Continuations: Multiple Invocations Considered Harmful

Luis Mateu and José M. Piquer
e-mail: {jpiquer,lmateu}@dcc.uchile.cl
Dpto. Cs. de la Computación
U. de Chile

July 17, 1997

Abstract

In this paper, we propose to replace Scheme first class continuations by a sequential thread system. Our threads are very close to continuations but they are easier to understand and to use for programmers, and its implementation is simpler and more efficient.

We show that with our threads we can recover almost completely the Scheme operator `call-with--current-continuation` (`call/cc` for short). The only functionality we dismiss is what makes this operator difficult to understand and to implement: the multiple invocations of the same continuation. We think this difference will affect the behavior of only a few programs written in Scheme, but it will allow a faster execution of all programs. Moreover, our threads can be naturally extended to a concurrent version of Scheme, without the known problems of continuations.

1 Introduction

Based on a minimal operation set principle, Scheme [Clinger and Rees 91] offers only one abstraction to implement advanced control structures: first class continuations [Clinger 87]. Dynamic escapes, coroutines and many other control structures can be implemented from continuations, without adding more special forms to the language. However, continuations are difficult to understand and to use, and its semantics is too complex to be well comprehended by programmers, specially when multiple invocations are used. There is no programming paradigm associated with continuations, and thus the programs using them are difficult to read. Continuations do have a well-defined semantics [Strachey and Wadsworth 74], but this does not make them a useful tool for computer programming.

On the other hand, coroutines and dynamic escapes implemented with continuations are very inefficient compared to a native implementation. The main problem is that the multiple invocation semantics of continuations makes them heavier to implement, even though coroutines and dynamic escapes do not need multiple invocations. Surprisingly, it is difficult to find examples where this multiple invocation semantics is really useful.

In this paper, we propose to replace the Scheme continuations by another abstraction based on the *thread* concept. Like continuations, a thread is represented by a first class procedural object. When this object is invoked, a context switch is performed to transfer control to that thread.

Our proposed threads are almost compatible with Scheme continuations. Programs using continuations, but invoking them only once, will remain correct. The only incompatibility is that our threads behave differently when invoked many times, but it enables a behavior easier to understand and a more efficient implementation. We believe that this change will only affect a reduced set of the existing Scheme programs, because the multiple invocations of continuations are rarely used.

The paper is organized as follows: in section 2 we describe the thread concept, in section 3 we describe our proposal for Scheme. An example of a program using our threads is shown in section 4 and a comparison with Scheme continuations is done in section 5. Finally we conclude in section 6.

2 Threads

Most of today's programs are single-threaded. This means that their execution follows a sequence of calls to and returns from procedures in a strict LIFO order.

When a program is multi-threaded, its execution follows many threads. Each one of these threads executes a LIFO sequence of calls and returns, but procedures in two different threads generally do not follow a specific order. At execution time, all threads share the same memory space.

To execute a multi-threaded program, the programming language (or the operating system) must provide a thread system. In this paper we propose a sequential thread system for Scheme, meaning that, at any time, only one thread is executing, calling it the *active* thread. All the other threads are suspended. The active thread has full control of the CPU until it explicitly gives control to another thread[1].

On the other hand, in a concurrent thread system there can be many active threads at the same time, which can exploit paralellism in a multi-processor architecture. A thread system sharing a unique CPU via time-slicing is concurrent, because the programmer's view is the same as in a system executing threads on a multi-processor.

Our thread system can be extended to a concurrent one naturally and without the known problems of mixing concurrency and continuations [Katz and Weise 90].

Thread Diagrams

A thread diagram shows graphically the procedure calls and returns during a time interval. We will use such diagrams to show the meaning of the operations of our thread system. For example, Figure 1 shows a single-thread execution diagram.

The vertical axis represents the execution time, progressing downwards. The horizontal axis represents the procedure call nesting, i.e. the stack size. A procedure call is represented as a curve shift to the right, and it can be labeled with the procedure's name. A return is represented as a shift to the left.

