Tail-Recursive Stack Disciplines for an Interpreter

Richard A. Kelsey
NEC Research Institute
kelsey@research.nj.nec.com

8 March 1993

Abstract

Many languages, including Scheme, ML, and Haskell, require that their implementations support tail-recursive calls to arbitrary depth. This requirement means that a traditional stack discipline cannot be used for these languages. This paper examines several different methods of implementing proper tail recursion in a stack-based interpeter, including passing arguments in a heap, copying arguments to tail-recursive calls, and garbage collecting the stack. Benchmark timings and other run-time statistics are used to compare the different methods. The results show that using a stack is a good idea, and that the overhead of the interpreter largely overshadows the differences in performance of the various stack disciplines.

This is a slightly enhanced version of Technical Report NU-CCS-93-03, College of Computer Science, Northeastern University, 1992.

1 The Problem

A programming language implementation is properly tail recursive if unbounded iterative computations that are expressed recursively can be executed in constant space. Implementations of languages such as Scheme [Rees 86], ML [Milner 88], and Haskell [Hudak 90] are required to be tail recursive.

These languages also require that some lexical environments have indefinite extent. That is, arguments passed to a procedure may need to be preserved until long after the procedure has returned. Similarly, in Scheme, continuations also may have indefinite extent. A continuation is all of the information that is saved across a procedure call and used to restart the procedure's caller once the procedure returns. In Scheme procedure invocations may return more than once, so continuations may need to be saved for later use.

Why do I, normally a compiler hacker, want to use interpreters to implement such languages? There are several advantages of an interpreter over a native-code compiler. Interpreters are much more portable and usually provide much better debugging information for the user. The interpreter I used for the experiments reported here is a C program that can be run without modification on a large number of different types of machines. An interpreter usually does a minimal amount of processing of a source program before execution. This makes it easy to relate an interpreter's internal state to the source program in a way that is comprehensible to the user. Interpreters should be simple, fast, and provide good debugging information for the user. I will judge the stack disciplines on how well they support these goals.

The best stack discipline might be not to have a stack at all. It is advocated in [Appel 87] that with a very efficient garbage collector and a good compiler, heaps may be more efficient than stacks. With an interpreter this appears not to be the case. Making efficient use of a heap for procedure calls requires more sophisticated compilation than is usual for interpreters. When not using a stack, and with an efficient garbage collector, the benchmarks used in this paper ran 13% more slowly than the slowest of the stack implementations, and 27% more slowly than the fastest.

2 Why a traditional stack won't work

Stacks are the standard way to implement continuations and lexical environments [Aho 86]. To make a procedure call, the arguments are pushed on the stack followed by the saved program counter, environment pointer and whatever else will be needed once the procedure returns. In the called procedure, the pushed arguments become the new lexical environment and the saved state of the caller is the current continuation. The environment and continuation together are often refered to as an 'activation record'. When the procedure returns, the environment and the continuation are removed from the stack. Figure 1 shows how such a stack might look just as control transfers to the body of procedure `G` and again at the beginning of `F` in the following Scheme example:

```
(DEFINE FOO
  (LAMBDA (A D)
    (F A B (G C D))))
```

The interpreter is assumed to have three registers: a program counter `*PC*`; the current environment `*ENV*`; and the current continuation `*CONT*`. All three of these are saved when a new continuation is created. The environment includes a pointer to the called procedure's saved lexical environment to be used for reaching variables bound in outer lexical contours.

