`call/cc` in Ten Minutes

Spencer Tipping

November 30, 2010

Contents

1 Introducing Continuations

In Javascript you often make AJAX calls to the server, but because these calls don't return immediately you have to specify a callback for the data. The idea of using a callback is sort of like saying, "I know you aren't going to have any data for me right away and I've got other stuff to do now, but when you do get it send it here."

Importantly, it isn't possible to get rid of the callback for AJAX.[1] You simply can't say this:

```
// Using jQuery:
var x = $.getAndWaitForIt('/some/url');
alert('I just got ' + x + ' from the server');
```

Instead, you write a callback function and put the `alert` in there. The reason for this is that the *continuation* of the AJAX call isn't invoked right away. (I'll explain this further in a minute.)

Contrast this with normal operations like addition. You would also never write this:

[1] Well OK, you can through a browser hack, but it really is evil. (Worse than `goto`.)

```
add(3, 5, function (x) {
  alert('Finally, we got the value of 3 + 5!');
});
```

Callback-form is silly for addition because addition is blocking. By the time your addition is finished, the value is available and the code can move on.[2] So rather than allocating a callback, you can just say `var x = 3 + 5`. Here, the continuation of `3 + 5` is assigning into `x` and going to the next statement.

1.1 Getting to the point

A continuation, intuitively speaking, is just the next thing that you're going to do. So each of these things is a continuation:

1. An AJAX callback
2. "`var x = `" in the expression `var x = 3 + 5`
3. `b` in the block `{a; b;}`

Every language needs some way of specifying continuations. If there weren't one, exactly one operation would occur and then your program would stop. Most languages, Javascript included, use sequential execution as their continuation model; so you get items 2 and 3 above without really thinking about it.

1.2 Continuation-passing style

One AJAX callback is bad enough, but suppose you have two of them and one depends on the result of the other. This happens a lot for REST APIs that first give you a list of names, and then require a separate call to get an item from a name:

```
$.getJSON('/people', function (people) {
  // Let's just get the first person:
  $.getJSON('/people/' + people[0], function (person) {
    // And now we can get on with the app
  });
});
```

If you've run into this, you may have factored it out into its own function like this:

[2] And because Javascript is single-threaded, nothing else can be going on while the addition is happening.

```
var load_person = function (callback) {
  $.getJSON('/people', function (people) {
    $.getJSON('/people/' + people[0], callback);
  });
};
load_person(function (person) {
  // ...
});
```

Code written this way passes around `callback`, which is a continuation, so it's said to be written in *continuation-passing style* (CPS).

1.3 CPS-conversion

Any code with a well-defined evaluation order can be converted to CPS. I'll go through some examples:

1.3.1 Basic sequences

```
// Regular form:
var add = function (x, y) {return x + y};
var x = add(3, 5);
alert(x);

// CPS:
var add = function (x, y, k) {k(x + y)};
add(3, 5, function (x) {
  alert(x);
});
```

This example is a good place to start because it illustrates a fundamental difference between CPS and regular code. In regular code, functions return things. In CPS, functions pass things along. You'll probably notice that there isn't a return statement in the CPS-converted `add` function; that's because CPS-converted functions theoretically never return. Any return statement gets turned into another function call.

Another fundamental thing is that CPS-converted code often uses `k` to name a continuation. This is patently absurd, as "continuation" clearly starts with a `c`.

