Implementation Strategies for Continuations

William D Clinger
Anne H Hartheimer

Semantic Microsystems, Inc
4470 SW Hall, Suite 340
Beaverton, Oregon 97005

Eric M Ost

Metaphor Corp
Bloomington, Indiana

Abstract

Scheme and Smalltalk continuations may have unlimited extent. This means that a purely stack-based implementation of continuations, as suffices for most languages, is inadequate. Several implementation strategies have been described in the literature. Determining which is best requires knowledge of the kinds of programs that will commonly be run.

Danvy, for example, has conjectured that continuation captures occur in clusters. That is, the same continuation, once captured, is likely to be captured again. As evidence, Danvy cited the use of continuations in a research setting. We report that Danvy's conjecture is somewhat true in the commercial setting of MacScheme+Toolsmith™, which provides tools for developing Macintosh user interfaces in Scheme. These include an interrupt-driven event system and multitasking, both implemented by liberal use of continuations.

We describe several implementation strategies for continuations and compare four of them using benchmarks. We conclude that the most popular strategy may have a slight edge when continuations are not used at all, but that other strategies perform better when continuations are used and Danvy's conjecture holds.

1. Introduction

A *continuation* is the abstract concept represented by the control stack, or dynamic chain of activation records, in a typical programming language implementation. Continuations correspond to *contexts* in Smalltalk-80™. In languages such as Scheme and Smalltalk-80, continuations may become first class objects with unlimited extent (lifetime) [Rees 86] [Goldberg 83]. This means that a purely stack-based implementation of recursive procedure calls, which suffices for most languages, is inadequate.

In Scheme, the `call-with-current-continuation` procedure is the mechanism that allows continuations to outlive their more usual dynamic extent. One possible implementation of this procedure, in terms of lower-level procedures `creg` and `creg-set!`, is shown below. This code assumes that the `creg` procedure returns the contents of the continuation register as a Scheme object with unlimited extent, and that the `creg-set!` procedure takes such an object and stores it back into the continuation register. The operation performed by `creg` is called a *capture*. The operation performed by `creg-set!` is called a *throw*.

```
(define (call-with-current-continuation f)
  (let ((k (creg)))
    (f (lambda (v)
         (creg-set! k)
         v))))
```

The simplest implementation strategy for such continuations is to allocate storage for each continuation frame (activation record) on a heap and to reclaim that storage through garbage collection or reference counting [Goldberg 83]. Several more efficient implementation strategies for continuations with unlimited extent have been described [Deutsch 84] [Suzuki 84] [Bartley 86] [Moss 87] [Danvy 87] [Miranda 87]. Determining which is best requires knowledge of the kinds of programs that will commonly be run.

Danvy, for example, has conjectured that continuation captures occur in clusters. That is, the same continuation, once captured (by a call to `call-with-current-continuation`), is likely to be captured again—either an enclosing continuation will be captured, or some subpart of it will be captured. Danvy's thesis, as we will call it, is at least partially true of most programs written using MacScheme+Toolsmith, a commercial development system based on MacScheme® [Semantic 87]. The reasons have to do with the Macintosh™ user interface and Toolsmith's presentation of it to the Scheme programmer.

The Macintosh user interface is built upon a polling model. Programs are expected to take the form of a centralized dispatcher that busy-waits for an event, which represents a user action such as the depression of a key on the mouse or keyboard. When an event becomes available the dispatcher passes it to an appropriate subroutine for processing. When the subroutine has finished it returns control to the dispatcher, which resumes its busy-wait. This seemingly simple program structure leads either to unresponsive programs, which signal by

means of a wait cursor that they can accept no further instructions from the user until they have completed their current task, or to poorly modularized programs, which remain responsive by means of explicit calls to a polling routine strewn throughout computation-intensive code.

MacScheme+Toolsmith makes it possible to program the Macintosh using an interrupt model instead of the polling model. The compiler takes care of polling implicitly by inserting code at every backward branch and at every procedure call. Interrupt handlers are written in Scheme and may be installed and removed dynamically. The default interrupt handlers take care of the standard events by dispatching them to the appropriate object (such as a window or menu) using a standard message protocol. Preemptive multitasking is provided for such tasks as blinking the insertion point in a text window, or changing the shape of the cursor in response to its position relative to objects on the screen. These features (interrupts and multitasking) make it much easier to program the Macintosh and lead to more modular and more responsive programs.