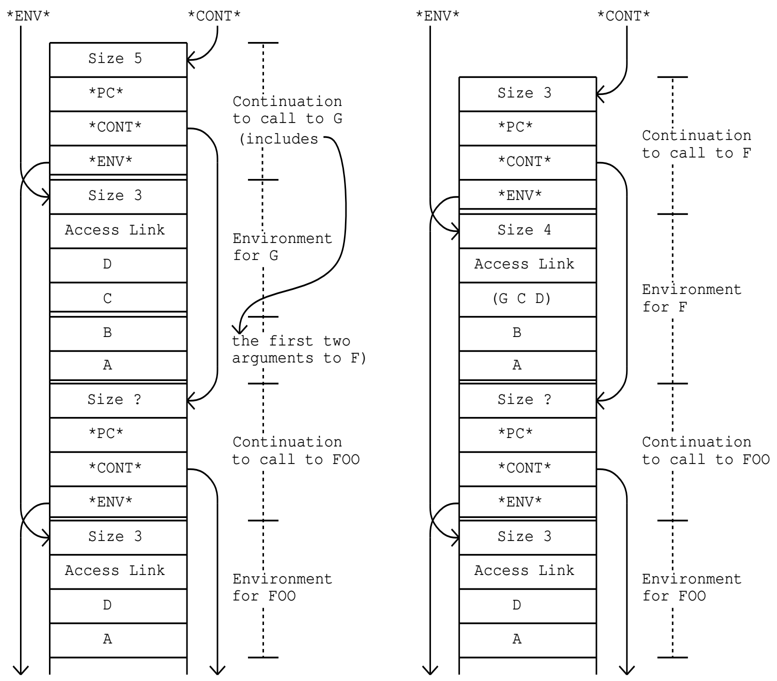

Figure 1: Examples of traditional stack usage

The standard stack discipline is not an obstacle to implementing long-lived environments or continuations. In most cases, the environment and continuation passed to a procedure are removed from the stack when the procedure returns. If they need to be preserved they may be copied to a heap [Clinger 88]. Copying requires that each environment and continuation include an indication of its size. For continuations, this may be either unnecessary, if continuations need not be preserved, or it may be found at a known offset from the saved program counter and not take up any space on the stack.

The difficulty with the standard stack discipline is that it does not implement proper tail recursion. When control transfers to the body of `F` the stack contains the environment and the continuation of the call to `FOO`. But since the call to `F` is tail recursive, these do not contain any necessary information. The values in the environment will never be used again. The continuation will just remove itself and the environment from the stack and return to its saved value of `*CONT*`. After a series of such calls the stack may overflow, even though it contains very little useful data. The traditional stack discipline eventually recovers the stack space used for a procedure call, but in the case of tail-recursive loops, it recovers it too late.

Overflowing the stack in this fashion puts a limit on the number of times a recursively expressed iterative loop can execute. This is not a problem in languages such as C, Pascal, or FORTRAN which include various forms of iterative control constructs. For languages that provide recursion, but not direct iteration, environments and continuations must be implemented in some other fashion.

Dealing with the continuations is easy. Simply do not make a new continuation for a tail-recursive call, and reuse the current continuation instead. In the example, the call to `F` does not require a new continuation and could use the one passed to `FOO`. The real problem is the environments. A tail-recursive call simultaneously creates a new environment and makes the old one obsolete. On a stack, the storage used for the old environment is lost, as it lies beneath the new environment.

3 Methods for implementing proper tail recursion

I will consider three general approaches to implementing proper tail recursion on a stack.

1. Use the heap instead of a stack

One obvious solution, since there has to be a heap for some environments and continuations, is just to put the environments there in the first place. Unused heap storage will eventually be reclaimed, which allows for an unbounded number of tail-recursive calls. [Appel 87] advocates placing all environments, as well as all continuations, in the heap.

2. Overwrite the environment during tail-recursive calls

It is also possible to implement proper tail recursion and still use a stack for the environments. If no new continuation were created for the call to `F` and its arguments overwrote the environment of `FOO`, the call could be made with no unnecessary data on the stack. Any number of such calls could be made without overflowing.

