Discussion Questions for
Representing Control in the Presence of One-Shot Continuations

N. Ramsey and B. LaChance, for COMP 250RTS

December 4, 2017

Instructor's commentary

A little bit of contextual information, first about continuations.

- First-class continuations are a superpowered feature. They are widely perceived to be scary, difficult to implement, and expensive—and they are mandated by the Scheme standard. Brian and I scrutinized four different papers that use first-class continuations to implement more familiar things like threads and coroutines, but none of those papers made us happy, so we choose not to inflict them on you.

- In the Scheme community, Chez Scheme is highly regarded. So when Dybvig and his colleagues published their 1990 paper on implementing first-class continuations, lots of people paid attention.

- This paper is a follow-on. On rereading it, I was struck at how slender is the problem that the authors are trying to solve. For myself, I would have been perfectly happy to stick with full ("multi-shot") continuations. But I'm confident that criticism was leveled at using multi-shot continuations to implement things like context switching, which systems like Cormack's would be perceived to do about much lower cost. This paper answers such criticism. I find the experimental results less than thrilling, but somebody had to do the experiments.

  A final note: I am impressed by the design and implementation, and I find them reasonably well described. I am not so impressed by the experimental evaluation.

In addition, one note on calling conventions and stack layout. If your primary experience of language implementation is with the x86 or AMD64 platform, you might not know that the stack structure of "frame pointer, but no stack pointer" (section 3.1, third paragraph) has been standard on RISC platforms since they first started deploying in the 1980s. But this structure has never quite penetrated the Intel world to the degree that it should.

Questions for class

Run-time data structures

(1) We have now seen three different data structures for representing an active call stack:

- The classic call stack specified in the SYSV ABI
- The linked stacks used by SML/NJ and by Lua
- The segmented stack used by Chez Scheme (and described in this paper)

Describe the layout of each representation.

Costs

(2) Section 3.2: When a *multi-shot* continuation is *captured* via `call/cc`,

(a) What new memory is allocated?
(b) What data is copied?
(c) What other linked data structures are created, and where do they reside?
(d) Roughly what memory traffic is needed?

(3) Section 3.2: When a *one-shot* continuation is *captured* via `call/1cc`,

(a) What new memory is allocated?
(b) What data is copied?
(c) What other linked data structures are created, and where do they reside?
(d) Roughly what memory traffic is needed?

(4) At the bottom of Figure 2, we see that capturing a one-shot continuation allocates a new stack segment, and that there is unused space in the captured continuation (the space on the right that is part of N1 but not N2). A one-shot continuation, whose use is restricted, seems to cost more to capture than a multi-shot continuation, whose use is unrestricted.

(a) Is this analysis correct?
(b) If not, what is wrong with the analysis?
(c) If so, what savings elsewhere could possibly justify the increased cost of capturing a one-shot continuation?

(5) Section 3.2: When a *multi-shot* continuation is *invoked*,

   (a) What new memory is allocated?
   (b) What data is copied?
   (c) What other linked data structures are created, and where do they reside?
   (d) Roughly what memory traffic is needed?

(6) Section 3.2: When a *one-shot* continuation is *invoked*,

   (a) What new memory is allocated?
   (b) What data is copied?
   (c) What other linked data structures are created, and where do they reside?
   (d) Roughly what memory traffic is needed?

(7) What cost savings do you expect from one-shot continuations?

Cost Comparison

The third paragraph of the Conclusions section compares the work with the "heap-based representation of control" used in SML/NJ.

(8) The paper briefly mentions an "anti-bouncing" strategy that bounds the overhead of stack overflow and underflow.

   (a) If no such strategy is in place, what goes wrong?
   (b) Does SML/NJ have such a strategy? If so, what is it? If not, does SML/NJ suffer from the adverse consequences of "bouncing"? If not, why not?

(9) Find at least two things wrong with Appel and Shao's model. That is, identify two issues mentioned in the paragraph that you think allow Appel and Shao's analysis to support a misleading conclusion about the merits of representing a call stack as a linked list of heap-allocated activation records.

Bonus questions to take home

How segmented stacks work

(10) How does stack walking work? In particular, if you are looking at a frame, how do you find the caller's frame?

   (a) If the caller is in the same stack segment?
   (b) If the caller is in the next older stack segment?

(11) After `(call/cc f)`, a return from `f` is described as an invocation of the captured continuation. Supposing `f` returns normally, how is the invocation accomplished?

(12) Two parts about stack segments:

   (a) Suppose we are using this run-time system with a language that does *not* expose `call/cc` or `call/1cc` to the programmer. What, if anything, would cause the allocation of a new stack segment?

   (b) In Chez Scheme, what causes the allocation of a new stack segment?

(13) Stack overflow is described as "an implicit `call/cc`" (page 101, bottom right). Here, `call/1cc` is just as good. Explain the model of stack overflow as continuation capture.