A Tutorial on Continuations

Nathaniel Gray

June 20, 2002

1 A few notes on Python

The examples in this tutorial are written in Python. Even if you don't know Python you shouldn't have much trouble reading them since Python has the virtue of being very easy to read. There are, however, a few bits of Python syntax that need to be introduced so that the examples make sense to Python newbies.

- Indentation is used to denote block structure. This means that there are no curly-braces or begin/end statements in Python programs
- The `#` character is used to begin line comments
- Python Lists support efficient subscripting of arbitrary elements (unlike LISP or ML lists)
- Square braces are used to construct and subscript lists. `x = [1, 2, y]` makes a three-element list and `x[1]` refers to the second element in the list (equal to 2).
- The + operator is used to concatenate lists
- The : operator within list subscript operations is used for indexing ranges. In particular, `l[1:]` returns the list of all of the elements in `l` except for the first one
- The syntax: `x = [e1 for v in l if pred]` is what's known as a *list comprehension*. It is equivalent to the following code:

```
x = []  # An empty list
for v in l:
    if pred:
        x = x + e1  # Append e1 to the list
```

Note that `pred` and `e1` are expressions that can depend on `v`. So if you wanted to return a list of the squares of all even elements in a list `l` you might write: `[v*v for v in l if v%2 == 0]`

- Functions are first-class in Python. This means that a function can be assigned to a variable or passed as an argument to a function. Here's a short example:

```
def sqr(x):
    return x*x

def apply(f, x):
    return f(x)  # f is a function

y = sqr      # Assigning a function to a var.
apply(y, 5)  # Returns 25
```

2 Continuations: Mysterious? Frightening?

Continuations are a programming language concept that many people find very confusing. This confusion is not only experienced by neophytes, but also by hackers of great renown. For example, at the first Lightweight Languages Workshop hosted by the MIT AI Lab in November of 2001, the Perl language was represented by Simon Cozens and Dan Sugalski. Eugene Kim's report on the workshop for Dr. Dobb's Journal relates that during Sugalski's presentation, "[he] explained that Perl's run-time engine needed to support a number of high-level programming concepts, and he showed a slide that listed some of them. When coming across 'continuations' on his slide, he said, 'Let's skip that. I don't really understand them.' Gasps were heard throughout the room ..."

In reality, continuations are not very complicated at all. A continuation is simply a function that, once called, never returns control to its caller. The way you use them, though, and what you might want to use them for is not at all clear from such a simple description. To understand the implications of "functions that never return" we need to take a detour and discuss the stack-based model of computation that many programming languages and/or compilers use at run-time.

3 The Stack

At run-time in most popular imperative programming languages (C, C++, Java, Python, etc.) when you call a function, a chunk of information about the function and all of its local variables is pushed onto a stack. This stack is generally referred to as the *call stack*, the *execution stack*, or in many cases just *The Stack*. The information placed on the stack is known as a *stack frame*, a *call frame*, or an *execution frame*. When control returns from the function call, the call frame is popped back off of the stack. So as the program executes, the stack grows and shrinks with each function call and return, and the size of the stack is proportional to the "depth" in function calls of the current state of the program.

In some languages, like C and C++, this stack-based implementation is deeply tied into the design of the language. In others, like Python, the stack is an artifact of the way the compiler was implemented and a different implementation might not use one. In fact, there is a version of Python that does not use a stack. This version was created by Christian Tismer and is called, appropriately enough, Stackless Python. (http://www.stackless.com)

The philosophy of this model of computation might be described as "do your computation, then give me back the result," since the control flow of the program always comes back to a function's caller. Each caller's stack frame must be preserved while the callee executes because it has to be there when the callee returns, even if the caller doesn't do anything after it gets control back. This philosophy is so deeply ingrained in the mind of the average programmer that it becomes an unspoken assumption about the way programming works. In fact, there's no requirement that a programming language work this way, and there are many times when the stack is more a hindrance than a help.

4 The Problem with The Stack

The main drawback of using an execution stack is that the stack only has limited space, and each stack frame takes up space on it until its function returns. In functional languages (Lisp, ML, Scheme, etc.) recursion is often used in places where loops would be used in imperative languages. In these languages you often need to recurse to arbitrary depths to complete a computation, so a stack-based control scheme just wouldn't work—you would run out of space on the stack before you could finish the computation. It would fill up with frames, each saying, "do your computation, then give me

```
def qsort(l):
    if len(l) <= 1:
        return l
    pivot = l[0]    # The pivot is the first element of the list
    tail = l[1:]    # We operate on the rest of the list
    lower = [x for x in tail if x<pivot]    # Get list of elements < pivot
    upper = [x for x in tail if x>=pivot]   # Get list of elements >= pivot
    # Recursively sort the sublists and return
    # the pivot joined with the sublists
    return qsort(lower) + [pivot] + qsort(upper)
```

Figure 1: A (rather naive) recursive implementation of the quicksort algorithm

back the result." If we want to write a compiler for these languages we have to find a different technique.