There are two possibilities here. The current environment could be incrementally overwritten as the new arguments are produced, as in [Kranz 88, Hanson 90], or the arguments could be stored in temporary locations and then copied on top of the current environment once they are all available, also in [Hanson 90, McDermott 80]. The first is usually more efficient, as a clever implementation can get away with moving fewer values, and the second is clearly simpler, as data dependencies between the values in the environment and the arguments can be ignored. In the worst case, where every element of the environment is needed to compute each argument, the first method degenerates to the second.

There is a difficulty with incrementally overwriting the current environment that makes it unsuitable for an interpreter. If an error occurs during the evaluation of an argument, the calling procedure's environment will be partially overwritten by the arguments to the called procedure. This makes

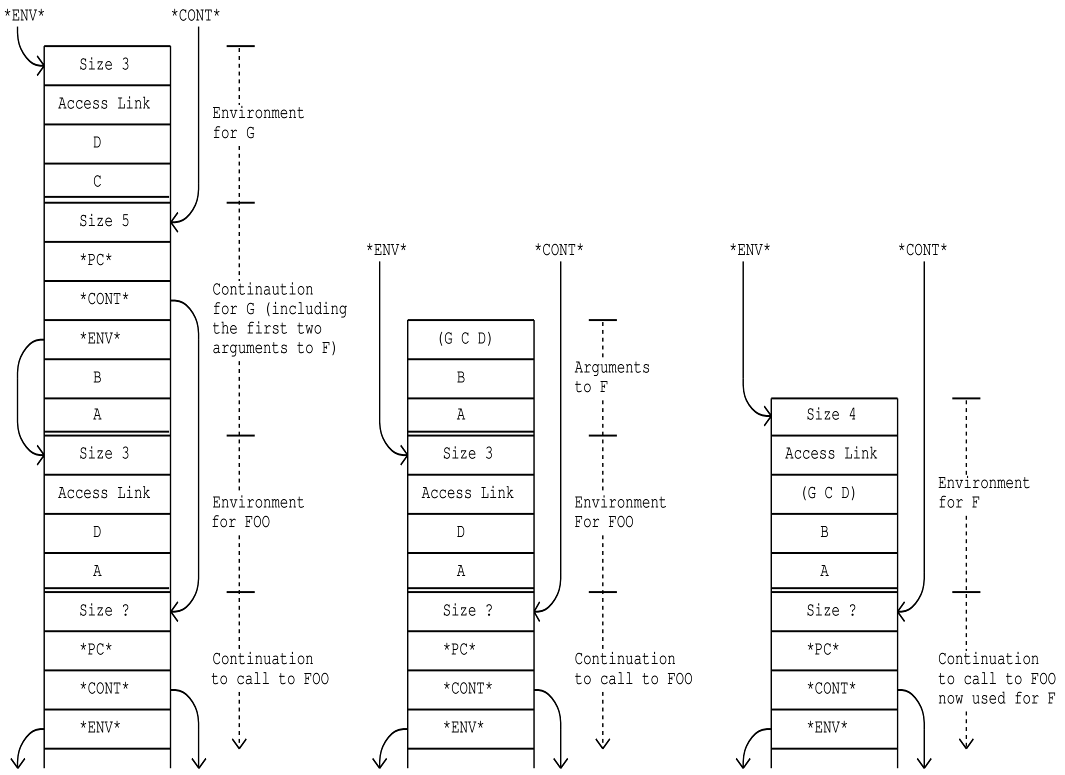

Figure 2: The stack with environments and continuations switched: just before jumping to G; after G returns; and after overwriting the environment of FOO

providing coherent debugging information to the user very difficult. The goals of simplicity and maximal debugging information become mutually exclusive. Delaying the overwriting until all of the arguments have been produced avoids this problem, so from here on I will only consider that alternative.

To avoid complications, the stack usage would have to be changed so that continuations to calls were pushed before the arguments, rather than afterwards. Figure 2 shows how such a stack would appear at various times during the execution of the example. A similar calling sequence is presented in [Steele 77]. Otherwise a procedure would sometimes be called with its arguments on the top of the stack, while at other times there would be a continuation above the arguments.