In the Figure 1 we can see that, at instant t' the procedure R is executing, ant it was called from Q,

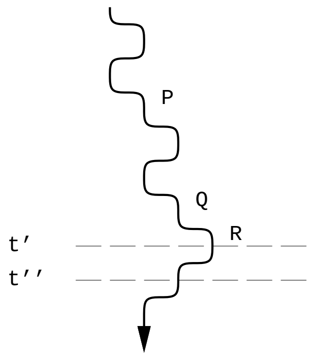

Figure 1: A single-threaded execution

which was called from P. At time t'' R has returned from procedure Q.

Figure 2 shows an example of a sequential multi-threaded system.

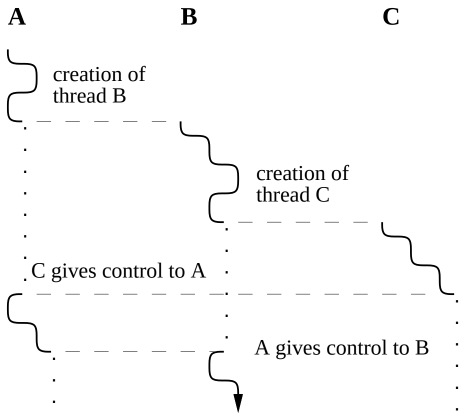

Figure 2: A multi-threaded execution

When a thread is suspended, a dotted line shows that the thread does not execute any instructions. A thread only resumes when the active thread gives the control back to it.

3 A sequential thread system for Scheme

The main primitive introduced is `start-thread`. This primitive allows the creation of new threads, using the following syntax[2]:

[1]In this sense, a sequential thread is equivalent to a coroutine.

[2]The `start-thread` syntax is identical to `call/cc` for partial compatibility. We will discuss compatibility issues later.

```
(start-thread proc)
```

The argument `proc` must be a one-argument procedure. The call to `start-thread` creates a new thread which receives the control immediately, executing `proc`. The new thread (named the child) calls `proc`, passing it as argument the thread identifier of the caller of `start-thread` (named its parent). In Figure 3 we can see the effect of a call to `start-thread`.

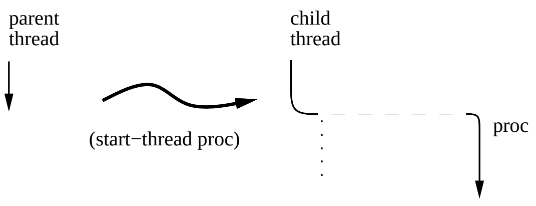

Figure 3: Execution of `start-thread`

The left size shows the moment just before calling `start-thread`. The call to `start-thread` generated the change shown at the right size.

The thread identifier

Each thread has a unique thread identifier. Following the Scheme tradition, this identifier is a procedural object, which can be called as a one-argument procedure. Invoking a thread identifier means to pass the control to that thread, suspending the currently active thread.

Besides `start-thread`, the only other primitive manipulating thread identifiers is `current-thread` which returns the current thread identifier. For example:

```
     ;; this is the thread A
(1)  (define *t-b* '())

(2)  (start-thread
(3)    (lambda (t-a)
         ; this is a new thread B.
         ; t-a is the thread-id of A.
         ; First save the thread-id of B
(4)      (set! *t-b* (current-thread))
         ; pass control to thread A.
(5)      (t-a 'any-1)
         ; B is suspended until A
         ; passes control back to B
(6)      (display "B resumes")
(7)      (t-a 'any-2) ))

(8)  (display "A resumes")
(9)  (*t-b* 'any-3)
(10) (display "A resumes")
```

Figure 4 shows the execution of this code.

