

Implementing Tail Recursion in Python

Mike Slegeir

mikeslegeir@mail.utexas.edu

The University of Texas at Austin

Department of Computer Sciences

December 10, 2007

Tail recursion is a style of recursion which can be easily optimized to perform comparably to iteration because there is no need to create a new stack frame. Using this style of programming allows the programmer, in some cases, to write very concise, readable code which performs as fast as a slightly more complex, iterative solution would. However, the optimization isn't inherent in the format of the code, the compiler or interpreter must recognize this format and perform the optimization itself (called tail call optimization). Therefore, although most mainstream languages do perform this optimization (eg: C, C++, Java, Scheme, Haskell, etc), Python is one of the few languages that does not. The utility of recursion in Python is even further limited by the restriction on the depth of recursion: there is a set limit (default: 1,000) to the number of recursive calls a function can make before an exception is thrown. Thus, any programmer who wishes to use recursion not only can not optimize away the actual recursion by writing his recursive call in tail call position, but must be wary of the limit that is imposed on the depth of the function's recursion. Fortunately, the tail call optimization solves both the recursion depth and performance problems (since there are no new stack frames being created) without sacrificing the conciseness of a recursive solution.

The target of my research is to learn about how function calls in Python work in order to add tail call optimizations in such a way that it is simple, efficient, and natural. That is, the programmer writing a tail recursive function shouldn't need to do anything more to his code than writing the recursive call in tail call position (aside from setting a flag to optimize tail calls). The tail call should be relatively simple to execute and a tail recursive function should perform comparably to an iterative solution. Finally, the tail call optimization should fit into and remain consistent with implicit and explicit standards of Python. My main goal is to demonstrate not only that the tail call optimization can be performed in Python, but it can be done elegantly, and that it would be a beneficial addition to the Python language because it can allow for concise solutions to problems without sacrificing performance benefits of iteration.

My first approach to performing the tail call optimization (abbreviated as TCO from now on) was a proof of concept approach: I created a function (in Python) which given another function, would perform the TCO and return an optimized function. I identified a pattern in the byte code of functions which are tail recursive:

```
LOAD_GLOBAL <func>
<Evaluate and load arguments onto stack>
CALL_FUNCTION <# arguments>
RETURN_VALUE
```

I performed the optimization by creating a new function consisting of just a 'pass'. I then copied the byte code from the given function into the new one until I reached the instruction 'LOAD_GLOBAL func'. I then copied the code into the new function slightly modified to the form:

```
<Evaluate and load arguments onto stack>
STORE_FAST <arg n>
...
STORE_FAST <arg 0>
JUMP_ABSOLUTE <Label at top of function>
```

The new byte code takes the arguments which are on the stack to be applied as arguments to the function call and stores them into the appropriate arguments and then just branches to the top of the function and starts over. This solution meets the efficient requirement of the TCO, and the basic code structure is the best I had come up with. However, it is neither simple nor natural: no one wants to have to call a function on the function they wrote in order to perform an optimization; Python itself needs to take care of it. Not only that, but the function makes a lot of assumptions: namely that the function that is given to it is tail recursive, and because if these assumptions aren't met, the function will fail, it puts a big burden on the programmer to understand when to use the TCO function and to use that discretion when applying the function. Therefore, this style of solution is far from desirable, but a good starting and reference point in actually implementing the TCO in Python because it does work for basic functions which use pure tail recursion (ones that only make a single function call to themselves).

The next step was to take the external TCO function and move it into the internals of Python. I was naturally looking to do something very similar to what I had done with the external function, but as an automatic optimization, so the byte code compiler could decide if it was appropriate to perform the TCO; however, I was still trying to detect the same byte code pattern and modify it in a similar fashion. I had, however, since realized that the true signature of the tail call was the last byte codes of the pattern:

```
CALL_FUNCTION <# args>  
RETURN_VALUE
```

By looking for just this pattern, I could detect and optimize any tail call (simple recursion, mutual recursion, or a non-recursive but still optimizable tail call). I discovered that Python will run a few peephole optimizations (that is, will perform simple byte code manipulations based on special behavior) and performs the optimizations in the file `'Python/peephole.c'`. Although this environment seemed ideal for the exact same sort of byte code manipulation I had performed in the external function, the byte code modification was performed in-place so that no new the size of the byte code could not change. This was an issue because the size of the byte code generated in my modification was a function of the size of the number of arguments and it wouldn't be possible to fit support for functions with any more than one or two arguments.