3. Garbage collect the stack

Another possibility is to wait until the stack is full and then reclaim any unused space. The stack can be treated as a constrained heap, containing only environments and continuations. With this method many environments can be reclaimed without cost when procedures return. The additional cost of garbage collection matters only in the relatively rare instance of the stack actually filling up.

Garbage collecting the stack is simple as pointers between continuations and environments follow a very regular pattern. There are at most three pointers into the stack from the outside: the stack pointer, the control link, and the current environment. In a highly optimized implementation there may only be the stack pointer, with the other two pointers being implicit. Continuations contain a control link which points into the stack, and a saved environment pointer that may or may not point into the stack. Again, in a complex implementation the control link may be implicit. The access link of environments on the stack may also point into the stack. There are no other pointers into the stack. Control links always point to continuations, and saved environment pointers and access links always point to environments. Thus the stack garbage collector never needs to check the type of an object.

There are at least three strategies for a stack garbage collector. The stack could be compacted in place, it could be copied to another stack, or it could be copied into a heap.

The first possibility, compacting the stack in place, can be implemented using a pointer reversal algorithm as long as the compiler maintain certain invariants. The requirement is that pointers to environments on the stack must point to the next highest used environment; they cannot skip over any environment that is still in use. This is not difficult for the compiler to do – indeed generating correct code that violates this requires some program analysis – but it does eliminate certain potential optimizations.

The second method, copying the live data into a second stack, does not restrict the compiler as the additional space allows the use of forwarding pointers. It could result in thrashing if a program pushed a large quantity of live data and then repeatedly caused the stack to overflow. A system similar to that described in [Hieb 90] for minimizing the cost of saving continuations could be used to reduce or avoid copying data multiple times.

The third possibility, copying the live data into a heap, reduces the possibility of thrashing. It is also simpler as it avoids the necessity of preserving the relative positions of the environments and continuations. There is the additional overhead of copying continuations back onto the stack as they are needed (the environments might as well remain in the heap). Checking for stack underflow during procedure returns can be avoided by placing a special continuation at the base of the stack which copies the next continuation from the heap onto the stack. Locality considerations might keep the cost of copying fairly low, as argued in [Clinger 88]. This is certainly the most attractive strategy for languages that already require a heap.

4 Saving Environments

All of the languages which were given as examples of properly tail-recursive languages are also lexically scoped and allow procedures to escape upwards. This means that a local environment may be needed long after the return of the call that created it. When it is determined that an environment may outlive the call that created it, the environment can be preserved by a sort of sub-garbage-collection, which migrates the environment (and its superior environments if they are also on the stack) from the stack to the heap. The decision to save an environment on the heap may be made at compile time or by a check at run time. The same restrictions on pointers into the stack that make the full garbage collection simple are useful here as well. The cost of migrating an environment to the heap is linear in the size of the environment and the number of continuations that reference it; the size of the rest of the stack does not matter.

Compile-time analysis and run-time checks can both be used to reduce the number of environments that are migrated to the heap, as reported in [McDermott 80, Kranz 88, Hanson 90] and elsewhere. The ability to garbage collect the stack increases the number of environments that can be allocated on the stack, even for a compiler that normally overwrites the current lexical environment when making tail-recursive calls. Environments that would not be allowed on the stack according to a strict interpretation of proper tail recursion can be stack-allocated and migrated in the unlikely event of a stack overflow. Using a compiler that allows destructive reuse of environments will greatly decrease the frequency of the already rare stack overflows.

5 Analysis

The three possibilities are: put all environments in the heap, move the arguments to tail-recursive calls, or garbage-collect the stack. What are the relative costs? For brevity, I will call the three disciplines HEAP-ENVS, MOVE-ARGS, and STACK-GC. A fourth option, not to use a stack at all, will be refered to as NO-STACK.