1.3.2 Composed functions

```
// Regular form:
var f = function (x) {return x + 1};
var g = function (x) {return x * 2};
var composed = function (x) {
```

```
  return f(g(x));
};
var x = composed(10);
alert(x);

// CPS:
var f = function (x, k) {k(x + 1)};
var g = function (x, k) {k(x * 2)};
var composed = function (x, k) {
  f(x, function (xPrime) {
    g(xPrime, k);
  });
};
composed(10, function (x) {
  alert(x);
});
```

I took a shortcut here, but to make a point. `composed` can be written out longhand like this:

```
var composed = function (x, k) {
  f(x, function (xPrime) {
    g(xPrime, function (result) {
      k(result);
    });
  });
};
```

However, this misses the point that functions are just places to go. There isn't a reason to use the innermost function because it just ends up calling `k` with whatever you give it.[3]

2 The "Current" Continuation

I mentioned earlier that "`var x =`" is a continuation. This probably seems weird, but it should make sense if you consider that it turns into a function definition in CPS. Returning to the regular world, though, it should become clear that there are a lot of continuations. One per expression, in fact. Each time an expression is evaluated, it sends its value to the "current continuation". So, for example, consider the expression `3 + (4 * 5)`. The "current continuation" for `4 * 5` is `3 + x`, where `x` is the result.

Here's where Scheme does something cool. Scheme lets you grab the "current continuation" at any point and use it later on. So, for example, consider this code:

[3] In the lambda-calculus, getting rid of the inner function is called η-reduction, and you can do this safely in any non-pathological language. Javascript is sometimes pathological by this definition.

```
(display (+ 3 6))

// If you're allergic to Lisp:
display(3 + 6);
```

Now let's grab the current continuation for `6` and call it manually:

```
(display
  (+ 3 (call/cc (lambda (k)
         (k 6)))))

// If you're allergic to Lisp:
display(3 + call_cc(function (k) {k(6)}));
```

This code does the same thing as the first example. The simplest way to explain what's going on is that `call/cc` gives you a function that sends some value into `call/cc`'s continuation.[4] The data then goes into that continuation's continuation, etc. And that's where things start to get interesting.

2.1 Escaping continuations

You're probably familiar with `try` and `catch` from Java-like languages. You can use `call/cc` to emulate these. For example:

```
var might_fail = function () {
  throw new Error('Oh no! We hit an error!');
  alert('And this will never be run');
};
try {
  might_fail();
} catch (e) {
  alert('This is the continuation of throw');
}
```

Now let's write an approximation in CPS with `call_cc`:

```
var might_fail = function (k) {
  k(new Error('Oh no! We hit an error!'));
  alert('And this will never be run');
};
var e = call_cc(function (k) {
  might_fail();
});
alert(e);
```

[4] Hence the name, `call-with-current-continuation`.

This isn't quite like the original because it doesn't manage the bookkeeping of knowing whether an error occurred vs. was returned, but the important thing is that continuations can bypass normal execution. The continuation that `call/cc` creates returns directly from the `call/cc` and continues from there.

2.2 Re-entrant continuations

Here's where things get awesome. You know how `call/cc` gives you a function? You can do whatever you want to with that function, including stashing it somewhere to use later. And even better, you can use it multiple times. Here's a simple way to do that (in Scheme, so that you can actually run it):

```
(define (print-lots-of-stuff)
  (let ((continuation '()))
    (display (call/cc (lambda (k)
      (set! continuation k)
      "Hello there")))    ; If you return a value, it gets returned as usual
    (newline)
    (continuation "This will print forever!")))
(print-lots-of-stuff)     ; This kicks off an infinite loop
```

Here's roughly what's going on when you call `print-lots-of-stuff`:

1. We enter the function and set `continuation` to nil
2. We see the `display` call and create a continuation to call it once the parameter is ready
3. We then pass that continuation into the `(lambda (k) ...)`
4. We store that continuation into `continuation`
5. The `lambda` returns `"Hello there"`, which is then sent to `call/cc`'s continuation
6. `display` is `call/cc`'s continuation, so `Hello there` is displayed
7. `(newline)` is `display`'s continuation, so a newline is printed
8. The string `"This will print forever"` is sent to the continuation in `continuation`
9. `continuation` stores the continuation for `display`, so `This will print forever` is displayed
10. `(newline)` is `display`'s continuation, so a newline is printed
11. Step 8 happens again, causing an infinite loop