Even in imperative languages there are times when the stack gets in the way. In many cases, a recursive algorithm is much easier to write and understand than an equivalent iterative one. The quicksort algorithm, for example, uses a "divide and conquer" strategy, where a list is split into two smaller lists, each sub-list is sorted, and the results are joined to form the final sorted list. This process is quite natural to describe recursively, as demonstrated in Figure 1.

This is almost identical to the way that the algorithm would be described mathematically. Versions of quicksort that *don't* use recursion, on the other hand, become very long and convoluted, losing the obvious connection to the definition of the algorithm and introducing many opportunities for programmer error. If you're using a stack-based compiler and need to be able to sort lists of arbitrary length, though, you have to use a non-recursive implementation or risk running out of stack space.

5 Going Stackless

So what does any of this stack business have to do with continuations? Using continuations we can eliminate the stack, changing from "do your computation and give me back the result" to "here's my result, you do the rest". We do this by chaining together a list of continuations—functions that never return. The way this works is that each function ends in a so-called "tailcall" to the next continuation. Figure 2 is a really contrived Python

```
>>> import sys
>>> def print_num(x):
...     print "  print", x
...     sys.exit()  # Tailcall to sys.exit()  (If this wasn't here then
...                 # print_num wouldn't be a continuation -- it would return)
...
>>> def add35(x):
...     print "  add35"
...     print_num(x + 35) # Tailcall to print_num
...
>>> def square(x):
...     print "  square", x
...     add35(x*x)  # Tailcall to add35
...
>>> square(add35, 10)
  square 10
  add35
  print 135
```

Figure 2: An example of programming with continuations

example that calculates and prints $10^2 + 35$ using continuations.

Now why on earth would you ever want to write code like this?? Actually, for simple computations like this one you probably wouldn't, but there are reasons you might want your compiler to *transform* your usual code into something similar to this during compilation. There are also other more advanced situations where you might actually want to write programs in so-called "continuation-passing style."

What is the advantage of this scheme? Notice that there are no return statements anywhere! Once a function has finished, control never returns to it. This means we don't need to keep a stack of call frames sitting around to use after returning from function calls—we can just throw away each function's frame after calling its continuation[1]. You can think of the control flow as being a singly-linked-list of call frames instead of a growing and shrinking stack of frames. The practical benefit to the programmer is that since the run-time environment only needs to keep one (or a few) frames around regardless of recursion depth, she can now use recursive algorithms

[1] This is a bit oversimplified, but I don't want to discuss garbage collection and closures in this tutorial.

without fear of running out of stack space.

Earlier, I mentioned that a compiler can transform traditional code to continuation-passing-style code behind the scenes. This process is called CPS transformation. Let's look at how CPS transformation works, using a simple Python function.

```
def greet_user():
    print "Please enter your name"
    n = read_name()
    print "Hello,", n
    return n    # Return the name
```

There are two goals of CPS transformation:

- Transform return statements into tailcalls
- Transform all function calls into tailcalls

Let's start with the return statement. We want to change this into a tailcall to a continuation, but what continuation do we use? Well, we don't really know what the program needs to do after calling `greet_user`, and if `greet_user` is called from different places it might even need to do different things each time. The only sensible thing to do is to require the caller to pass in a continuation for us to call. So our new version looks like this:

```
def greet_user( cont ):
    print "Please enter your name"
    n = read_name()
    print "Hello,", n
    cont( n ) # Pass the name to the continuation
```

Next we need to turn all function calls into tailcalls. The only function call in `greet_user` is the call to `read_name`, and it appears in the middle of the body of the function. When we transform this into a call to a continuation that never returns control back to us, how will we execute the rest of the body of the function? The answer is that we have to split the rest of the body off into its own function and then pass this function to `read_name` as its continuation. Here's how that will look (assuming we've already CPS transformed `read_name`):

```
def greet_user( cont ):
    # Define a function to finish greet_user after
```

```
# A quicksort implementation that uses continuation-passing style
# Notice that cont is a continuation.
def qsort(cont, l):
    # Some variables we'll need in intermediate computations
    lower, upper, pivot = [], [], None

    # First the compiler needs to define some functions used for
    # intermediate computations. Notice that these functions don't take
    # continuations as arguments.

    # Join the sorted lists and call the main continuation
    def q_finish( sorted_upper ):
        cont( lower + [pivot] + sorted_upper )

    # Sort the upper elements
    def q_upper( sorted_lower ):
        lower = sorted_lower
        # Recursively sort the elements that are >= pivot, passing
        # the result to q_finish
        qsort( q_finish, upper )      # RECURSIVE CALL
```

Figure 3: Recursive quicksort algorithm after CPS transformation (Part 1)

```
    # the call to read_name
    def finish_greet( n ):
        print "Hello,", n
        cont( n )   # Pass the name to the continuation
    print "Please enter your name"
    # Tell read_name to pass its result to finish_greet
    read_name( finish_greet )
```

That's it! For a more challenging example of CPS transformation, let's have a look at a CPS-transformed version of our quicksort program[2] (Figures 3 and 4), making some simplifying assumptions about which operations require function calls.