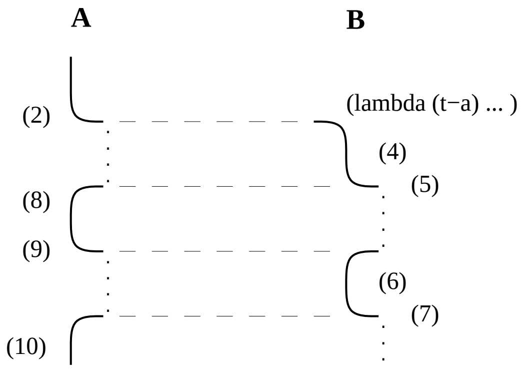

Figure 4: Thread Execution Example

The detailed execution is as follows:

- At (2) thread A calls `start-thread` to create thread B. Control continues at B.
- Thread B calls the lambda expression (3).
- At (4) thread B calls `current-thread` and it stores its thread identifier in variable `*t-b*`.
- At (5) thread B invokes A's identifier to transfer control to it[3].
- Thread A resumes execution returning from `start-thread` (2), displaying its first message (8).
- At (9) thread A invokes B's identifier to pass control to B.
- Thread B resumes execution (5) and displays a message (6).
- At (7) thread B pass control back to A again.
- Thread A resumes execution (9) and displays a new message (10).

[3]When invoking a thread identifier, an argument must be supplied. In this case it is not useful, but it can be used to supply values to the resumed thread.

Pass Value

When a thread transfers the execution control to another thread, we will call the former the *emitter* thread and the latter the *receiver* thread. The emitter can only transfer the control in two ways: (i) invoking the thread identifier of another thread or (ii) creating a new thread with `start-thread`. Every thread that is not active must be suspended in a call of type (i) or (ii).

When a control transfer of type (i) ocurrs, the emitter must supply a value of any type to the receiver. We will call this value the *pass value*. At the emitter, this value is the only argument to the thread identifier. At the receiver, it is the return value of the type (i) or (ii) expression in which it was suspended.

Active Thread Identifier

Any thread can obtain its own identifier through a call to `current-thread`. This procedure does not need to be primitive, because a thread could obtain its own identifier with the following expression:

```
      (set! my-thread
        (start-thread
(1)       (lambda (thread)
(2)         (thread thread) )))
```

In this code, a thread is created receiving its parent identifier at (1) and returning control to its parent immediately, with the identifier as the pass value in (2). Of course, `current-thread` could also be implemented as a procedure calling `start-thread` in the same way.

However, it would be very inefficient to create a new thread just to compute our own identifier, an operation than could be a frequent one. This is why it is better to provide a primitive version or `current-thread`, with an efficient implementation.

Finishing a Thread

When the initial procedure of a thread (the procedure passed as argument to `start-thread`) returns, the thread is finished. In this case the execution control passes automatically to its parent and the return value of the procedure becomes the pass value. If the main thread (where the program execution began) finishes, then the whole program ends.

For example, in the last code example, the lambda expression could be simplified to just `(lambda (thread) thread)`. It is an execution error to transfer control to a finished thread.

4 A same-fringe implementation with threads

A classic problem, difficult to solve efficiently in a single-threaded system, is to compare the fringes from two different trees. A tree fringe is the sequence of leaves obtained traversing the tree from left to right. The idea is to visit only the matching leaves and to stop the execution at the first difference.

This solution uses three threads, one for visiting each tree and a third to compare the results. Using our threads the solution is simple and efficient.

The procedure to perform the comparison is `same-fringe`. This procedure calls `make-walker` to start the tree traversals.

```
     ; The end of tree token
(1)  (define *eot* '(eot))

(2)  ; Compares the fringes of two trees
(3)  (define (same-fringe tree-a tree-b)
       ; Create one thread for each tree
(4)    (let ((t-a (make-walker tree-a))
(5)          (t-b (make-walker tree-b)))
         ; Now get a leaf for each tree
(6)      (let loop ((leaf-a (t-a 'any))
(7)                 (leaf-b (t-b 'any)))
(8)        (cond
(9)          ((eq? leaf-a *eot*)
                ; no more leaves on tree-a
(10)            (eq? leaf-b *eot*))
(11)         ((eq? leaf-a leaf-b)
                ; get a new leaf for each
                ; tree and loop
(12)            (loop (t-a 'any) (t-b 'any)))
(13)         (else
                ; found two different leaves
(14)            #f) ))))

(15) (define (make-walker tree)
(16)   (start-thread
(17)     (lambda (t-comp)
           ; t-comp is the thread-id of the
           ; comparator thread
(18)       (define (walk-tree tree)
             ; A local procedure to traverse
```

```
               ; a tree
(19)           (if (atom? tree)
(20)             (t-comp tree)
                 ; else
(21)             (begin
(22)               (walk-tree (car tree))
(23)               (walk-tree (cdr tree)) )))
             ; pass my thread-id
(24)         (t-comp (current-thread))
             ; now traverse the tree
(25)         (walk-tree tree)
             ; Finally, pass the end-token
(26)         *eot* )))
```

