`call/cc` in Ten Minutes

Spencer Tipping

April 7, 2012

Contents

1 Introduction

I'm going to go off on an inexplicable tangent for a minute here, but I promise it has something to do with continuations.

I was writing an ext4 filesystem driver recently and came across something I didn't expect. You know how you can say `cd ..` to go up one directory? I always assumed this worked by removing a directory from your current location, sort of like returning from a function. But it doesn't work this way at all. Instead, every directory in ext4 actually has a *subdirectory* called `..` that you can `cd` *into*. This special subdirectory points back to the parent of whichever directory you're in.[1]

At first this made no sense to me because it meant that your path would just keep growing forever. But then I realized that programs don't store their path, they store only their current directory by its inode. This meant that it didn't matter how you got to a directory, it only mattered where you were and where you could go from there.[2]

So instead of a call stack working like this:

```
/           cd usr (add usr/)
/usr/       cd bin (add bin/)
/usr/bin/   cd ..  (remove bin/)
/usr/
```

it in fact works like this:

[1]Each directory also has a subdirectory called `.`, which points back to the current directory.

[2]The path can be constructed after the fact by continuously `cd`ing into `..` until the root is reached. Wild, huh?

```
/              cd usr (follow usr/ pointer)
/usr/          cd bin (follow bin/ pointer)
/usr/bin/      cd ..  (follow ../ pointer)
/usr/bin/../
```

Now imagine that your path is like a function call stack. You `cd` into a subdirectory, you do stuff, and you `cd` back out of it. Logically this resembles a single function call, and if `cd ..` were implemented as an "undo" of the last `cd` it would indeed be a real stack. However, each `cd` is doing the same thing: It's calling into a subdirectory of wherever you are. And this is exactly how continuations work.

2 `return`

If you've written code in any normal programming language like C, Java, Javascript,[3] etc, you've probably written the word `return`. And further, you've probably used it to escape from a function earlier than you normally would. For instance:

```
var factorial = function (n) {
  if (n === 0) return 1;
  var result = factorial(n - 1);
  return n * result;
};
```

If the first `return` is hit, the second one will never happen. So `return` is really doing two things; it's setting up the value that the function will have, and it's jumping back to the parent function. The jump works because when we called into `factorial`, we left the return address on the stack. `return` doesn't really know where to go, it just goes to whichever address the caller specifies.

Let's step back for a moment and think about a different interpretation of the word `return`. What if it were a function? The call into `factorial` is a jump, and the return from `factorial` is also a jump (just like `cd` *`directory`* and `cd ..` are both jumps). Are they really the same thing?

As it turns out, they are indeed.[4] Here's the normal way you'd call `factorial`:

```
alert('About to call factorial');
var x = factorial(5);
alert('The factorial of 5 is ' + x);
exit();     // Exits immediately and never returns.
```

And here's what it looks like when we model `return` as a function:[5]

[3] Ok, Javascript isn't normal, but you get the idea.

[4] With the exception of how state is saved, which I'll get to in a bit.

[5] My original `factorial` definition was incorrect; thanks to Brian Ollenberger for submitting the correct version shown here.

```
var factorial = function (n, ret) {
  if (n === 0) ret(1);
  factorial(n - 1, function (result) {
    ret(n * result);
  });
};

alert('About to call factorial');
factorial(5, function (value) {
  var x = value;
  alert('The factorial of 5 is ' + x);
  exit();   // Exits immediately and never returns.
});
```

Todo: explain the `exit()` function.