Understanding Continuations

Frank Terbeck*

October 2, 2019

Not too many programming languages have the concept of first-class continuations built into them. Scheme is one that does[1], so it will be our vehicle for today's adventure. Now what really is a continuation? In order to get an idea, consider the following simple expression:

```
(+ 5 4)
```

Remember that Scheme uses prefix notation[2], where the first item in a parenthesised list is the operator, that is applied to the list of arguments, which is the rest of the list. `+` is the sum-operator, so the above expression evaluates to `9`. So far, so good. How does this help with continuations? Well, a continuation is a way of bundling up "what's left to do" with respect to an arbitrary sub-expression within a program.

Imagine the perspective of the literal `5` in the expression above. If you are this five, look around you and try to come up with a function, that would represent the rest of the computation after you are done evaluating[3]. So in the expression above, after `5` evaluates to five, what is left to be done in order to finally evaluate to 9? Here's a suggestion:

```
(define (kont v)
  (+ v 4))
```

Indeed, after evaluating the `5` literal, the rest of the expression can be represented as a function, that takes one value and adds `4` to it. And if you stick the value of `5` into that function, it will return `9`, like it must.

Once you realise that you need to take the perspective of a part of an expression in order to figure out what the continuation does, it is not hard to figure out these functions for arbitrarily complex expressions. It just takes a little bit of practice. But thus far, this is just a theoretical idea. How would you work with continuations? How to you *get* to them?

In Scheme (and also Standard ML), there is an operator called `call/cc`, which is short for `call-with-current-continuation`. Its purpose is to capture the continuation at a certain point in a program and make it available to the programmer. The name `call-with-*` suggests, that the operator needs something that it can call. That would be a function. This function has to take one argument and `call/cc` will call that function with the continuation it captured as the function's only argument. This function parameter is the only argument that `call/cc` expects and the return value of the `call/cc` invocation is the value of the the supplied function applied to the captured continuation. Here are simple examples, that use `call/cc`:

*ft@bewatermyfriend.org

[1] See [CR+91] p.28f

[2] See [AS96] p.8f

[3] Number literals evaluate to themselves — but in general, the subexpression that would be picked could be arbitrarily complex.

```
(call/cc (lambda (k) 5)) → 5
(+ (call/cc (lambda (k) 5)) 4) → 9
```

Since these don't **use** the continuation at all, it is straight forward to tell the values these expressions evaluate to. Now for something more interesting: Since Scheme allows us to mutate memory[4], we can just store the continuation in a parameter for later use:

```
(define kont #f)

(+ (call/cc (lambda (k)
              ;; Store the continuation k in the global
              ;; parameter kont, that we defined before.
              (set! kont k)
              ;; Ignore the continuation otherwise, and
              ;; just return 5; so this expression boils
              ;; down to (+ 5 4); though we used call/cc
              ;; to extract the continuation from the
              ;; perspective of our value 5.
              5))
   4) → 9

;; Take a look at 'kont' - this is how Racket pretty prints it.
kont → #<continuation>

(kont 10) → 14
```

Remember when we defined a *function* called `kont` earlier? We figured out that the continuation right after evaluating the `5` is a function that adds `4` to its one and only argument. Now applying the continuation (stored in `kont`) to a value of `10` results in: 14 — just as we predicted.

The captured continuation has a special property, that sets it apart from being a simple function: When it is invoked, it resurrects its execution context entirely and abandons the current one, no matter where the program went in the meantime. This property can be used to implement all sorts of useful high-level control flow facilities. A simple, easily understood one is "early exit". Consider this:

```
(fold * (*) '(1 2 3 4 0 6 7 8 9 10)) → 0
```

This is an expression that computes the product of all values in the list, that is the last argument to `fold`[5]. Since any integer being multiplied by zero results in zero, the computation could stop at the fifth element of the input list. However, `fold` has no concept of exiting early. Since we have `call/cc`, we can easily add that:

```
(call/cc (lambda (return)
           (fold (lambda (x acc)
                   (if (zero? x)
                     (return 0)
                     (* x acc)))
                 (*)
                 '(1 2 3 4 0 6 7 8 9 10)))) → 0
```

[4] I know, I know.

[5] The `*` symbol evaluates to the multiplication function and `(*)` evaluates to the identity element of multiplication, namely `1`.