The execution is as follows: when `same-fringe` requires in (6) or (7) the first leaf of a tree, it invokes the corresponding thread identifier. This thread initiates a recursive traversal of the tree in (25) until a leaf is found in (20). At this point the thread passes control back to `same-fringe`. When `same-fringe` needs a new pair of leaves at (12) it resumes once again the walker threads.

This solution is efficient because it only visits the leaves that are equal, stopping its execution at the first difference in (13). The walker threads remain suspended afterwards, until the garbage collector reclaims them.

5 Comparison with `call/cc`

Our thread system uses a syntax for `start-thread` that is identical to `call/cc`. However they create different objects: threads and continuations. Even though threads and continuations are not the same, in this section we show that just replacing `call/cc` by `start-thread` will produce the same results for most programs.

Table 1 compares `call/cc` with `start-thread`. Table 2 compares Scheme continuations with our threads, being both procedural first-class objects.

The main differences between continuations and threads are then:

- The invocation of a continuation never returns. The invocation of a thread identifier returns when another thread invokes the suspended thread identifier.

- All the invocations of a continuation return at the same `call/cc` which yielded that continuation. This means that multiple invocations of the same continuation generate multiple returns of the same `call/cc`. On the other hand, multiple invocations of the same thread identifier return at the different points where control was suspended.

The following example shows the difference between a continuation and a thread.

```
     (define *P-k* 'still-undefined)
     (define *Q-k* 'still-undefined)
     (define *v* 'still-undefined)
     ;
     (define (P a)
(1)    (call/cc Q)
(2)    (set! a (+ a 1))
(3)    a )
     (define (Q k)
(4)    (set! *P-k* k)
(5)    (call/cc R)
(6)    (*P-k* 'dummy) )
     (define (R k)
(7)    (set! *Q-k* k)
(8)    (*P-k* 'dummy) )
     ; main program
(9)  (set! *v* (P 0))
(10) (display *v*)
(11) (*Q-k* 'dummy)
(12) (display "end")
```

In Figure 5, at the left size we can see the execution of the code above. At the right size, we see the execution replacing `call/cc` by `start-thread`. With `call/cc`, the program loops displaying the sequence 1, 2, 3, 4, etc. With `start-thread`, it displays 1 and then it exits displaying the message `"end"`.

In the example with `call/cc`, the unique call to P at (9) has multiple returns. This happens because the continuation stored in `*P-k*` is invoked many times from (6) and (8).

Replacing `call/cc` by `start-thread`

Let us introduce the following redefinition in Scheme:
`(set! call/cc start-thread)`

As shown in the previous example, programs invoking many times the same continuation will be affected.

Let us examine what happens to a program invoking continuations only once. With such a redefinition, `call/cc` now returns a thread instead of a continuation, but both abstractions behave identically if

primitive	`call/cc`	`start-thread`
syntax	`(call/cc proc)`	`(start-thread proc)`
form of `proc`	`(lambda (k) ...)`	`(lambda (thread) ...)`
argument type	first-class continuation	first-class thread identifier

Table 1: Comparing `call/cc` and `start-thread`.

object	first-class continuation	first-class thread identifier
syntax	`(k` *val*`)`	`(thread` *val*`)`
returns?	no	yes
first invocation effect	return from `call/cc`	return from `start-thread`
pass value	*val*	*val*
next invocations effect	return from same `call/cc`	thread resumes at the point where it was active last time

Table 2: Comparing continuations and threads.