Since it seemed difficult to perform the byte code optimization within Python, I did some investigation on detecting a tail call in the abstract syntax tree. I had a hard time finding what I was looking for in the parser code, and I often found myself accidentally trying to read through code which was automatically generated. Eventually, I got frustrated trying to follow the chain of functions involved in generating the byte code, and I just came up with another idea.

My idea was to create a new byte code which was specific to tail calls: `CALL_FUNCTION_TAIL`, to be used as a drop-in replacement for the `CALL_FUNCTION`, `RETURN_VALUE` pair. However, it won't create a new stack frame or, at the worst, replace the current stack frame with a new one. The first thing to do is create the definition of a new opcode. I simply added a new define to `'Include/opcode.h'`:

```
#define CALL_FUNCTION_TAIL 144
```

I then needed to detect the tail call and replace it with my new opcode. I did this in `'Python/peephole.c'` in the `switch(opcode)` statement in `PyCode_Optimize`:

```
case CALL_FUNCTION:
    if (codestr[i+3] != RETURN_VALUE ||
        !ISBASICBLOCK(blocks,i,4))
        continue;
    codestr[i] = CALL_FUNCTION_TAIL;
    break;
```

Which when it reaches a `CALL_FUNCTION` opcode which is immediately followed by a `RETURN_VALUE` opcode, it replaces the `CALL_FUNCTION` with my opcode, `CALL_FUNCTION_TAIL`. (The `RETURN_VALUE` is left because currently, `CALL_FUNCTION_TAIL` defaults to `CALL_FUNCTION` when its not recursion, for reasons that will be clear later, and therefore the `RETURN_VALUE` is still necessary). The code for all the opcodes is found in `'Python/ceval.c'` in `PyEval_EvalFrameEx` and we define the simple behavior (only supporting pure tail recursion) as such:

```
case CALL_FUNCTION_TAIL:
{
    int i;
    for (i=oparg-1; i>=0; --i){
        // For each argument
        // Pop it
        x = POP();
        // Store it in the appropriate arg
        SETLOCAL(i, x);
    }
    // Pop the function from the stack
    u = POP();
    // Jump to the top of the frame and start over
    JUMPTO(0);
    continue;
}
```

This behavior is analogous to the earlier, external solution: for each argument of the function being called (this is the argument to `CALL_FUNCTION_TAIL` or `oparg`) we store it back from the stack. However, since I didn't do away with the `LOAD_GLOBAL <func>` as I had in the external solution, I have to remove it from the stack (Note: I might have to decrease the reference count of the function, but I could not be sure that is the proper behavior since the function is still being referenced). The next statement (`JUMPTO(0);`) branches to the top of the function (since the function should be the only thing loaded in the frame at the time). Then `continue;` simply continues the byte code evaluation loop from the top.

