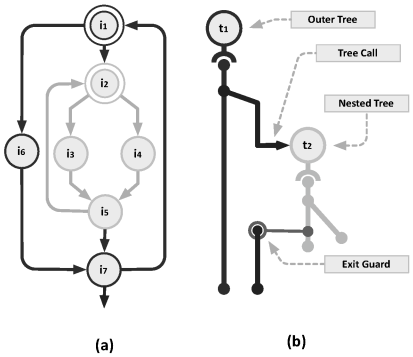



**Figure 7.** Control flow graph of a nested loop with an if statement inside the inner most loop (a). An inner tree captures the inner loop, and is nested inside an outer tree which "calls" the inner tree. The inner tree returns to the outer tree once it exits along its loop condition guard (b).

In general, if loops are nested to depth $k$, and each loop has $n$ paths (on geometric average), this naïve strategy yields $O(n^k)$ traces, which can easily fill the trace cache.

In order to execute programs with nested loops efficiently, a tracing system needs a technique for covering the nested loops with native code without exponential trace duplication.

### 4.1 Nesting Algorithm

The key insight is that if each loop is represented by its own trace tree, the code for each loop can be contained only in its own tree, and outer loop paths will not be duplicated. Another key fact is that we are not tracing arbitrary bytecodes that might have irreduceable control flow graphs, but rather bytecodes produced by a compiler for a language with structured control flow. Thus, given two loop edges, the system can easily determine whether they are nested and which is the inner loop. Using this knowledge, the system can compile inner and outer loops separately, and make the outer loop's traces *call* the inner loop's trace tree.

The algorithm for building nested trace trees is as follows. We start tracing at loop headers exactly as in the basic tracing system. When we exit a loop (detected by comparing the interpreter PC with the range given by the loop edge), we stop the trace. The key step of the algorithm occurs when we are recording a trace for loop $L_R$ ($R$ for loop being recorded) and we reach the header of a different loop $L_O$ ($O$ for other loop). Note that $L_O$ must be an inner loop of $L_R$ because we stop the trace when we exit a loop.

- If $L_O$ has a type-matching compiled trace tree, we call $L_O$ as a nested trace tree. If the call succeeds, then we record the call in the trace for $L_R$. On future executions, the trace for $L_R$ will call the inner trace directly.
- If $L_O$ does not have a type-matching compiled trace tree yet, we have to obtain it before we are able to proceed. In order to do this, we simply abort recording the first trace. The trace monitor will see the inner loop header, and will immediately start recording the inner loop. [2]

If all the loops in a nest are type-stable, then loop nesting creates no duplication. Otherwise, if loops are nested to a depth $k$, and each loop is entered with $m$ different type maps (on geometric average), then we compile $O(m^k)$ copies of the innermost loop. As long as $m$ is close to 1, the resulting trace trees will be tractable.

[2] Instead of aborting the outer recording, we could principally merely suspend the recording, but that would require the implementation to be able to record several traces simultaneously, complicating the implementation, while saving only a few iterations in the interpreter.

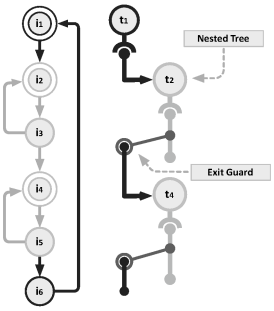



**Figure 8.** Control flow graph of a loop with two nested loops (left) and its nested trace tree configuration (right). The outer tree calls the two inner nested trace trees and places guards at their side exit locations.

An important detail is that the call to the inner trace tree must act like a function call site: it must return to the same point every time. The goal of nesting is to make inner and outer loops independent; thus when the inner tree is called, it must exit to the same point in the outer tree every time with the same type map. Because we cannot actually guarantee this property, we must guard on it after the call, and side exit if the property does not hold. A common reason for the inner tree not to return to the same point would be if the inner tree took a new side exit for which it had never compiled a trace. At this point, the interpreter PC is in the inner tree, so we cannot continue recording or executing the outer tree. If this happens during recording, we abort the outer trace, to give the inner tree a chance to finish growing. A future execution of the outer tree would then be able to properly finish and record a call to the inner tree. If an inner tree side exit happens during execution of a compiled trace for the outer tree, we simply exit the outer trace and start recording a new branch in the inner tree.

### 4.2 Blacklisting with Nesting

The blacklisting algorithm needs modification to work well with nesting. The problem is that outer loop traces often abort during startup (because the inner tree is not available or takes a side exit), which would lead to their being quickly blacklisted by the basic algorithm.

The key observation is that when an outer trace aborts because the inner tree is not ready, this is probably a temporary condition. Thus, we should not count such aborts toward blacklisting as long as we are able to build up more traces for the inner tree.

In our implementation, when an outer tree aborts on the inner tree, we increment the outer tree's blacklist counter as usual and back off on compiling it. When the inner tree finishes a trace, we decrement the blacklist counter on the outer loop, "forgiving" the outer loop for aborting previously. We also undo the backoff so that the outer tree can start immediately trying to compile the next time we reach it.

## 5. Trace Tree Optimization

This section explains how a recorded trace is translated to an optimized machine code trace. The trace compilation subsystem, NANOJIT, is separate from the VM and can be used for other applications.