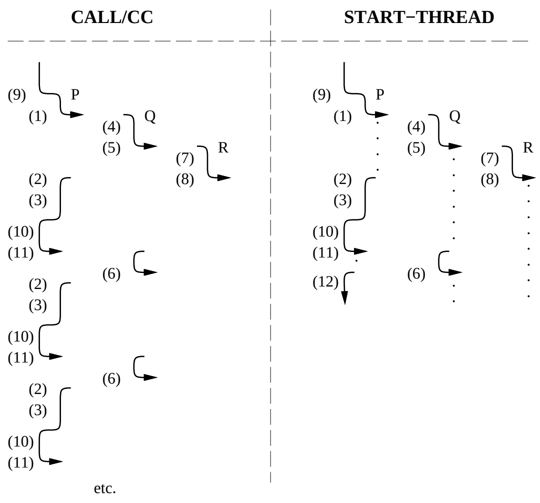

Figure 5: `call/cc` vs `start-thread`

invoked only once. So, for such a program, the redefinition should not change its execution. However, the following expression will evaluate to a different result, even though it invokes continuations only once:

```
(eq? (call/cc (lambda (k-1) k-1))
     (call/cc (lambda (k-2) k-2)) )
```

In Scheme this expression evaluates to false, because the continuation passed to `lambda` is different for each call. Replacing `call/cc` by `start-thread` the expression compares the parent identifier of each thread, which is the same, returning true.

Therefore, the only programs affected by this redefinition would be those invoking multiple times the same continuation and those comparing continuations with `eq?`. Based on that, we conclude that the number of programs affected by this change is minimal.

The `start-thread` semantics

In this section we describe the `start-thread` semantics in terms of `call/cc`. This is only useful as a reference for real implementations of `start-thread`, because we really propose to eliminate the `call/cc` operator from Scheme.

```
(define current-thread 'still-undefined)
(define start-thread 'still-undefined)

(let ( (*curr-thread* 'dummy)
       (*curr-switcher* 'dummy) )

  (define (make-thread next-k)
    (define (switcher k-receiver val)
      (call/cc
        (lambda (new-k)
          (set! next-k new-k)
          (k-receiver val) )))
    (define (thread val)
      (let ((old-switcher *curr-switcher*))
        (set! *curr-thread* thread)
        (set! *curr-switcher* switcher)
        (old-switcher next-k val) ))
    (thread *curr-thread*) )

  ; initialization for main thread
  (set! *curr-switcher*
    (lambda (proc val) 'dummy-switcher) )
  (make-thread 'never-invoked)

  (set! current-thread
    (lambda () *curr-thread*) )

  (set! start-thread
    (lambda (proc)
      (make-thread
        (lambda (parent-thread)
          (parent-thread (proc parent-thread))
          (error "can't resume a dead thread")
) ) ) ) )
```

It is worth noting that this implementation always invokes a continuation once. Thus, `call/cc` can be replaced by a call to a pre-existant `start-thread`. The new operator `start-thread` thus defined, is semantically correct.

5.1 Implementation Efficiency

Our threads can be implemented just using one stack per thread[4]. The `start-thread` operator could be efficiently implemented in C with some lines in assembler, with all the usual optimizations, as it is done for threads in languages like C, C++, Java, etc.

However, this scheme is not valid to implement `call/cc`. Due to the multiple invocations semantics, it is not always possible to free the activation frame of a returning procedure. Since a continuation could have been captured during the procedure execution, the activation frame could be used again.

