

A program is an expression with no free variables. We define a machine that rewrites a program until it is a value. The rewrite rules rely on the notion of a *context*, which is an expression with a single hole. The symbol $\Box$ represents the hole in a context. A context is filled by substituting an expression for the hole, resulting in a complete expression. The notation $C[e]$ indicates the operation of filling the context C with the expression e. Here we are interested in *evaluation contexts*:

$$C \rightarrow \Box \mid C\,e \mid v\,C \mid l : C$$

Evaluation contexts determine when a term may be evaluated. Here we have specified leftmost, outermost evaluation.

By using evaluation contexts we are able to dispense with a separate control component for our machine. Our machine is further simplified by using β-substitution instead of an environment to record the results of applications; this simplification is made possible by the lack of assignments. Consequently, we are able to define a machine in terms of only one component, the program. A program is evaluated by rewriting it according to the following rules:

$$\begin{array}{rcll} C[(\lambda x\,.\,e)v] & \Rightarrow & C[e[x \leftarrow v]] & (1) \\ C[l : v] & \Rightarrow & C[v] & (2) \\ C_1[l : C_2[e \uparrow l]] & \Rightarrow & C_1[e\,(\lambda x\,.\,l : C_2[x])] & \\ & & \text{if } l \text{ does not label } C_2 & (3) \end{array}$$

The first rule is the usual β-substitution rule for reducing applications in the call-by-value λ-calculus. The second rule specifies that the value of a labeled expression is simply returned to the immediate context. The interesting rule is the last rule, which determines how control expressions are evaluated. 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.

The labeling primitives are used to define the *spawn* operation. The only complication is that each *spawn* application needs a unique label that can be used both to label the context of the application and to build a process controller that can capture the correct context. Given a unique label l, we could define *spawn* as $(\lambda x\,.\,l : x\,(\lambda x\,.\,x \uparrow l))$. The simplest way to find such a label without complicating the semantics is to examine the entire program. Thus, we define *spawn* applications using a rewrite rule that has access to the entire program:

$$\begin{array}{rcl} C[\mathit{spawn}\,v] & \Rightarrow & C[l : v\,(\lambda x\,.\,x \uparrow l)] \\ & & \text{where } l \notin \mathit{labels}(C[v]) \end{array}$$

where $\mathit{labels}(e)$ is the set of labels occurring in e.

7 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 [3, 2, 1]. 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 [5].

Process continuations can be implemented in a similar manner. However, instead of a single stack of activation records, the system maintains a stack of labeled stacks, the *process stack*. A call to *spawn* results in the addition of an empty stack to the process stack; this new stack is assigned a unique label associated with the process controller created by the call to *spawn*. This label defines the root of the process. When a process controller is invoked, all stacks down to and including the stack with the associated label are removed from the process stack and packaged into a process continuation. It is an error if the process stack contains no stack with the appropriate label.

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

A concurrent implementation of process continuations can be accomplished by using a process tree instead of a process stack. A call to *spawn* adds an empty labeled stack to the branch of the tree in which the call occurs. When the process 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 process continuation is invoked, the saved subtree is grafted onto the current tree of stacks. Because continuations are represented as stacks or trees of stacks, operations involving process controllers and process continuations are linear with respect to the number of control points (labels and forks) within the process continuation rather than with respect to the size of the process continuation itself.

8 Conclusions

In this paper we have dealt with tree-structured concurrency, where concurrent computations eventually complete and return to the parent process that initiated 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 save and restore 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.

Although we have focussed on the usefulness of process continuations in the presence of concurrency, it should also be noted that even without concurrency there are clear advantages and no disadvantages to providing *spawn* in lieu of *call/cc*. One common criticism of *call/cc* is that it is too powerful, in that it always manipulates the continuation of an entire program. Programs written with *spawn* are more easily analyzed, because the effects of a process controller created by *spawn* are limited to the dynamic context of the call to *spawn* and because access to the controller can be restricted.

Our work is based on work by Felleisen, *et al.* [8, 7, 9]. In [8] and [7], new continuation control mechanisms are introduced. In [9], a continuation algebra is developed that makes it convenient to specify the semantics of different models of continuations. Johnson and Duggan [12] have developed a notion of partial continuations that also extends traditional continuation control. However, none of these mechanisms are adequate for process-oriented programming. In a related work, Sitaram and Felleisen [17] 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 [14] does address the issue of using continuation control in a parallel Scheme implementation. In his implementation, concurrency is based on *placeholders*, which are simi-

lar to Halstead's *futures,* and thus he does not treat the problems inherent in using continuations to control tree-based concurrency.

References

[1] David H. Bartley and John C. Jensen, "The Implementation of PC Scheme," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming,* August 1986, 86–93.

[2] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost, "Implementation Strategies for Continuations," *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming,* July 1988, 124–131.

[3] R. Kent Dybvig, *Three Implementation Models for Scheme,* University of North Carolina at Chapel Hill Department of Computer Science Technical Report #87-011 (PhD Dissertation), April 1987.

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

[5] R. Kent Dybvig and Robert Hieb, "Representing Control in the Pressence of First-Class Continuations" (submitted for publication).

[6] R. Kent Dybvig and Robert Hieb, "Engines from Continuations," *Computer Languages 14,* 2, 1989, 109–123.

[7] Matthias Felleisen, "The Theory and Practice of First-class Prompts," *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages,* January 1988, 180–190.

[8] Matthias Felleisen, Daniel P. Friedman, Bruce Duba and John Merrill, "Beyond Continuations," Indiana University Computer Science Department Technical Report No. 216, 1987.

[9] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman and Bruce F. Duba, "Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps," *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming,* July 1988, 52–62.

[10] Robert H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on on Programming Languages and Systems 7,* 4 October 1985, 501–538.

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

[12] Gregory F. Johnson and Dominic Duggan, "Stores and Partial Continuations as First-Class Objects in a Language and its Environment," *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages,* January 1988, 158–168.

[13] John McCarthy, "A Basis for a Mathematical Theory of Computation," *Computer Programming and Formal Systems,* ed. by P. Braffort and D. Hirschberg, North Holland, 1963, 33–70.

[14] James S. Miller, *MultiScheme: A Parallel Processing System Based on MIT Scheme,* Laboratory for Computer Science, Massachusetts Institute of Technology Technical Report #402 (PhD Dissertation), September 1987.

[15] Peter Naur, et al., "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM 6,* 1, January 1963, 1–17.

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

[17] Dorai Sitaram and Matthias Felleisen, "Control Delimiters and their Hierarchies," to appear in *Lisp and Symbolic Computation.*