Daniel Cardenas
Comp333 T/Th

Tail Recursion Vs. Non-Tail Recursion

To understand Tail Recursion one must first understand what recursion is. Recursion can be defined as the process of a function calling itself. On the face of it, that may seem redundant and apt to go on forever. This is true, unless certain cares are made to prevent such unintended consequences. A recursive function is best utilized when a large task can be broken into smaller, more repetitive sub-tasks [ProgrammerInterview Recursion]. In order to prevent the function from calling itself indefinitely, a Base Case is used so that when a condition is met the recursion stops and the intended output or process is returned. Otherwise, the recursion will continue with the Recursive Case. Figure 1 (in the Appendix) provides an example of this format in the Java programming language (an imperative language). While recursive functions may often be alternatively done iteratively, it benefits from a more readable code and algorithmic representation. Unfortunately, it also can suffer from a fatal flaw: Stack Overflow. Essentially, Stack Overflow is a consequence of the Call Stack (a data structure used by the program to store information about the active functions [PorgrammerInterview Recursion]) becoming "overfilled". What this means is that because a recursive function needs to "pause" (in order for the recursively called function to be carried out) all the variables for that specific invocation of the function needs to be stored in memory, specifically, in the Call Stack. Each invocation of a recursively called function takes up a space in the Call Stack as a Stack Frame. Figure 2 provides an illustration of this process. As subsequent invocations occur, more Stack Frames are created and more memory is taken up in the Call Stack, until, eventually the Call Stack may reach a maximum of allowed memory and then give a Stack Overflow Error. Enter Tail Call Optimization.

Tail Call Optimization (TCO) sought to eliminate the issue of Stack Overflow. It was found that under certain recursive conditions, the compiler could be allowed to use just one Stack Frame in the Call Stack, as opposed to creating a Stack Frame for every single invocation of the recursively called function. In order to implement this, a specific type of recursion is required: Tail Recursion. Tail Recursion can be defined simply as a function that returns only a call to itself. This means that the final result of the recursive call is also the final result of the calling function itself. The final function call (the tail-end call) actually holds the final result [ProgrammerInterview Tail Recursion]. To illustrate this, let us first look at examples of non-tail recursive functions.

In Figure 3, three examples of a return from a non-tail recursive function are shown in the Lua programming language (a multi-paradigm language). From top first, we see that the first example function must first perform an addition with the result of the recursively called function before it can return anything. In the second and third examples, it must adjust to a 1 result before it can return [Lerusalimschy 2003]. In needing to perform such operations prior to the return, it forces the compiler to create a Stack Frame in the Call Stack for the calling function, since its data and variables will be needed in order to complete the return. In other words, the calling function is required to "stay around" waiting for its recursive call to complete. Now let us look at an example of a Tail Recursive function.

In Figure 4, we see an example of a tail recursive function in the Python programming language (an imperative language). In both return instances, only the result of the recursively called function is returned, even when there are parameters (As an aside, it is interesting to note that in the Lua programming language the recursive function calls can be complex expressions since Lua evaluates them before the function call [Lerusalimschy 2003]. This is a consequence

of how the Lua language has been constructed, which allows the appropriate precedence to be performed in order to carry out the evaluations prior to function calls.). As a consequence of the tail recursion format, subsequent recursive calls no longer need the variables or data from prior function calls. Thus, by writing the function out in as tail recursive, the compiler is allowed to disregard all data contained within the prior function call and only the arguments passed on the subsequent recursive function calls are utilized. By contrast, a non-tail recursive function, say function A, is required to remain open with all data in function A remaining in memory as a Stack Frame in the Call Stack. Only when the recursively called functions have completed is the "waiting" function A allowed to close and its Stack Frame cleared from the Call Stack. Again, this is because function A contains a variable necessary for 'return' to be executed. With tail recursion, the calling function does not need to stay around. It has done everything it needs to do, and since it is returning another functions output there is no longer a need for its variables or data. This provides an opportunity for the language to optimize the stack and memory, allowing more room for more calls [ProgrammerInterview Tail Recursion]. With that said, not all languages choose to utilize Tail Recursion in such a manner.

Tail Call Optimization (TCO) is the process by which a tail recursive function call is optimized to use just one stack frame. A language that chooses not to implement TCO may still allow for the coder to write a function in a tail recursive manner, however, it will not benefit from any performance or memory optimization. In other words, a function can be written as tail recursive but will still behave like a normal recursive call (non-tail recursive). TCO is a feature that must be built in as part of the compiler to a language. Interestingly, many imperative languages, such as Python and Java, have chosen not to utilize TCO. On the other hand, function languages such as F# and LISP require it. TCO is a necessity for functional languages due to

their reliance on function calls. The advantages and disadvantages of TCO are to be considered when choosing whether to allow a language (and its compiler) to implement it.

Different language paradigms may view tail recursion or TCO differently. In the case of functional languages such as F#, imperative looping structures are discouraged. Instead, functions are heavily relied upon. As explained previously, this introduces the issue of an unrecoverable type of CLR error: StackOverflow. Thus, TCO prevents recursive use of functions from using up the valuable and limited resource that is the Call Stack. By avoiding excessive use of the stack space, recursion can be performed safely and, as a consequence of TCO, tail recursion can be as efficient as iteration [Smith 2008]. Even when a tail recursive call is looped infinitely, there are no Call Stack resources being used and so it can carry on indefinitely, where as a non-tail recursive function would eventually reach a StackOverflow error. Function languages are not alone in using TCO. Even some multi-paradigm languages, such as Lua, support tail call optimization. As such it treats the tail call as a GoTO with arguments.