It's the same expression as before, but it's executed inside the function, that is passed to `call/cc`. That way, we have access to the surrounding continuation. In the call to `fold` we pass a function that's more complex than the direct reference to multiplication function. `fold` will hand the current element of the list as the first argument of its function argument and the current value of its accumulator as the second one. What we need to do is straight-forward: Check if the current argument is zero, and if that is the case, just invoke the surrounding continuation with `0` as its argument. As mentioned earlier, this resurrects the continuations execution context and abandons the current one.

Now in order to see if you understood continuations, at this point in other texts on the subject, you get confronted with mind-benders like the following, which is from [Dyb09] page 64[6]:

```
(((call/cc identity) identity) "Hey!")
```

If you evaluate this at a Scheme prompt, it will return the string `"Hey!"`. This might be guessable, but the question is: Why? Let's analyse the expression's form first: At its most primitive, this expression can be summarised like this:

```
(operator operand)
```

The `operand` has the value `"Hey!"`, trivilly. And since we know the result of the final evaluation is that same value, we can (barring side effects) reason that `operator` has to be the identity function[7]. Since that function appears twice in the original expression, it feels like we are on the right track! A guess would be, that the second incarnation of `identity` somehow gets transposed into the operator spot. Maybe we can just replace it with something else, like `string-upcase`. But if you try, you will notice that things are not quite as easy is they seemed[8]:

```
(((call/cc identity) string-upcase) "Hey!")
→ string-upcase: contract violation
    expected: string?
    given: #<procedure:string-upcase>
```

... and not `"HEY!"` like one might have guessed. Time to take a look at the expression, that is our `operand`:

```
((call/cc identity) identity)
```

With our previous strategy we need to look at `(op identity)`, put ourselves in the perspective of `op` and ask: What function represents the rest of the computation after I am done evaluating myself? And you can follow that strategy pretty mechanically:

```
(define (k v)
  (v identity))
```

This is what gets handed in as the argument to `identity` in `(call/cc identity)`. And since it is `identity`, all it does is return the continuation it is handed. That is all it does. Which means, we end up with this:

[6] According to the author it is "probably the most confusing Scheme program of its size", a Confusion Desity Maximum, if you will.

[7] `(lambda (x) x)`

[8] As we will see later, this intuition is not completely wrong, but there is a twist!

```
((lambda (op) (op identity)) identity)
;; Which can be reduced to:
(identity identity)
;; Which can be further reduced to:
identity
```

Let's come back to our guess-work from earlier, where we naïvely used `string-upcase` instead of `identity`: It would reduce to this: `(string-upcase string-upcase)`. And now all of the sudden the error message makes sense too: It says `string-upcase` expected a string argument, but instead got a procedure; and not any procedure but `string-upcase` itself.

To summarise the above: The `call/cc` uses its `identity` argument to feed the right `identity` *into* itself, ultimately returning itself. So `(((call/cc identity) identity) "Hey!")` reduces to `(identity "Hey!")` which evaluates to `"Hey!"`.

It's a fair bit mind-bendy if you think it all through, but managable if you follow the strategy that was presented earlier. What remains to examine is what continuations are useful for beyond clever puzzles. Well, they are a primitive that allows the implementation of a lot of high-level control structures such as early-exit (as we have seen earlier), exceptions, co-routines, generators, and more. In a language with first-class continuations all these features can be implemented in a library without further language support.

Finally, let's not overlook that there are problems with full, unbounded continuations as well: For details on that, see [Kis12]. To alleviate most of these concerns, we might take a look at delimited continuations. That is, however, a story for another day.

References

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. The MIT Press, 1996. ISBN: 0262011530.

[CR+91] William Clinger, Jonathan Rees, et al. *R4RS, The revised^4 report on the algorithmic language Scheme*. New York, NY, USA: ACM Lisp Pointers, 1991, pp. 1–55.

[Dyb09] R. Kent Dybvig. *The Scheme Programming Language*. 4th ed. MIT Press, 2009, pp. I–XII, 1–491. ISBN: 978-0-262-51298-5.

[Kis12] Oleg Kiselyov. *An argument against call/cc*. okmij.org/ftp. 2012. URL: http://okmij.org/ftp/continuations/against-callcc.html.

This article was published at http://bewatermyfriend.org/p/2019/002/; its markup source can be viewed at https://github.com/ft/continuations.