In MacScheme+Toolsmith, the interrupt system and multitasking are implemented using first class continuations. In fact, every exception captures a continuation. Hence a continuation is captured every time a key is struck or the mouse clicked. In practice, use of continuations for these purposes tends to cause the same continuation to be captured repeatedly. This is because interrupts for occurrences such as keyboard events usually occur in bursts, and task switches occur frequently as well. Since the continuation tends to change little during such short periods of time, Danvy's thesis holds fairly well for most programs written using MacScheme+Toolsmith.

Of course we didn't have to use the full power of continuations in order to implement interrupts or multitasking, but it was certainly convenient to do so. MacScheme+Toolsmith has in fact achieved great simplicity of implementation by using continuations for these things. We were emboldened to do so by our implementation of continuations, which makes them significantly more efficient than in other implementations of Scheme.

2. Danvy's thesis

We investigate whether Danvy's thesis holds for application programs written using MacScheme+Toolsmith by considering programs that make no explicit use of `call-with-current-continuation`. If Danvy's thesis holds for these programs, then it surely holds for programs that use `call-with-current-continuation` explicitly.

Our test programs come from the Gabriel benchmark suite, translated into Scheme [Gabriel 85]. Of these, only the `ctak` benchmark uses explicit continuations.

Multitasking is a standard use of continuations in Scheme [Haynes 84]. Most application programs written using MacScheme+Toolsmith make use of multitasking, because concurrent tasks are the most practical way to perform various chores mandated by the Macintosh user interface. On a Macintosh II, approximately ten task switches occur each second. Each task switch involves two captures of a continuation (one by the timer interrupt and the other by the task scheduler) and one throw (by the task scheduler; the timer interrupt is not really returned from until the interrupted task is later resumed by the task scheduler).

To measure how well Danvy's thesis holds under these circumstances, we wrote a handler for timer interrupts that records both the size in words of the continuation structure being captured and the number of words shared between that structure and the previously captured continuation structure. From this data we calculated the average percentage of the continuation structure that had been captured previously. Figure 1 shows that Danvy's thesis holds fairly well even for programs that don't appear to use continuations at all.

It is a bit odd that each timer interrupt captures a continuation. Since the task scheduler will capture exactly the same continuation, the average fraction of continuation structure that is being recaptured due to multitasking can never be less than half. We therefore adjusted the data to show what would happen if captures were performed only by the task scheduler. We also subtracted 37 words of continuation structure created by the read/eval/print loop and other system code. The adjusted percentages in Figure 1 show that Danvy's thesis might not hold up very well if timer interrupts had been implemented a little differently.

On the other hand, the adjusted percentages show that Danvy's thesis is more true of the larger benchmarks than of the smaller ones. If this trend continues to even larger programs, as seems plausible, then Danvy's thesis should be true of most realistically sized programs regardless of how multitasking is implemented. We tested this hypothesis by performing similar measurements on the MacScheme compiler. We found that the adjusted percentages were often higher than for any of the Gabriel benchmarks, and that the absolute size of the recaptured continuation structure was significantly greater.

Danvy's thesis may also be true of typical programs in other implementations of Scheme, but we doubt it. Most Scheme programmers seem to be aware that `call-with-current-continuation` is an expensive procedure, so they avoid it even if they are otherwise tempted to use it. Programmers would be more inclined to use continuations if they were implemented more efficiently.

Whether programmers should be encouraged to use first class continuations is beyond the scope of this paper.

3. Implementation strategies

The garbage collection strategy

The simplest strategy for ensuring that continuations have unlimited extent is to allocate them in the heap and rely on garbage collection or reference counting to recover their storage [Holloway 80] [Goldberg 83]. We call this the *gc strategy*. When used with a fast garbage collector, the gc strategy is fast enough to be considered for use with byte code interpreters but is impractical for native code. (We used it in MacScheme until we added the native code compiler.) Since most continuations are short-lived, we would like to recover their storage using some more efficient mechanism than garbage collection.