[2]Note that this is not quite valid Python code. Python's scoping rules are a bit messy and this style of programming exposes one of the rough edges. Python wasn't designed for coding with continuations, so it shouldn't be too surprising that the straightforward way of doing it doesn't work.

```
    # The continuation for the call to len
    def q_len_cont( length ):
        if length <= 1:
            cont(l)
        # For brevity, we'll assume these list operations don't require
        # function calls
        pivot = l[0]
        tail = l[1:]
        lower = [x for x in tail if x<pivot]  # Get list of elts < pivot
        upper = [x for x in tail if x>=pivot] # Get list of elts >= pivot
        # Recursively sort the elements that are < pivot, passing
        # the result to q_upper
        qsort( q_upper, lower )     # RECURSIVE CALL

    # Kick off the whole process with a call to len.
    # Find the length of the list, then pass it to q_len_cont
    # (We assume the len function is built-in and takes a continuation
    #  as its first argument.)
    len( q_len_cont, l )

# Now to actually use qsort, we would do something like this:
import random, sys
l = range(15)       # A list of numbers from 0 to 14
random.shuffle(l)   # Randomize the numbers
print "Input: ", l

# Make a continuation that prints its argument then exits
def prt( l ):
    print "Result: ", l
    sys.exit()  # Again, we have to exit for this to be a continuation

# Call qsort and pass it our list and continuation
qsort( prt, l )
```

Figure 4: Recursive quicksort algorithm after CPS transformation (Part 2)

There are a few interesting things to notice about this example. First notice that every execution path of every function ends in a tailcall to a continuation. There are no return statements anymore so we can mostly eliminate the stack. Next notice that every function call in the original function has been replaced by a tailcall to that function.

Finally, notice that qsort takes its continuation as its first argument. This is for the same reason mentioned in the discussion of `greet_user` in the last example. The results of qsort can be used in different ways depending on where it was called from. After an array is sorted, what to do next depends on factors external to qsort. Thus qsort must have its continuation provided by the caller. We have also assumed that the "len" built-in takes a continuation as an argument for the same reason. You can think of this model as "here's my result, do your computation and give the result to *him*," where "him" is the continuation.

In fact, the same fact is true of *any* user-defined function. So in CPS transformation, user-defined functions are *always* transformed to take their continuations as arguments.

Contrast this to the various *internal* functions that the compiler defined (those beginning with "`q_`"). In these cases the continuation is fixed at compile time. `q_upper` *always* ends with a tailcall to `qsort`, for example, so this tailcall can be hardwired at compile time. Thus, none of these functions take continuations as arguments.

That's it! All returns and function calls have been changed to tailcalls and the stack can be eliminated[3]. Remember, though, that the real Python interpreter doesn't do CPS transformation. If you write recursive code in Python you *will* have to worry about overflowing the stack (unless you use Stackless Python).

There is one other important benefit that continuations can give us. Imagine that you wanted to take a program in mid-computation and freeze it, then restore it later. This is the idea behind multithreading—threads get suspended and restored all the time. In a stack-based language, saving the state of a thread means saving its entire call stack. If we eliminate the stack using continuations (in combination with some other powerful techniques) then we need to save much less information, and context switching time

[3]The alert reader will notice that I haven't quite eliminated the stack. I'm still assuming that the variables `lower`, `upper`, and `pivot` can be referenced from nested scopes (within `q_upper`, for example), which suggests that qsort's frame is kept around during the execution of all of these inner functions. A technique called closure conversion can be used to eliminate this last remnant of the stack, but that's beyond the scope of this tutorial. The important thing, for now, is that all return statements have been eliminated.

(the amount of time it takes to suspend one thread and restore another) is dramatically reduced. Individual threads also take up much less memory. This is why Christian Tismer's Stackless Python is able to support tens of thousands of concurrently running threads (or micro-threads, as he calls them) without breaking a sweat. Not too shabby for an interpreted language!

6 Recap

So to summarize continuations:

- A continuation is a function that is called but never returns control to its caller.
- A call to a continuation at the end of a function is called a tailcall.
- Using continuations one can transform programs to eliminate return statements.
- Programs without return statements do not require a stack of execution frames at run-time.
- A compiler that exploits continuations can allow the programmer to use recursion without worrying about running out of stack space.
- A compiler that exploits continuations can also implement multithreading in a very efficient manner.