Cost of making continuations

Continuations are identical in the three stack disciplines, with one minor exception. In HEAP-ENVS the stack contains only continuations, since the environments are all in the heap. With no environments to interfere, the current continuation will always be found on the top of the stack. The `*CONT*` register can be eliminated, and no longer needs to be saved in continuations. This saves some work when continuations are created, and again when procedures return.

Continuations have an additional cost for NO-STACK because they require copying values into and out of the heap. Avoiding this copying would require signifigantly more complex preprocessing of the user's program.

Cost of procedure calls

Here I need to distinguish between two possible implementations of HEAP-ENVS. The sequence for creating a new environment for a call may be either (method A)

1. Create a new heap environment
2. Put the arguments in the environment as they are created

or (method B)

1. Push all the arguments on the stack
2. Create a new heap environment
3. Copy the arguments from the stack to the environment

Method A does not require copying each argument, but it does require a new register to hold the new environment while the arguments are being evaluated. This register must be saved in continuations. In a sense it replaces the `*CONT*` register. The other methods need the `*CONT*` register because an environment may be on the top of the stack, while Method A requires a `*NEW-ENV*` register because the new environment cannot be on top of the stack.

Method A does not work for Scheme. In Scheme, use of `call-with-current-continuation` may cause procedure calls to return more than once. If the environment were created before the arguments were generated, and one of the arguments was produced by a call that returned twice, there would be no place to put the second version of the argument. It cannot replace the first value in the environment as both values may be needed by the program. With Method B, the code that created the environment for the call would be executed twice, creating two different environments.

NO-STACK and HEAP-ENVS create all environments in the heap, with method B of HEAP-ENVS also requiring that every argument be moved once. MOVE-ARGS and STACK-GC create all environments on the stack, with MOVE-ARGS requiring that every argument to a tail recursive call be moved once. For MOVE-ARGS and STACK-GC, when an environment needs to be preserved for future use, that environment must be copied into the heap. In NO-STACK and HEAP-ENVS this is unnecessary.

Garbage collection costs

HEAP-ENVS puts environments in the heap that the other methods leave on the stack. NO-STACK puts everything in the heap. These will result in more frequent heap garbage collections.

The STACK-GC method will cause occasional stack garbage collections. These may either be in place, or operate by copying all of the currently live

	NO STACK	HEAP ENV	MOVE ARGS	STACK GC	Description
bubblesort	2.44	2.15	1.99	1.83	bubble sort on 200 integers
quicksort	1.97	1.74	1.70	1.64	quicksort on 1500 integers
tak	2.94	2.38	2.00	1.98	Takeuchi
tak2	1.73	1.44	1.24	1.20	" with inlined call to `not`
cpstak	3.30	2.93	3.10	2.92	Takeuchi in CPS
cpstak2	2.07	1.97	2.31	2.14	" with inlined call to `not`
fib	2.82	2.40	2.15	2.15	Recursive Fibonnaci
towers	1.90	1.57	1.31	1.29	Towers of Hanoi
matrix-mult	4.46	3.91	3.43	3.36	Integer matrix multiply
matrix-mult2	2.08	2.06	2.03	1.99	" with inlined procedures
length	1.05	1.01	0.91	0.87	Length of a list

Figure 3: Benchmark times in seconds running a DEC5000 workstation

data on the stack into the heap. The second method is equivalent to preserving the current continuation for later use. This is exactly what is required to implement the Scheme procedure `call-with-current-continuation`, so for Scheme the stack garbage collector does not involve any additional code.

The bottom line

There is no analytical bottom line. The relative costs of the different disciplines depend on the behavior of the executed code. If the stack never overflows and no environments need to be preserved, then STACK-GC is clearly the fastest, as it would not copy any arguments or have any other overhead. If every environment needed to be preserved, then HEAP-ENVS would be fastest, as it would not have to copy any arguments (for the first method). MOVE-ARGS will likely be slower than STACK-GC, because of the additional argument copying it involves, but it is simpler than STACK-GC, and might be worthwhile if the difference in speed were not much. Unless the code makes frequent use of `call-with-current-continuation`, NO-STACK is likely to be slow due to increased garbage collection overhead and higher costs for continuations.