In contrast to the functional paradigms, imperative programming languages rely on iterative loops. Though the benefits of TCO have been previously laid out, a number of disadvantages have been identified, thus resulting in many imperative languages choosing not to implement it. For one, if the tail call is not used in a truly recursive manner, there may be no faster algorithmic complexity to the code. This means that the recursively formatted function is no more effective or faster than an iterative version. As such there is no benefit to the "optimization" of tail recursive functions [Rossum 2009]. Secondly, debugging can be made more difficult due the loss of variable data from disregarded calling functions. The call stack is like a journal. It tells us where we've been and how we got there. Part of debugging requires us

to “retrace” our steps whenever a program does not run as it is expected to. When walking through a program’s execution, we need to be able to see how and why the program changed its states and ended up where we did not want it to go. In other words, we are looking for “wrong turns”. Look at Figure 5 as an example. If we end up in the debugger inside some_call(), the data (such as x and y) from the calling function is lost due to its stack frame being replaced or overridden [Rossum 2009]. This would make it very difficult to track down the root of our erroneous branching. Part of the issue is that such optimization assumes that the calling function has performed without error and thus may be forgotten. This is why Python does not support TCO [ProgrammerInterview Tail Call Optimization]. Thus, code that uses tail recursion will behave like a normal recursive call and use multiple stack frames. This is because Python is built more around iteration than recursion [Penner 2016]. Along with Python, many other imperative or multi-paradigm languages, such as Java and C, continue to not support tail call optimization overtly. Although, some compilers may still choose to use it even though it is not a part of the language specification [GeeksForGeeks Tail Recursion]. Different approaches can be used to avoid the limitations of StackOverflow. In the example of Python, a decorator can be used to get around that problem by continually entering and exiting a single function call so that it is not technically a recursive function anymore [Penner 2016].

Every language designer has to weight the benefits and drawbacks of tail call optimization. This can mean trade-offs, such as whether to free up the Call Stack or keep it filled as a debugging measure. Sometimes the paradigm, such as iterative or functional, may necessitate or prioritize the choice of implementing TCO. An iterative language may want to retain its iterative approach, while a functional language may require TCO to retain safe

recursive function calls. Ultimately, it may be a fine feature for some languages, but not a fit for others.

Appendix

```
public int factorial (int x)
{
  if (x == 1)
  {
    return 1;  //base case
  }

  else  /*recursive case*/
   return factorial(x-1) * x;
}
```

Figure 1 Java non-tail recursive example [ProgrammerInterview Tail Recursion]

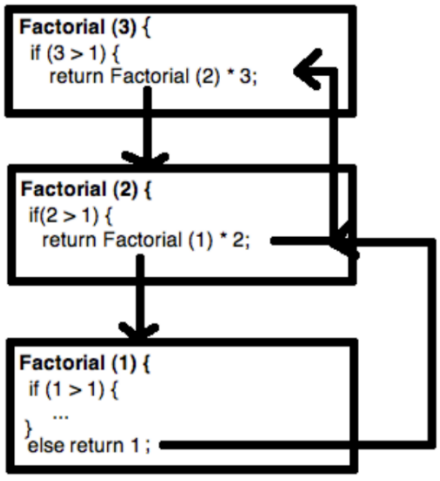

Figure 2 Recursive Function Calls in Call Stack [ProgrammerInterview Recursion]

Appendix (continued)

```
return g(x) + 1
return x or g(x)
return (g(x))
```

Figure 3 Lua non-tail calls examples [Lerusalimschy 2003]

```
def factorial(i, current_factorial=1):
  if i == 1:
    return current_factorial
  else:
    return factorial(i - 1, current_factorial * i)
```

Figure 4 Python tail recursive example [Penner 2016]

```
if x > y:
 return some_call(z)
else:
 return 42
```

Figure 5 Python debugging tail recursive example [Rossum 2009]

References

GEEKSFORGEEKS. *Tail Recursion*. https://www.geeksforgeeks.org/tail-recursion/

LERUSALIMSCHY, R. 2003. 6.3 – Proper Tail Calls, *Programming in Lua*. http://www.lua.org/pil/6.3.htm.

PENNER, C. 2016. *Tail Recursion in Python*. https://chrispenner.ca/posts/python-tail-recursion

PROGRAMMERINTERVIEW.COM. *Provide an explanation of recursion, including an example*. https://www.programmerinterview.com/index.php/recursion/explanation-of-recursion/

PROGRAMMERINTERVIEW.COM. *What is Tail Call Optimization? What is tail call elimination? Provide an example of tail call optimization*. https://www.programmerinterview.com/index.php/recursion/tail-call-opimization

PROGRAMMERINTERVIEW.COM. *What is Tail Recursion? Provide an example and a simple explanation*. https://www.programmerinterview.com/index.php/recursion/tail-recursion/

ROSSUM, G.V. 2009. Final Words on Tail Calls, *Neopythonic*. http://neopythonic.blogspot.com/2009/04/final-words-on-tail-calls.html

SMITH, C. 2008. *Understanding Tail Recursion*. https://blogs.msdn.microsoft.com/chrsmith/2008/08/07/understanding-tail-recursion/