A Portable Implementation of First-Class Continuations for Unrestricted Interoperability with C in a Multithreaded Scheme

Marc Feeley
Département d'informatique et recherche opérationnelle
Université de Montréal
http://www.iro.umontreal.ca/~feeley

Abstract

The implementation of first-class continuations in a Scheme system that interfaces to a stack-based language such as C is difficult when Scheme and C frames can be interleaved (i.e. Scheme and C can nest calls to each other arbitrarily). Such a situation occurs when using a combination of callbacks, higher-order functions, exception processing, and a Scheme level thread system built on top of first-class continuations. We show that in this context the use of C threads to implement first-class continuations allows unrestricted interoperability with C.

1 Introduction

Scheme systems that allow linking with C code (and more generally any stack-based language) whether they generate native code or C code, require special treatment of continuations when Scheme continuation frames and C frames can be interleaved. In our examples we will denote continuations with a string of letters; C frames are in upper-case, Scheme frames are in lower-case, and the last frame added to the continuation is on the right. As a running example we will use the continuation "`abCDefGHi`" which corresponds to the chain of frames shown in Figure 1 (the arrows represent a logical link between frames, not necessarily real pointers).

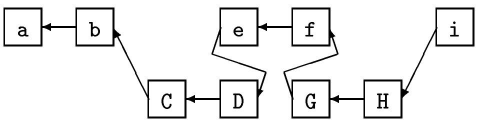

Figure 1: The continuation "`abCDefGHi`".

A context where such interleaving is useful is when a Scheme procedure is passed as a "callback" to a C procedure (e.g. `qsort` or a GUI event dispatcher) and the callback invokes a continuation.

The fundamental problem is that Scheme continuations have unlimited extent whereas C frames have dynamic extent. To simplify, we will assume that Scheme frames are consistently allocated on a heap managed by a garbage collector and that C frames are allocated LIFO on a stack (techniques such as [HDB90] which manage Scheme continuation frames on a stack between calls to `call/cc` are thus viewed as optimizations).

Consequently, in our example the C stack only contains "`CDGH`" and when control returns to `D` there is no way to return to frame `G` and `H` because they have been deallocated from the stack. The fact that C frames cannot be returned to more than once, i.e. that they are "one-shot", is not a concern in this paper. Our aim is to allow control to return to C frames in an arbitrary order rather than LIFO order. This is of particular interest when using Scheme level threads implemented with `call/cc` as explained later.

The following approaches can be used to cope with the inconsistent management of frames by C and Scheme.

2 Approach 1: Copy the C Stack

This approach makes a heap copy of the C stack frame portion of the continuation (i.e. "`CDGH`") when it is captured with `call/cc` (Bigloo [Ser00] essentially uses this approach except that Scheme frames are also allocated on the C stack). Invoking this continuation copies the C frames back to their original location which is needed for correct handling of the "`&`" operator. Aside from being highly C compiler dependent and having non-linear space and time behavior, this approach coalesces the store and the continuation, which means that all assignments to local C variables are obliterated when a continuation is invoked. This major departure from the Scheme semantics is only viable when using C procedures written in a very special style (no assignments to local variables after a continuation capture), which precludes its use with off-the-shelf C libraries. Moreover, pointers to local C variables cannot be passed freely to Scheme code because there is no guarantee that the appropriate C frame is still on the stack when the pointer is dereferenced.

3 Approach 2: Deallocate C Stack Frames Lazily

This approach, which is used by Gambit-C 3.0 [Fee98], consists in removing frames from the C stack (with a normal return or `longjmp`) only when control returns to a C frame. In our example, if the program invokes the continuation "`ab`" and then "`abCDefGHi`" and then "`abCDe`", the C frames "`CDGH`" remain on the C stack. However, if the continuation "`abCD`" is invoked the frames "`GH`" are removed. Note that the continuation "`abCDefGHi`" can still be invoked; it is only when attempting to return to a deallocated C frame that a run-time error needs to be signaled (after all, the code attached to frame "`i`" could decide to invoke the continuation "`ab`"). If not used carefully this approach may result in longer retention of memory and even space leaks (if control never returns to a C frame). To avoid this the programmer may have to force a cleanup of the C stack by artificially returning to an appropriate C frame.