6 Experimental Results

I have implemented the three stack disciplines, along with a heap-only implementation, in Scheme48, which is based on a byte-code interpreter. A very simple compiler, which performs almost no optimizations, is used to compile Scheme programs into byte-codes. Scheme48 runs these benchmarks at 35% of the speed of Chez Scheme, a commercial Scheme implementation

	Conts		Envs		Closed		Tail	
	#	words	#	words	#	words	#	words
bubblesort	50	201	111	263	1	1	41	41
quicksort	32	130	102	243	22	50	54	61
tak	111	493	127	382	0	0	16	48
tak2	48	239	64	254	0	0	16	48
cpstak	64	254	175	541	48	143	111	302
cpstak2	0	0	111	413	48	143	111	302
fib	93	417	93	185	0	0	0	0
towers	57	254	90	254	0	0	8	25
matrix-mult	85	477	143	509	3	6	58	86
matrix-mult2	1	6	31	90	3	6	29	57
length	0	0	50	150	0	0	50	100

Figure 4: The number of and total number of words used in environments, closed-over environments, and tail-recursive calls while running the benchmarks; all numbers are in 1000s

that uses a native code compiler (this number is somewhat unfair as the two systems were not running in identical modes; it does show that Scheme48 is not unreasonably slow).

The results of running several benchmarks are shown in figures 3 and 4. With the exception of `length` and `cpstak`, the benchmarks were taken from [Kranz 88] and modified to run in Scheme instead of T. `length` is a simple loop in which all of the calls are tail recursive. `cpstak` was written by Will Clinger.

The benchmarks were run with a heap size of one megabyte and a 40,000 byte stack. All methods required heap garbage collections for the two `cpstak` benchmarks. Only NO-STACK and HEAP-ENVS caused heap garbage collections for the other benchmarks. These garbage collections accounted for no more than five percent of the time used, and usually much less. Live data accounted for less than two percent of the total heap space. With more live data or a slower garbage collector the relative runtimes could be very different. For example, in running `bubblesort`, NO-STACK used a total of 2.5 megabytes of heap space, while MOVE-ARGS and STACK-GC used 8K bytes. Only `cpstak` and `cpstak2` caused STACK-GC to overflow the stack.

The statistics require a little explanation. A word is the space required to store one pointer. The word totals for continuations, environments and closed-over environments do not include the space needed to store the sizes of the objects. The word totals for environments do include the space taken up by each environment's pointer to its lexically superior environment. The number of environments is equal to the number of calls with at least one argument. For example, the figures for the `length` benchmark show that 50,000 calls were made, each of which passed two arguments (two arguments plus the pointer to the superior gives 150,000 words). There were 50,000 tail-recursive calls, so every call was tail recursive, and indeed had two arguments, as the tail-recursive calls totalled 100,000 arguments.

Only on `cpstak2` is STACK-GC slower than another method. MOVE-ARGS is almost as fast, running only three percent slower. HEAP-ENVS is eleven percent slower than STACK-GC. Not using a stack at all is twenty-six percent slower than STACK-GC. These differences make sense given the numbers in figure 4. Very few of the environments need to be preserved (the 'Closed' column), and the majority of arguments are to non-tail-recursive calls. HEAP-ENVS was expected to prevail only if many environments had to be preserved, while STACK-GC and MOVE-ARGS are identical except for MOVE-ARGS copying the arguments to tail-recursive calls. With relatively few tail-recursive calls and almost no preserved environments, HEAP-ENVS loses and STACK-GC and MOVE-ARGS tie.