Benchmark	Average Size of Continuation (32-bit words)	Per Cent Recaptured	Adjusted Percentage
tak	189	69	23
ctak	166	76	37
takl	195	75	37
boyer	381	81	58
browse	105	76	25
destructive	69	78	3
traverse-init	68	82	23
traverse	205	90	74
deriv	96	70	2
dderiv	96	70	2
div-iter	61	80	0
div-rec	476	54	0
fft	80	91	65
puzzle	159	88	68
triangle	207	81	54
fprint	103	82	42
fread	202	83	58
tprint	106	81	41
compiler	455	92	83

Figure 1. Multitasking and Danvy's thesis.

The spaghetti strategy

The spaghetti stack used in Interlisp is a variation of the gc strategy [Bobrow 73]. The spaghetti stack is in effect a separate heap in which storage is reclaimed by reference counting rather than garbage collection. Though complex, the spaghetti stack is more efficient than a straightforward gc strategy because its storage management is optimized to support procedure call, return, and a host of related operations. In the normal case, when all frames have dynamic extent, the spaghetti stack behaves as a conventional stack.

The strategies discussed in the rest of this section can be regarded as simplifications of the spaghetti strategy. They perform worse than the spaghetti strategy in exceptional cases, but achieve better average performance because they simplify the normal case.

The heap strategy

The lifetime of a continuation frame created for a procedure call normally ends when the called procedure returns. The only exception is for continuation frames that have been captured. This suggests the *heap strategy*, in which a one-bit reference count in each frame indicates whether the frame has been captured. Continuation frames are allocated in a garbage-collected heap as in the gc strategy, but a free list of uncaptured frames is also used. When a frame is needed by a procedure call, it is taken from the free list unless the free list is empty. If the free list is empty, then the frame is allocated from the heap. When a frame is returned through, it is linked onto the free list if its reference count indicates that it has not been captured. Otherwise it is left for the garbage collector to reclaim. Capturing a continuation involves setting the reference count of every frame in the continuation to indicate that it has been captured. Throwing to a continuation involves nothing more than storing the continuation in the continuation register.

The heap strategy is most practical if all continuation frames are the same size. [Danvy 87] describes an incremental variation of the heap strategy that avoids the need to mark all continuation frames when a continuation is captured, at the cost of a few extra instructions for each call and return. A very clever and efficient variation of the heap strategy is presented in [Moss 87] and will be discussed in a later section.

The stack strategy

Only continuations that have been captured can outlive their stack life. This leads to the *stack strategy*, in which the active continuation is represented as a contiguous stack in an area of storage we call the *stack cache*. Non-tail-recursive calls push continuation frames onto this stack cache, and returns pop frames from the stack cache, just as in an ordinary stack-based language. When a continuation is captured, however, a copy of the entire stack cache is made and stored in the heap. When a continuation is thrown to, the stack cache is cleared and the continuation is copied back into the stack cache.

The stack strategy can use standard calling sequences, making procedure calls and returns just as fast as for languages that restrict continuations to have dynamic extent. This apparently means that programs that don't use `call-with-current-continuation` don't have to pay for it. There is a small hidden cost because the existence of continuations with unlimited extent precludes some compiler optimizations, but this is hard to quantify.

Variations on the stack strategy are used by most implementations of Scheme and Smalltalk-80 [Deutsch 84] [Suzuki 84] [Bartley 86] [Samples 86] [Ungar 87]. PC Scheme, for example, uses a chunked stack strategy: By maintaining a small bound on the size of the stack cache, and copying portions of the stack cache into the heap or back again as the stack cache overflows and underflows, PC Scheme reduces the worst-case

Strategy	Allocate and Link			Deallocate and Unlink		
gc	sub.l	#size,avail		move.l	link(cont),cont	
	cmp.l	limit,avail	*			
	blt	overflow	*			
	move.l	cont,link(avail)				
	move.l	avail,cont				
heap	move.l	link(free),temp		move.l	link(cont),temp	
	beq	allocateMoreFrames	*	bmi	doNotFree	*
	move.l	cont,link(free)		move.l	free,link(cont)	
	move.l	free,cont		move.l	cont,free	
	move.l	temp,free		move.l	temp,cont	
Moss's variation	move.l	next(cont),cont		move.l	prev(cont),cont	
	bmi	overflow	*	bmi	doNotFree	*
stack	sub.l	#size,cont		add.l	#size,cont	
	cmp.l	limit,cont	*			
	blt	overflow	*			
stack/heap	sub.l	#size,stack		cmp.l	stack,stackBottom	*
	cmp.l	limit,stack	*	beq	heapCase	*
	blt	overflow	*	add.l	#size,stack	

Figure 2. Assembly code for several strategies.

latency of captures and throws [Bartley 86]. In PC Scheme the stack cache really is a cache, since much of the active continuation may reside in the heap.

