CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A.[1]

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436, (818) 986-1436 (818) 986-1360 (FAX)

Abstract

Previous Schemes for implementing full tail-recursion when compiling into C have required some form of "trampoline" to pop the stack. We propose solving the tail-recursion problem in the same manner as Standard ML of New Jersey, by allocating all frames in the (garbage-collected) heap. The Scheme program is translated into continuation-passing style, so the target C functions never return. The C stack pointer then becomes the allocation pointer for a Cheney-style copying garbage collection scheme. Our Scheme can use C function calls, C arguments, C variable-arity functions, and separate compilation without requiring complex block-compilation of entire programs.

Introduction

IEEE Scheme [IEEE90] requires that all functions be properly *tail-recursive*, in order that tail-recursive programs not require an unbounded amount of stack space. Several Scheme compilers have targeted the C language [Bartlett89], because C is an efficient systems programming language which is available on nearly every computer, thus ensuring portability of the compiled code. Unfortunately, C compilers are not required to be properly tail-recursive, and only the GNU C compilers [Stallman90] attempt to achieve tail recursion in their implementations.

A popular method for achieving proper tail recursion in a non-tail-recursive C implementation is a *trampoline*.[2] A trampoline is an outer function which iteratively calls an inner function. The inner function returns the address of another function to call, and the outer function then calls this new function. In other words, when an inner function wishes to call another inner function tail-recursively, it returns the address of the function it wants to call back to the trampoline, which then calls the returned function. By returning before calling, the stack is first popped so that it does not grow without bound on a simple iteration. Unfortunately, the cost of such a trampoline function call is 2-3 times slower than a normal C call, and it requires that arguments be passed in global variables [Tarditi92].

Appel's unpublished suggestion for achieving proper tail recursion in C uses a much larger fixed-size stack, continuation-passing style, and also does not put any arguments or data on the C stack. When the stack is about to overflow, the address of the next function to call is `longjmp`'ed (or `return`'ed) to a trampoline. Appel's method avoids making a large number of small trampoline bounces by occasionally jumping off the Empire State Building.

The Proposed Scheme

We propose to compile Scheme by converting it into continuation-passing style (CPS), and then compile the resulting lambda expressions into individual C functions. *Arguments are passed as normal C arguments, and function calls are normal C calls*. Continuation closures and closure environments are passed as extra C arguments. (Of course, calls to closures perform a C call on the code portion of the closure, and pass the environment portion of the closure as an additional argument.) Such a Scheme never executes a C `return`, so the stack will grow and grow.

Since the C "stack" never contracts, we can allocate all of our closures and user data structures on this stack as automatic/dynamic/local data. All closures and user data structures whose sizes are known at compile time are statically allocated in the C "stack" frame; dynamic arrays and other data structures whose size is unknown at compile time can be allocated by C's `alloca` primitive (or equivalent), which also obtains space from the "stack".[3] Since our C "stack" is also the "heap", there is no distinction between stack and heap allocation.

---

[1] Reference to the 1959 song *Charlie on the M.T.A.* [Steiner56] recorded by the Kingston Trio. Lyrics include the refrain

> Oh, will he ever return?
> No, he'll never return,
> and his fate is still unlearned.
> He will ride forever,
> 'neath the streets of Boston,
> he's a man who'll never return.

Charlie couldn't get off the Metropolitan Transit Authority (now the Massachusetts Bay Transit Authority "MBTA") subway (tube) train, because he lacked the money to pay the fare. (By the way, "Charlie" is also the military name for "C".)

[2] Other names for a trampoline are a *dispatcher* or an *operator*, who has to transfer your telephone calls for you.

[3] Maximally portable implementations may shun `alloca`, since it is not required by either ANSI C or Unix.

Since none of our C functions ever returns, the only *live* frame on this "stack" is the top one. However, within many of the dead frames will be found live closures and live user data objects. Eventually, the C "stack" will overflow the space assigned to it, and we must perform garbage collection. Garbage collection (GC) by copying is a relatively straight-forward process. There are a number of static roots, as well as the latest continuation closure, which is passed to the GC as an argument. (Forming an explicit continuation closure for the GC avoids the necessity of scanning C stack frames.) The live objects and live closures are all copied (and thereby condensed) into another area, so that execution can be restarted with a "stack" frame at the beginning of the C "stack" allocation area.