Many Scheme implementations [Clinger *et al* 88, Hieb *et al* 90] use the stack to store activation frames while `call/cc` is not used. Upon a call to `call/cc` or to a continuation, the stack must be copied totally or partially. Making copies of the stack has the following problems:

- Programs using continuations are not efficient. Copying stack segments is slow, and implementing a thread system based on continuations is much slower than a native thread system like our proposal.
- Even programs not using continuations are inefficient: to respect the `call/cc` semantics, any variable subject to a side-effect through a `set!` must be created on the heap. These variables will only be deallocated by the garbage collector, meaning an overhead in execution time. It is worth noting that this implies that `call/cc` can not be implemented just as an special form, the compiler must generate ad-hoc code.

The other existing variations of Scheme implementations are also inefficient when using `call/cc` and they introduce an important execution overhead even when not using `call/cc`.

The most surprising thing is that all these problems are related to the multiple invocations semantics of `call/cc`, a feature rarely used. Just prohibiting the multiple invocation, would make `call/cc` easier to understand and more efficient to implement.

6 Conclusions

In this paper we have shown how a thread system could replace the Scheme continuations, simplifying the language and its implementation.

Scheme justifies the presence of continuations in the language [Haynes *et al* 84] as a flexible base to solve many control problems, like coroutines and dynamic escapes. We claim that our system is flexible and general enough to fulfill the same mission.

The change we are proposing could affect existing programs, when invoking multiple times the same continuation or when comparing them with `eq?`.

[4]Although implementations using multiple stacks are time-efficient, they are space-inefficient. In [Mateu 92] we propose a heap-based implementation being space and time efficient.

However, we think that the affected programs will be less than those broken by the change of the evaluation of `'()` from false to true [IEEE 90].

Currently, the multiple invocation of a continuation is not useful in practice. They are there because they appear naturally from the conversion to CPS (Continuation Passing Style) [Strachey and Wadsworth 74, Steele 78], but when such multiple invocations are used, the resulting program is unreadable. Moreover, when extending the language to a concurrent execution, multiple invocations are difficult to integrate into the new system[Katz and Weise 90].

Our main point is to redefine the semantics of multiple invocations based on threads. This semantics is useful as a programming tool, flexible to implement advanced control structures, easy to extend to any concurrent Scheme and efficient to implement.

References

[Clinger 87] William Clinger : "The Scheme Environment : Continuations", *Lisp Pointers*, Vol. 1 (2), 22–28, June-July, 1987.

[Clinger *et al* 88] William D. Clinger, Anne H. Hartheimer and Eric M. Ost : "Implementation Strategies for Continuations", *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, 124–131, 1988.

[Clinger and Rees 91] William Clinger and Jonathan A. Rees : "Revised^4 Report on the Algorithmic Language Scheme", *ACM Sigplan Notices*, Vol. 21 (12), December 1991.

[Haynes *et al* 84] Christopher T. Haynes, Daniel P. Friedman and Mitchell Wand : "Continuations and Coroutines", *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 293–298, 1984.

[Hieb *et al* 90] Robert Hieb, R. Kent Dybvig and Carl Bruggeman : "Representing Control in the Presence of First-Class Continuations", *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, 66–77, White Plains, New York, June 1990.

[IEEE 90] *IEEE Standard 1178-1990. IEEE Standard of the Scheme Programming Language*, IEEE, New York, 1991.

[Katz and Weise 90] Morry Katz and Daniel Weise : "Continuing into the Future: On the Interaction of Futures and First-Class Continuations", *ACM Conference on Lisp and Functional Programming*, Nice, France, 1990, pp. 176–184.

[Mateu 92] Luis E. Mateu : "An Efficient Implementation of Coroutines", *International Workshop on Memory Management*, Y. Bekkers, J. Cohen (Editors), LNCS 637, pp. 230–247, Springer-Verlag, Saint-Malo (France), September 1992.

[Steele 78] Guy L. Steele Jr. : "Rabbit : a Compiler for Scheme", MIT AI Memo 474, M.I.T., Cambridge, 1978.

[Strachey and Wadsworth 74] C. Strachey and C. P. Wadsworth : "Continuations : A mathematical semantics for handling full jumps", Technical Monography PRG-11, Oxford University, England, 1974.