Since the stack strategy can create multiple copies of a continuation frame when the frame is captured more than once, continuation frames should not be side effected once they have been allocated and initialized. This means, for example, that storage for variables that appear on the left hand side of an assignment should not be allocated in a continuation frame, because an assignment to the variable that alters one frame cannot affect other copies of it unless the implementation is doing something complicated. This is why many Scheme compilers allocate all such variables in heap storage, with the happy result that knowledgeable Scheme programmers try to avoid assignments because they are so inefficient.

SOAR avoids this problem by using a variation of the chunked stack strategy in which the copying of a captured frame is deferred until it is returned through (or a throw is performed). Then and only then is it copied into the heap. Pointers to the frame are updated using information maintained by the generation scavenging garbage collector [Ungar 87].

The stack/heap strategy

The *stack/heap strategy* is similar to the stack strategy. All continuation frames are allocated in the stack cache. When a continuation is captured, however, the contents of the stack cache are moved into the heap and the stack cache is cleared. Likewise when a continuation is thrown to, the new active continuation is left in the heap and the stack cache is cleared. This means that each procedure return must test to see whether the continuation frame being returned through is in the stack cache or the heap. Naturally, the frame should be popped off the stack cache only if it is in the stack cache.

The main advantage of the stack/heap strategy is that it makes throwing very fast, and recapturing a previously captured continuation is very fast also. It therefore works well when Danvy's thesis holds.

The stack/heap strategy is used by only a few systems: Tektronix Smalltalk [Wirfs-Brock 88], BrouHaHa [Miranda 87], and MacScheme. BrouHaHa is notable because continuation frames in the heap are represented quite differently from frames in the stack cache. BrouHaHa uses two distinct interpreters, depending on whether the topmost continuation frame is in the heap or in the stack cache.

With the stack/heap strategy, there is never more than one copy of a continuation frame. This means it is all right to allocate storage for assigned variables within a frame.

A disadvantage of the stack/heap strategy is that some compilers allocate only one continuation frame to be used by all the procedure calls within a procedure. This optimization can make a large difference in a system's performance on the infamous `tak` benchmark, for example. The stack/heap strategy makes this optimization inconvenient because the frame may be reused only if it is in the stack cache.

The incremental stack/heap strategy

The *incremental stack/heap strategy* is a minor variation on the stack/heap strategy: When returning through a continuation frame that isn't in the stack cache, a trap occurs and copies the frame into the stack cache. This makes it easier for compiled code to reuse the frame for other calls.

Benchmark	gc strategy	stack strategy	stack/heap strategy	incremental stack/heap
tak	2.56	2.28	2.41	2.42
ctak	24.0	53.3	24.7	25.3
loop1	14.3	14.3	14.3	14.3
loop2	83.9	173.5	83.8	98.0
takl	11.8	11.0	11.3	11.3
takl (tasking)	12.8	12.3	12.4	12.4
boyer	58.3	49.0	50.8	50.6
boyer (tasking)	61.3	54.3	54.6	54.8
deriv	14.4	12.6	13.5	13.6
deriv (tasking)	15.5	13.8	14.6	14.7
puzzle	33.2	33.6	32.9	33.0
puzzle (tasking)	34.6	35.2	34.3	34.4

Figure 3. Four strategies compared on a Macintosh II. Times are in seconds.

4. Code sequences

Figure 2 lists typical Motorola 68000 assembly code to allocate a new frame and link it to its predecessor, and to deallocate and unlink a frame. Only the normal case is shown. Instructions marked with an asterisk can be eliminated in many systems by relying on address faults or other trickery.

For example, the extra instructions incurred by the stack/heap strategy on each procedure return can be eliminated by maintaining a permanent continuation frame at the bottom of the stack cache. This frame's return address points to system code that immediately returns through the topmost continuation frame in the heap.

The code sequences for the incremental stack/heap strategy are the same as for the stack/heap strategy since they differ only in the exceptional case of returning through a frame that is not in the stack cache.

