Programming Languages (CS302 2007S)

Paul Graham - Scheme Continuations

Comments on:

Graham, P. (1993). Scheme Continuations. Section 20.1 of *On Lisp*, pp. 258-266. Prentice-Hall. Available online at `http://www.paulgraham.com/onlisptext.html`.

It makes sense to use continuations sometimes but it kind of looks like a GOTO statement which leaves the code hard to follow since it jumps around. What's the difference between a GOTO and continuations? Is it just that in continuations the stack is saved vs. a GOTO statement where it isn't saved? If that's all, wouldn't Dijkstra have the same problems with a continuation as he would with GOTOs?

You are correct that continuations save the stack (and, arguably, some state) But you are also correct that they can get as ugly as GOTO statements. I don't think that Dijkstra commented on continuations, but I expect that you are correct that he would not have liked them.

How many items is the continuation stack designed to hold? Is there any way to get the second-latest value (or any other value other than the most recent) from the continuation stack?

The size of the continuation stack is somewhat implementation-dependent. In general, it's about as large as the available memory allows. Continuations are atomic, so there's no way to get access to the internals of a continuation. If you need the stack at the previous step, you need to package it up as a separate continuation.

As the writer suggests, why do explicit success and failures continuations help avoid the extra communication necessary to seperate the successful execution of a procedure from an unsuccessful execution?

Suppose a procedure call fails, and it is deeply nested. If it returns a special failure value (rather than using a different control path to exit), the caller must check for that special failure value, and its caller must check for that special failure value, and so on and so forth.

Is it possible to replicate the behavior of continuations in languages like C and Java? How would this be done? While loops and break statements?

I'm sure anything is possible, but it's not easy. The most likely technique would be to implement something like a Scheme interpreter :-). You could also grab the stack pointer and then copy the stack (and, I suppose, the state of any variables) somewhere else in memory. To resume, you would overwrite the stack and restore the state of those variables.

There are certainly programs that run indefinitely. Servers are the most typical example. Servers are supposed to keep running, accepting requests and returning results, ad infinitum. You are correct, however, that we spend a lot of time trying to ensure that programs do not accidentally *run forever.*

On p. 261, Graham says that `(+ 1 (frozen 'safely))` ignores the +. He explains this by saying that the last line calls the stack that was pending at the time the continuation was first created. Does this mean that the call/cc will call the function and pass it to another function representing the current continuation, hence updating this stack and displaying the most current value?

I'm not quite sure what you've asked. `call/cc` *packs up the stack and passes it to its argument, a function that expects a continuation as a parameter. When you finally call the continuation, it replaces the current stack with the old stack.*

With LISP designed mainly for experimentation on Artificial Intelligence, I am guessing computations would eventually grow to be complex. Since continuations were not a built-in feature in Common LISP, how did the programmers negotiate their use against the effect on the machines given the limitations (mainly of small memory) of these machines?

Common Lisp was not really implemented on small machines, so that's not as much of an issue. CPS, like many things that come from the Lisp community, took time to find a home. (Last year, students read an O'Reilly book on Java and Ruby that was pushing for continuation servers as a tehcnique.)

In the following code from page 261, could you explain what the codes do after `append`, expecially the `set!`?

```
(append '(the call/cc returned)
        (list (call-with-current-continuation
                (lambda (cc)
                  (set! frozen cc)
                  'a))))
```

Well, the continuation for the code in the `call/cc` *block is "shove this value in a list and then append the other list to the front of it". The* `set!` *saves that continuation in a global variable.*

/section name="graham-continuations">