A key point is that since *only live objects are traced*—i.e., garbage (including the C frames, which are all dead) is not traced—*the GC does not have to know the format of a stack frame and can be written in C itself.* A Cheney-style scanner must know the format of all tospace objects, but we copy only objects—never C frames. When the GC is done, it creates a new frame to execute its continuation. The GC is called explicitly from the C code after checking whether the stack pointer has reached its preset limit. Although stack-pointer checking in this way may require a few more instructions than if it were done in assembly language, it is still faster than a trampoline call would be.

```
/* The following macro definition is machine-dependent.  (cpp input is unreadably ugly, isn't it?) */
#ifdef stack_grows_upward
#define stack_check(sp) ((sp) >= limit)
#else
#define stack_check(sp) ((sp) <= limit)
#endif
...
object foo(env,cont,a1,a2,a3) environment env; object cont,a1,a2,a3;
{int xyzzy; void *sp = &xyzzy; /* Where are we on the stack? */
 /* May put other local allocations here. */
 ...
 if (stack_check(sp)) /* Check allocation limit. */
    {closure5_type foo_closure; /* Locally allocate closure object with 5 slots. */
     /* Initialize foo_closure with env,cont,a1,a2,a3 and pointer to foo code. */
     ...
     return GC(&foo_closure);} /* Do GC and then execute foo_closure. */
 /* Rest of foo code follows. */
 ...
}
```

After the GC is done copying, it must reset the C stack pointer to the beginning of the allocation area and call its continuation argument. Since the GC itself has been executing out of its frame in the fromspace, it must cause the stack pointer to be reset to the allocation area in tospace and then call its continuation. One way to relocate the stack pointer is for the GC to call the `alloca` function with an argument which is the difference—*positive or negative!*—between the current stack pointer and the desired stack pointer. Then, the GC can call its continuation.

(An alternative method suggested by Appel follows SML/NJ more closely. After the live data has been copied elsewhere—this is a *minor collection*—the GC can `longjmp` to a trampoline (or simply `return` its argument!) and the trampoline will restart the continuation with the stack pointer allocating at the bottom of the stack again. When the total copied data in the second area exceeds a certain amount, a *major collection* is performed on the second area.)

CONS Should Not CONS Its Arguments

In our Scheme, the compiler does all consing using stack-allocated (dynamic) local storage. Thus, the `revappend` function to reverse one list onto another list looks like the following code. (The C `return` statement is curious, since we don't actually ever return, unless we use Appel's method; `return` simply tells the C compiler that the local variables are all dead (except for the return address!), and therefore it needn't save them "across" the call.)

```
object revappend(cont,old,new) object cont,old,new;
{if (old == NIL)
  {clos_type *c = cont;
   /* Call continuation with new as result. */
   return (c->fn)(c->env,new);}
  {cons_type *o = old; cons_type newer; /* Should check stack here. */
   /* Code for (revappend (cdr old) (cons (car old) new)). */
   newer.tag = cons_tag; newer.car = o->car; newer.cdr = new;
   return revappend(cont,o->cdr,&newer);}}
```


Closures, whose size are always known by the compiler, are explicitly constructed in the local frame in a similar manner. Vectors and arrays whose size is unknown at compile time can be allocated by calling `alloca` (or its equivalent) to extend the current stack frame. (See also section below on `malloc`-allocated objects.)

Variable-arity Functions

Variable-arity Scheme functions can be compiled into variable-arity C functions using either the Unix `varargs` or the ANSI C `stdarg` mechanism. Using CPS, variable-arity multiple "returned" values can be similarly handled.

Iteration

Scheme's only mechanism for expressing iteration is the "tail call", hence Scheme compilers perform a great many optimizations to ensure the efficiency of these "tail recursions". With our Scheme, the definition of an iterative routine is narrowed considerably—only those iterations which do *no* storage allocation may be converted into an iteration, because all storage is being allocated on the stack. In particular, only the lowest-level functions can operate without allocating some storage—e.g., calls to C library routines that restore the "stack" pointer can be made within an iteration. *Local* tail-recursion optimizations [Bartlett89] are thus sufficient to obtain efficient iteration. We thus achieve the *spirit*, if not the *letter*, of ANSI Scheme's tail-recursion law.

Scheme Compiler Optimizations

Scheme compilers perform a number of optimizations to reduce the cost of closure creation and the cost of function calling—e.g., they take advantage of "known" functions when compiling function calls.[4] "Lambda lifting" [Peyton-Jones87] can be used to replace closure creation by additional argument passing; this optimization is very valuable when arguments can be kept in registers. Various kinds of type inference can be used optimize the representation of values and avoid the need for run-time type checks. These optimizations continue to be valuable in our stack/heap Scheme—i.e., they are orthogonal to the policies of memory management.

Using `malloc`

If `malloc`-allocated storage is distinguishable by address range from the stack/heap storage, then `malloc` may be used to allocate (non-relocatable) objects.[5] These objects must be enumerable, so that the GC can first trace these objects (they must have proper tags), and then sweep and explicitly free those objects which have become garbage.

Separate Compilation

In order to obtain proper tail-recursion, existing Scheme-to-C and ML-to-C compilers do large amounts of interprocedural optimizations (including block compilations) which interfere with separate compilation, create large C functions, and cause long C compilations. In our Scheme, every C function can in principle be a separate file.

Calling Normal C Functions

Calling normal C functions which return is trivial, except that you must assure enough "stack" space in advance so that the C "stack" does not overflow during the execution of the C functions. These C functions cannot store pointers to GC-able objects, except in "root" locations, nor can they "call back" to non-returnable Scheme functions.