In Moss's strategy, a clever variation of the heap strategy, the continuation register serves as both the continuation and the free list pointer [Moss 87]. It points into the middle of a doubly linked list. The free list is linked in one direction through the `next` fields, while the continuation frames are linked in the opposite direction through the `prev` fields. The ends of the list are indicated by links that point to themselves but have the high order bit set. The `prev` field for a frame that has been captured also points to itself and has its high bit set; the real dynamic link for such a frame is stored elsewhere in the frame.

5. An experiment

We implemented the gc, stack, stack/heap, and incremental stack/heap strategies by modifying MacScheme+Toolsmith version 1.5. Non-tail-recursive procedure calls are fairly slow in MacScheme because its calling conventions were designed for a byte code interpreter, where it is faster to have a byte code do potentially unnecessary work than it is to decode multiple byte codes. Since the operations associated with allocating and linking a continuation frame are so complex in MacScheme, they are performed by an out-of-line routine in native code as well as in byte code; likewise for deallocating and unlinking a continuation frame. This made it possible for us to implement the four strategies without making any change to the compiler, so all four strategies were tested using identical native code.

We were unable to test the heap strategy because MacScheme uses continuation frames of various sizes.

Our analysis begins with two outrageous benchmarks. The `ctak` benchmark, as modified for Scheme, captures a continuation on every procedure call and throws on every return. Since `call-with-current-continuation` is not open-coded by MacScheme, this benchmark involves three times as many procedure calls as the `tak` benchmark and creates 63609 closures. The `loop2` benchmark is a countdown loop that throws to a previously captured continuation every time through the loop. Source code for these benchmarks, together with their less exotic analogues `tak` and `loop1`, are shown in an appendix. The remaining benchmarks are straightforward translations of Common Lisp code found in [Gabriel 85]. Benchmarks were run on a Macintosh II with 5 megabytes of RAM and on a Macintosh Plus with 1 megabyte, using generic arithmetic and fully safe code. In particular, stack overflow was detected by software using equivalents of the starred instructions in Figure 2. The timings reported in Figure 3 are in seconds and represent the median of three runs for each strategy.

The stack strategy is easily the worst of the tested strategies on the continuation-intensive benchmarks `ctak` and `loop2`. The other three strategies are about twice as fast. The gc strategy has a slight edge on the `ctak` benchmark because it never has to copy any frames. The incremental stack/heap strategy is a little slower than the stack/heap strategy on the `loop2` benchmark because it has to copy a frame into the stack cache each time through the loop.

We also ran these two benchmarks on comparable hardware using PC Scheme and T3. These implementations use the stack strategy, but we found that they do not perform as well on continuation-intensive benchmarks as our experimental implementation of the stack strategy.

Benchmark	gc strategy	stack strategy	stack/heap strategy	incremental stack/heap
tak	58.4	10.4	10.5	10.5
tak (tasking)	60.5	13.0	11.4	11.4
takl	172.5	49.8	50.0	50.0
takl (tasking)	172.1	64.2	57.3	57.4
deriv	299.5	113.5	112.3	112.4
deriv (tasking)	280.0	117.0	112.7	112.7

Figure 4. Four strategies compared on a Macintosh Plus. Times are in seconds.

While the stack strategy performs poorly on continuation-intensive programs, it has the best performance when first class continuations are not used. The `tak` benchmark shows that the stack strategy performs well on procedure calls, while the gc strategy is the worst of the four. The stack/heap and incremental stack/heap strategies perform identically, suffering from one extra instruction per return compared to the stack strategy. This extra instruction isn't very significant for MacScheme but would be quite significant in a very high performance system like T3, which runs the `tak` benchmark in .22 seconds on a Sun 3/160 using fixnum arithmetic [Kranz 88]. High performance systems would have to eliminate this instruction using some sort of trick as described in the previous section.

We were surprised that one instruction appears to be so costly. The only difference between the code executed by the stack strategy on this benchmark and the code executed by the stack/heap strategy is that the stack strategy uses a

```
dbra    timer,usualCase
```

to decrement a software timer while the stack/heap strategy uses

```
cmp.l   bottom(globals),stack
dbeq    timer,usualCase
```

to test for an empty stack cache and to decrement the timer. Since these sequences are executed 47706 times by the `tak` benchmark, the second sequence appears to be about 2.5 microseconds slower than the first. Clearly we have chosen an unfortunate instruction ordering that causes pipelining or caching or alignment problems for the 68020. Rather than experiment by reorganizing the code, we ran `tak` and a few other benchmarks on the 68000-based Macintosh Plus, using a much smaller heap size. The results are shown in Figure 4.

