Programming Languages (CS302 2007S)

R. Kent Dybvig - Continuations and Continuation Passing Style

Comments on:

Dybvig, R. Kent (1993). Continuations. Section 3.3 of *The Scheme Programming Language*, Third Edition. Cambridge, MA: The MIT Press. Available online at
`http://www.scheme.com/tspl3/further.html#./further:h3`

Dybvig, R. Kent (1993). Continuation Passing Style. Section 3.4 of *The Scheme Programming Language*, Third Edition. Cambridge, MA: The MIT Press. Available online at
`http://www.scheme.com/tspl3/further.html#./further:h4`

I understand that a continuation is a function representing the future of a computation. However, what happens when the same expression is being evaluated simulateously? Let's take the case of a very basic example: `(/ (- x 1) (- x 2))`. What continuation is used in each of the expressions (- x 1) and (- x 2)?

It depends on what order the compiler decides the parameters to / should be evaluated. If it evaluates the first one first, the continuation to `(- x 1)` *is* `(lambda (tmp) (/ tmp (- x 2))` *and the continuation to* `(- x 2)` *is* `(lambda (tmp2) (/ tmp tmp2))`.

While continuations strike me as a powerful method in computation, they also strikes me as really inefficient. One of the reasons we like recursion is because we can do tail recursion wherein we don't have a bunch of information on the stack. Continuations seem entirely dependent on building up a stack, isn't that a poor use of memory?

Memory is fairly cheap. The timing results I've run on tail-recursive vs. non-tail-recursive procedures for the same problem typically show less than a 1% difference in running time. As importantly, well-implemented continuations can actually produce somewhat faster code.

I think I understand conditionals on a semantic level, but I don't understand the syntax for them. They seem to allow for flow control that is part recursion, part iteration, and part conditional behavior. Am I correct in thinking that Continuations are for flow control?

Yes, continuations provide a mechanism for flow control. (You might even say that continuations make flow control something like a first-class data value.)

I would like to see how the call/cc function is implemented in scheme.

Traditionally, the Scheme code is converted to continuation-passing style, so the continuation is immediately available.

The article mentions a number of specific uses of continuations but the different applications seem largely unrelated. What are the general properites of computations where continuations are useful?

When you want behavior that does not follow the standard call/return model, you can (and often must) use continautions. Exceptions seem to be the most common use. Continuations are also nice when you expect to interrupt a computation (which is a kind of violation of the call/return model).

On page 16, R. Kent Dybvig state, “Expressions written in CPS are more complicated, of course, but this style of programming has some useful applications” In what applications is CPS most frequently used, and why do those applications use CPS over the continuations obtained via call/cc?

First, call/cc might not be available (as in Lisp). Second, CPS can start to make more logical sense for describing control flow. It is certainly much more linear than traditional Scheme or Lisp. I’m not sure what applications, other than those I’ll mention in class.

The way I understood the reading was that Scheme is built on continuations. Every function, variable, and list implies a continuation. This paper is demonstrating explicit continuations rather than implicit continuations. Is this a correct understanding?

That’s a bit of an overstatement. Every function call implies a continuation. That’s about it. Scheme makes those continuations explicit, rather than implicit.

Wha’s the difference between the continations in scheme and the "continue" function in Java?

Um ... I’m not sure why you’d think that they are anything the same, other than that they start with the first six letters. The `continue` *keyword in Java says “go back to the top of the loop”. Continuations represent “what happens next”. One is a command, one is a value.*