The major difficulty in implementing a general TCO in Python, in my opinion, is the way the frames work. The reason the optimization is so simple in C is because all the compiler has to do is store the arguments to the appropriate place on the stack or in the right registers and branch back to the top of the function. The only limitation to branching to different functions is the reachable distance of the branch instruction in the assembly language (even this is easily overcome by a more complex branch) that it is being compiled to. However, in Python's byte code interpreter, its instruction memory is not so flat: only one frame of code (in my experience: one frame equates to one function's code from a single function call) is loaded at a time. Thus, we can not simply `JUMPTO` any other function we please in my implementation of `CALL_FUNCTION_TAIL`. A call to another function must then either modify the current or construct its own frame object. This becomes an issue because although this still eliminates the stack being filled by recursive calls, it loses the time benefits of not having to construct a new stack frame when recursing. Thus, we must detect tail calls which are not recursing by augmenting the behavior of `CALL_FUNCTION_TAIL` by inserting this code at the top:

```

    u = stack_pointer[-1 - oparg];
    // Try comparing the function's name to f->f_code->co_name
    if( PyObject_Compare(co->co_name, ((PyFunctionObject*)u)->func_name) )
#ifdef IMPLEMENTED_NON_RECURSIVE_TAIL_CALL
        // Here we could modify the current stack frame
#else
        goto dont_tail_call;
#endif

```

Where the label *dont_tail_call* is at the very beginning of case *CALL_FUNCTION*. I wasn't able to define the behavior of a non-recursive tail call in time, so currently, I am just passing the buck to *CALL_FUNCTION*, but one would simply fill in the code where the comment is and define *IMPLEMENTED_NON_RECURSIVE_TAIL_CALL* in order to define this behavior properly. With the current workaround, mutual recursive functions would still hit the stack limit, but if the code is properly defined to implement the non-recursive tail call, no tail call would hit the recursive ceiling.

In order to test the functionality of the TCO, I created several test functions and created iterative counter-parts in order to compare performance (note: the iterative functions aren't the natural solutions, but are designed to compute the same as the tail recursive solution, but using iteration):

```
def pow(x, n, acc=1):
    if n == 0:
        return acc
    elif n < 0:
        return pow(x, n+1, acc/x)
    else:
        return pow(x, n-1, acc*x)
def fact(n, acc=1):
    if(n == 0):
        return acc
    else:
        return fact(n-1, n*acc)
def foldl(f, acc, list):
    if list == []:
        return acc
    else:
        return foldl(f, f(acc, list[0]),
                    list[1:])
def reverse( list, rev=[] ):
    if list == []:
        return rev
    else:
        return reverse(list[:-1],
                    rev + list[-1:])
def enum(n, list=[]):
    if n == -1:
        return list
    else:
        return enum(n-1, [n] + list)

def pow_iter(x, n, acc=1):
    while n != 0:
        if n < 0:
            n += 1; acc /= x
        else:
            n -= 1; acc *= x
    return acc
def fact_iter(n, acc=1):
    while n != 0:
        acc *= n
        n -= 1
    return acc
def foldl_iter(f, acc, list):
    while list != []:
        acc = f(acc, list[0])
        list = list[1:]
    return acc
def reverse_iter( list, rev=[] ):
    while list != []:
        rev = rev + list[-1:]
        list = list[:-1]
    return rev
def enum_iter(n, list=[]):
    while n != -1:
        list = [n] + list
        n -= 1
    return list
```

The functions test various things like default arguments, arguments as return values from functions, multiple recursive calls, etc; however, they are all fairly simple examples of tail recursion, and all work fine with my TCO. However, some issues exist that I haven't tracked down: running *runtests.sh* results in only 21/326 passed tests (most of the rest result in Segmentation Faults which I'm not sure the cause of) probably due to some incorrect assumptions I've made or failure to properly increment or decrement reference counters. I have, however, created tests that show that multiple tail recursive calls can be placed as arguments to another call without corrupting the stack or anything of that nature. So although it clearly has issues with some code, for most simple purposes it seems to run fine. It's likely that the issue occurs when a C function is invoked as a tail call, but I haven't had a chance to fully investigate the problem.