Figures 3 and 4 show that the stack strategy's slight edge over the stack/heap strategies disappears when tasking is enabled in MacScheme+Toolsmith. Under these conditions the stack/heap strategies deliver essentially the same performance as the stack strategy on programs that don't use `call-with-current-continuation`, and should perform better on programs that do.

In Section 2 we argued that Danvy's thesis is more likely to hold for larger programs, and that the average size of the continuation structure should increase for larger programs as well. This means that the performance of the stack/heap strategies relative to the pure stack strategy should improve as the program size is increased. The chunked stack strategy used in PC Scheme appears to be a nice compromise, since it scales better than the pure stack strategy.

The large heap size used for Figure 3 makes the gc strategy look better than it is, but Figure 4 shows it in a harsher light. Figure 4 also sheds a harsher light on the stack strategy, which allocates more storage than the stack/heap strategies when Danvy's thesis is true. Garbage collection is responsible for most of the variation shown by Figure 4.

What might be wrong with this experiment? The main problem is that MacScheme's relatively slow procedure call dilutes the differences between the strategies. Since MacScheme was designed to use the stack/heap strategy, the calling conventions might also be biased against the stack strategy. We, however, are convinced that the calling conventions are biased against speed in general, and hurt the stack/heap strategy as much as the stack strategy.

6. Conclusion

We recommend the incremental stack/heap strategy. If implemented carefully, it should perform just as well as the stack strategy when first class continuations are not used, and should perform much better when continuations are exploited heavily and Danvy's thesis holds.

Danvy's thesis holds well enough for typical programs written in MacScheme+Toolsmith that even a simple-minded implementation of the stack/heap strategies performs as well or better than the stack strategy. We can't say that the stack/heap strategies will perform better than the stack strategy for typical uses of `call-with-current-continuation` in other systems because we don't have any data on typical uses. What we can say is that the stack/heap strategies take some of the fear out of using `call-with-current-continuation`.

On the other hand, our data confirm that the stack strategy is adequate for systems that make only modest use of continuations. A chunked stack strategy like that used in PC Scheme should be an effective compromise between the pure stack strategy and the incremental stack/heap strategy.

We are intrigued by Moss's variation on the heap strategy, which we were unable to test. It is very simple, though it uses an extra memory reference on each call and return.

What is the overall performance cost of first class continuations in a language like Scheme? With a sufficiently clever implementation, there is no apparent cost to programs that don't use `call-with-current-continuation`. A few compiler optimizations that depend on the movement of code with potential side effects across an unknown procedure call are thwarted by the existence of first class continuations, but we would be very surprised if these optimizations could improve the overall performance of a Scheme system by more than one or two per cent; the T3 compiler certainly performs rather well without them [Kranz 88]. The real performance cost of first class continuations is the time and money required to implement them. The days or weeks spent on continuations can't be spent tuning the rest of a system.

7. Acknowledgements

Under the influence of stack management techniques invented for Algol 60, early work in this area often used a single storage management technique for both environment structure and continuation structure. Sometime during the 1982–1983 academic year Jonathan Rees pointed out to one of us (Clinger) that we could forget about environment structure by assuming that all variables are in registers or in heap-allocated storage. This insight leads quickly to the stack and stack/heap strategies.

The comments and experience of Norman Adams, Richard Kelsey, Jonathan Rees, Allen Wirfs-Brock, and an anonymous member of the program committee were very helpful to us as we prepared this paper.

References

[Bartley 86] 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, pages 86–93.

[Caudill 86] Patrick J Caudill and Allen Wirfs-Brock, "A Third Generation Smalltalk-80 Implementation", *Conference Proceedings of OOPSLA '86, SIGPLAN Notices* **21**, 11, November 1986, pages 119–130.

[Danvy 87] Olivier Danvy, "Memory Allocation and Higher-Order Functions", *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June 1987, pages 241–252.