7 Comparison with other work

The methods of implementing tail recursion with a stack that I have described here have been discussed in [Hanson 90, McDermott 80, Appel 87, Kranz 88], usually in the context of a native-code compiler. Interpreters require a different set of priorities. Simple translation and preserving debugging information are much more important than with native-code compilers. The inherent inefficiency of interpretation affects the relative efficiencies of different techniques. For example, much of the concern in [Hanson 90] and [Kranz 88] is with trying to get the maximum number of environments on the stack. In the benchmarks in this paper, the effects of not putting any environments on the stack are not all that great.

In [Clinger 88] different stack implementations are compared in terms of implementing continuations with unlimited extent. In terms of implementing explicit continutions, the no-stack version here corresponds to their 'garbage collection strategy', and the others are all variations on their 'incremental stack/heap strategy'. None of the benchmarks in this paper made use of explicit continuations.

8 Conclusion

The above results are disappointing. The original title of this paper was *Stack Garbage Collection*, and for most code the stack garbage collector does greatly reduce the amount of data that the interpreter copies when performing procedure calls. Unfortunately the benchmarks show that this advantage is overshadowed by the relative inefficiency of the Scheme48's byte-code interpreter.

When compiled for a MIPS microprocessor, which uses a RISC architecture, byte-code dispatch takes five machine instructions, and the 17 byte-codes executed in the loop of the `length` benchmark average 8.3 machine instructions apiece. The argument copying that accounts for the differences is done by tight loops, using only a few instructions for every value copied. One iteration of the tail-recursive loop in `length`, which involves one procedure call with two arguments, requires around 180 instructions. The additional overhead, with HEAP-ENVS or MOVE-ARGS, of moving the two arguments does not matter enormously.

On the positive side, the benchmarks do show the advantage of using some kind of stack. For Scheme, the presence of `call-with-current-continuation` requires code to move the stack into the heap, and that same code can be used for implementing proper tail recursion. For other languages, MOVE-ARGS is probably the best choice: it is nearly as fast, requires little code to implement, and does not put the burden on the heap garbage collector that MOVE-ARGS does.

Acknowledgements

Scheme48 was written by Jonathan Rees and myself. Jonathan Rees and Mitch Wand provided many helpful comments on earlier drafts of this paper.

References

[Aho 86] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Appel 87] Andrew W. Appel. Garbage collection can be faster than stack allocation. In *Information Processing Letters 25*, 1987.

[Clinger 88] William D Clinger, Anne H Hartheimer, and Eric M Ost. Implementation strategies for continuations. In *Proceedings Record of the 1988 ACM Symposium on Lisp and Functional Programming*, ACM, 1988.

[Hanson 90] Chris Hanson. Efficient stack allocation for tail recursive languages. In *Proceedings of the 1990 ACM Symposium on Lisp and Functional Programming*, ACM, 1990.

[Hudak 90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell. Technical Report YALEU/DCS/TR-777, Department of Computer Science, Yale University, 1990.

[Hieb 90] Robert Hieb, R. Kent Dybvig and Carl Bruggeman. Representing Control in the Presence of First-Class Continuations. In *Proceddings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, ACM, 1990.

[Kranz 88] David Andrew Kranz. ORBIT: An optimizing compiler for Scheme. Technical Report YALEU/DCS/TR-632, Department of Computer Science, Yale University, 1988.

[McDermott 80] Drew McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped Lisp. In *Proceedings of the 1980 Lisp Conference*, ACM, 1980

[Milner 88] R. Milner. A proposal for Standard ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, ACM, 1984.

[Rees 86] Jonathan A. Rees and William Clinger, editors. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices* 21(12), pages 37–79, 1986.

[Steele 77] Guy Lewis Steele Jr. Debunking the "expensive procedure call" myth. AI Memo 443, Artificial Intellegence Laboratory, Massachusetts Institute of Technology, 1977.