Unfortunately, the lazy deallocation approach does not interact well when using Scheme level thread or coroutine systems implemented on top of first-class continuations, even in a uniprocessor setting where at most one thread is executing at a given time. To understand the problem consider the case shown in Figure 2 where there are two Scheme threads (T1 and T2) attached respectively to the continuations "`abCDe`" and "`fGHi`", and the C frames "`GH`" are allocated on the C stack after "`CD`".

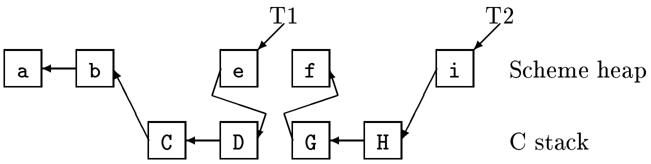

Figure 2: Two threads with interleaved Scheme and C frames.

If T1 is currently executing and it returns to "`abCD`", then the C frames "`GH`" will be removed from the C stack. A context switch back to T2 followed by a return to "`fGH`" is no longer possible. Note that if at most one thread builds a continuation with interleaved Scheme and C frames, then this problem does not arise. Of course we would like to lift this restriction and allow any number of Scheme threads to interleave Scheme and C frames.

4 C Threads Approach

If the management of C frames could be liberated from its strict LIFO allocation strategy we would be closer to a complete solution. But in fact *there is* another way to allocate C frames: C threads! Most platforms include C thread libraries (e.g. POSIX threads, Win32 threads, SUN LWP) which allow the creation of a thread with an associated stack (some thread systems automatically allocate the space for the stack, other systems require an explicit `malloc()`). The stacks of different threads are independent and can be deallocated in any order. Note that this use of threads is unusual because we are not trying to exploit any kind of concurrent execution; we simply want a portable way to allocate a new C stack and perform C procedure calls within it.

What we propose is that each contiguous segment of C frames in the continuation be assigned to one C thread. The creation of a new C stack (i.e. thread) can be done when Scheme calls C; all contiguous C frames added to the continuation will be allocated in this stack. However, because calls to C from Scheme are likely to be more frequent than to Scheme from C, it is probably more efficient to eagerly create a new C stack when C calls Scheme and to start adding C frames to it if and when Scheme calls C. The deallocation of the C stacks (and associated threads) is the responsibility of the garbage collector which will deallocate all C stacks that are not part of a reachable Scheme continuation.

With this approach the C frames in a continuation can be returned to in any order, and the garbage collector reclaims C frames with the same promptness as Scheme frames.

5 Practicality

We have not yet implemented this approach in Gambit-C. It is not clear that it is worthwhile for typical programs because of C thread system limitations and overheads. LinuxThreads for example cannot create more than 1024 simultaneous threads, each thread stack is a large fixed size (2 MBytes), and it takes 2 msecs on a 600MHz Athlon to create and join a thread so there is at most 500 C to Scheme calls per second.

6 Related Work and Conclusion

The use of native threads to implement first-class continuations was proposed in [KBD98]. We have shown that it can be used in a Scheme level thread system to allow an arbitrary nesting of calls between Scheme and C, and that there are no restrictions to the order in which continuations with interleaved Scheme and C frames can be invoked. However, the restrictions and overheads of most C thread systems reduces the practicality of this approach at this point in time.

References

[Fee98] Marc Feeley. Gambit-C version 3.0. Available at `http://www.iro.umontreal.ca/~gambit`, May 1998.

[HDB90] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 66–77, White Plains, New York, June 1990.

[KBD98] Sanjeev Kumar, Carl Bruggeman, and R. Kent Dybvig. Threads yield continuations. *Lisp and Symbolic Computation*, 10(3):223–236, May 1998.

[Ser00] Manuel Serrano. Bigloo 2.2a. Available at `http://kaolin.unice.fr/~serrano/bigloo`, July 2000.