In order to determine the relative efficiency of my TCO, I ran the above recursive and iterative functions several times:

	iter	rec	tail
pow(2, 900)	1.46	2.44	1.97
fact(900)	3.03	4.19	3.45
foldl (fact 900)	5.72	21.6	6.56
reverse(range(900))	12.75	42.71	13.24
enum(900)	5.79	19.8	6.42

comparison of run time of iteration, non-optimized, and optimized tail recursion

As you can see by the times for the different styles, non-optimized tail recursion takes up to almost 4 times as long to perform as the equivalent iteration in some cases, and never out-performs it. While the TCO function takes less than $1 \frac{1}{2}$ times the time the iteration took in every instance. Just looking at the numbers, its clear that the TCO functions perform very similarly to the iteration since it does not suffer the blow up in time due to the creation of stack frames. (Note: part of the huge difference in performance between the non-TCO functions and the iterative solutions is that the functions are so small, that the creation of the stack frames likely takes up most of the time the program is working, and that non-recursive tail calls would perform more similarly to non-TCO than the TCO).

Unfortunately, digging through the Python code was a lot tougher and more time consuming than I had planned, so I didn't accomplish everything I had hoped for. The TCO I implemented was always enabled so even if you didn't want to perform the TCO, if you compiled the code to byte code, it would perform the TCO. I had planned to enable it with a flag, either another level of optimization or a flag such as '-TCO' when Python is invoked so that users wouldn't be forced into the optimization if for some reason if they felt, as it seems some members of the Python community do, that the TCO was counter-productive. Of course, I had planned to implement the non-recursive tail call optimization as discussed earlier; probably by creating a new frame object and replacing the current one with it. Unfortunately, there are multiple ways to call functions, and types of arguments, necessitating several different opcodes for calling functions: *CALL_FUNCTION*, *CALL_FUNCTION_VAR*, *CALL_FUNCTION_KW*, and *CALL_FUNCTION_VAR_KW* which deal with variable arguments and keyword arguments. I only implemented the tail call equivalent of the first, *CALL_FUNCTION*, but in order to fully support all styles of tail calls, each function call opcode must have a defined tail call counter-part. It would be a little bit more tricky to implement a tail call counter-part to the other types of function calls just because the argument structure is more complex. However, it should still be relatively simple if one has a good understanding of how the variable and keyword arguments work. The biggest argument against support for TCO in Python I have seen has been the fact that if tail recursive function were to throw an exception, there would not be a good traceback to follow. Although I personally think this is a ridiculous excuse, a traceback for tail calls (that must be explicitly enabled) would be a nice feature to add. Especially if it helps silence the opponents of implementing TCO in Python. I never bothered to look into the normal traceback code, but I believe that the tail call traceback can be kept track of by either a separate stack which is deleted once the chain of tail calls return or by using the normal traceback interface.

Unfortunately, although I had planned to get a lot more accomplished this semester, implementing the TCO in Python was a lot trickier than I had imagined. The core of Python isn't very well documented, and I had a difficult time following the chain of function calls that occurred every time a function is called in the Python code. However, it seems that TCO in Python is just not nearly as simple as it is in other languages. The running environment is complex, and does not naturally lend itself to the TCO. It maybe that I wasn't trying the right strategy to implement the optimization, and something like trampolining would have been a better strategy; however, for the tail recursion, besides somehow detecting the tail call before the byte code is generated and making some clever optimizations, it'd be hard to beat the speed of my solution simply because it involves very little overhead: just storing arguments back into place off the stack. As for the non-recursive tail calls, since I had a hard time following everything that happened during a function call, I can't say whether just creating a new frame object and replacing the current one with the new would be a relatively cheap operation or would perform comparable to unoptimized recursion (apart from the recursion limit and stack space benefits). However, as I have shown, the tail-recursion optimization can be simple and efficient in Python because I was able to implement it as an automatic part of the byte code compiler and because its performance was comparable to that of the equivalent iterative solution. Some work would need to still be done to made it feel as a natural part of Python: it would have to only be invoked by a special flag, work with every different type of function call, and probably provide some sort of traceback if a tail recursive function fails. However, I feel that I have met an important part of my goal in that I've shown that there is a relatively simple solution to implementing the most common kind of tail call (a recursive one) and that it does not only provide a good performance increase over the unoptimized recursive solution (and also in my opinion, more concise and often easier to read), but it also overcomes the restrictions on recursive functions inherit to Python.