[Deutsch 84] L Peter Deutsch and Allan M Schiffman, "Efficient Implementation of the Smalltalk-80 System", *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, January 1984, pages 297–302.

[Gabriel 85] Richard P Gabriel, *Performance and Evaluation of Lisp Systems*, The MIT Press, 1985.

[Goldberg 83] Adele Goldberg and David Robson, *Smalltalk-80: the Language and its Implementation*, Addison-Wesley, 1983.

[Haynes 84] Christopher T Haynes and Daniel P Friedman, "Engines Build Process Abstractions", *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984, pages 18–24.

[Holloway 80] Jack Holloway, Guy L Steele, Gerald Jay Sussman, and Alan Bell, "The SCHEME-79 Chip", MIT AI Laboratory, AI Memo 559, January 1980.

[Kranz 86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams, "Orbit: An Optimizing Compiler for Scheme", *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, July 1986, pages 219–233.

[Kranz 88] David Andrew Kranz, *ORBIT: An Optimizing Compiler for Scheme*, PhD thesis, Yale University, May 1988.

[Miranda 87] Eliot Miranda, "BrouHaHa—A Portable Smalltalk Interpreter", *Conference Proceedings of OOPSLA '87, SIGPLAN Notices* **22**, 12, December 1987, pages 354–365.

[Moss 87] J Eliot B Moss, "Managing Stack Frames in Smalltalk", *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June 1987, pages 229–240.

[Rees 86] Jonathan Rees and William Clinger [editors], "Revised^3 Report on the Algorithmic Language Scheme", *SIGPLAN Notices* **21**, 12, December 1986, pages 37–79.

[Samples 86] A Dain Samples, David Ungar, and Paul Hilfinger, "SOAR: Smalltalk without Bytecodes", *Conference Proceedings of OOPSLA '86, SIGPLAN Notices* **21**, 11, November 1986, pages 107–118.

[Semantic 87] Semantic Microsystems, *MacScheme+Toolsmith*, August 1987.

[Suzuki 84] Norihisa Suzuki and Minoru Terada, "Creating Efficient Systems for Object-Oriented Languages", *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, January 1984, pages 290–296.

[Ungar 87] David M Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, The MIT Press, 1987.

[Wirfs-Brock 88] Allen Wirfs-Brock, personal communication, April 1988. Tektronix Smalltalk is described in [Caudill 86], which was not detailed enough for us to realize that Tektronix Smalltalk uses the stack/heap strategy rather than the stack strategy.

Appendix: Source code

```
;;; TAK -- A vanilla version of the TAKeuchi function

(define (tak x y z)
  (if (not (< y x))
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y))))

(run-benchmark "TAK" (lambda () (tak 18 12 6)))

;;; CTAK -- A version of the TAK procedure that uses
;;; continuations.

(define (ctak x y z)
  (call-with-current-continuation
   (lambda (k)
     (ctak-aux k x y z))))

(define (ctak-aux k x y z)
  (cond ((not (< y x))  ;xy
         (k z))
        (else (call-with-current-continuation
               (lambda (k)
                 (ctak-aux
                  k
                  (call-with-current-continuation
                   (lambda (k)
                     (ctak-aux k
                               (- x 1)
                               y
                               z)))
                  (call-with-current-continuation
                   (lambda (k)
                     (ctak-aux k
                               (- y 1)
                               z
                               x)))
                  (call-with-current-continuation
                   (lambda (k)
                     (ctak-aux k
                               (- z 1)
                               x
                               y)))))))))

(run-benchmark "CTAK" (lambda () (ctak 18 12 6)))
```

```
;;; LOOP1 -- A perverse way to write a loop.

(define (loop1 n)
  (let ((n n)
        (k 0))
    (define (loop ignored)
      (if (zero? n)
          'done
          (begin (set! n (- n 1))
                 (loop #t))))
    (loop #t)))

(run-benchmark "Loop1" (lambda () (loop1 1000000)))

;;; LOOP2 -- An extremely perverse way to write a loop.

(define (loop2 n)
  (let ((n n)
        (k 0))
    (define (loop ignored)
      (call-with-current-continuation
       (lambda (cont)
         (set! k cont)))
      (if (zero? n)
          'done
          (begin (set! n (- n 1))
                 (k #t))))
    (loop #t)))

(run-benchmark "Loop2" (lambda () (loop2 1000000)))
```