Hardware Architecture Issues

Appel and Dybvig warn that our pushy Scheme may break "register windows" in, e.g., the SPARC architecture.

Conclusions

The lack of tail recursion in C may be an advantage for implementing a tail-recursive, garbage-collected language like Scheme.[6] If all functions are transformed into continuation-passing style, if arguments are passed as C arguments (even for variable-arity functions), if C's "stack" allocation is used to perform all allocation (including closure allocation), and if a copying garbage collector is used, then we can achieve a Scheme implementation on top of C which is similar to the implementation of SML/NJ [Appel91], except that it will be much more portable. In our Scheme, the entire C "stack" is effectively the youngest generation in a generational garbage collector!

---

[4] Jérôme Chailloux tells me that the IBM RS6000 and HP 9700 C compilers generate slower "stubs" for indirectly-called C functions—e.g., our closures. Scheme-to-C compilers therefore generate calls to "known" functions wherever possible.

[5] Maximally portable implementations may prefer `malloc` to `alloca` for objects whose size is not known at compile time.

[6] C's which *do* attempt tail recursion—e.g., Gnu C—may be penalized if they restore caller-save registers!

A key feature of our Scheme is that the garbage collector avoids the necessity of tracing garbage, and therefore it need not know the format of this garbage, which includes all of the C stack frames. At the cost of continually checking and following forwarding pointers, we could make our Scheme "real-time" [Baker78]. A previous paper [Baker92] also advocated allocating objects on the stack within the local C frame. However, the current scheme is simpler, since it does not require that objects be forwarded dynamically during execution.

Acknowledgements

Many thanks to Andrew Appel, Jérôme Chailloux, R. Kent Dybvig, and David Wise for their comments on early drafts of this paper (and their fixes for my bugs!).

References

Appel, A.W. "Garbage collection can be faster than stack allocation". *Info. Proc. Letts. 25*,4 (1987), 275-279.

Appel, A.W. "Simple Generational Garbage Collection and Fast Allocation". *SW Prac. & Exper. 19*,2 (1989), 171+.

Appel, A.W. "A Runtime System". *Lisp & Symbolic Comput. 3*,4 (Nov. 1990), 343-380.

Appel, A.W., and MacQueen, D.B. "Standard ML of New Jersey". In Wirsing, M., *ed*. *Third Int'l Symp on Progr Lang Implementation and Logic Progr*., Springer, August 1991.

Baker, H.G. "List Processing in Real Time on a Serial Computer". *CACM 21*,4 (April 1978), 280-294.

Baker, H.G. "CONS Should Not CONS Its Arguments, or, a Lazy `alloc` is a Smart `alloc`". *Sigplan Not. 27*,3 (Mar. 1992), 24-34.

Bartlett, J. "Scheme->C: A portable Scheme-to-C compiler". Tech. Rept., DEC West. Res. Lab., 1989.

Berry, D.M. "Block Structure: Retention vs. Deletion". *Proc. 3rd Sigact Symp. Th. of Comp*. Shaker Hgts., 1971.

Cheney, C.J. "A nonrecursive list compacting algorithm". *CACM 13*,11 (Nov. 1970), 677-678.

Clinger, W., Hartheimer, A., and Ost, E. "Implementation strategies for continuations". *Proc. LFP*, 1988, 124-131.

Danvy, O. "Memory Allocation and Higher-Order Functions". *Proc. Sigplan '87 Symp. on Interp & Interp. Techs., ACM Sigplan Not. 22*,7 (July 1987), 241-252.

Fairbairn, J., and Wray, S.C. "TIM: a simple abstract machine to execute supercombinators". *Proc. 1987 FPCA*.

Fischer, M.J. "Lambda Calculus Schemata". ACM Conf. Proving Asserts. re Progs., *Sigplan Not. 7*,1 (Jan. 1972).

Friedman, D.P., and Wise, D.S. "CONS Should Not Evaluate Its Arguments". In Michaelson, S., and Milner, R., *eds*. *Automata, Languages and Programming*, Edinburgh U. Press, 1976, 257-284.

Hanson, C. "Efficient stack allocation for tail-recursive languages". *Proc. LFP*, June 1990.

IEEE. *IEEE Standard for the Scheme Programming Language*. IEEE-1178-1990, IEEE, NY, Dec. 1990.

Peyton Jones, S.L. *The Implementation of Functional Programming Languages*. Prentice-Hall, New York, 1987.

Stallman, R.M. "Phantom Stacks: If you look too hard, they aren't there". AI Memo 556, MIT AI Lab., July 1980.

Stallman, R.M. *Using and Porting GNU CC*. Free Software Foundation, Inc. February, 1990.

Steiner, J., and Hawkes, B. "The M.T.A." On the album *The Kingston Trio At Large*, released June 8, 1959, reached number 15 on June 15, 1959. Copyright Atlantic Music 1956-57.

Tarditi, D., & Lee, P. "No assembly required: Compiling standard ML to C". ACM *LOPLAS 1*,2 (1992), 161-177.

