Inlining: A Tool for Eliminating Mutual Recursion

OWEN KASER, C. R. RAMAKRISHNAN, and SHAUNAK PAWAGI
State University of New York, Stony Brook

Procedure inlining can be used to convert mutual recursion to direct recursion. We present a set of conditions under which inlining can transform all mutual recursion to direct recursion. This will result in fewer procedure calls, and will allow use of optimization techniques that are most easily applied to directly recursive procedures. Conditions are also provided which answer the question, “when can mutual recursion elimination not terminate?”. Also, a technique is presented to eliminate a mutually recursive circuit if it consists of only tail calls.

Categories and Subject Descriptors: **D.3.4 [Programming Languages]:** Processors—*compilers, optimization*
General Terms: Languages, Performance
Additional Key Words and Phrases: Inline substitution, procedure inlining, call graphs, mutual recursion, theory

1 INTRODUCTION

Inline expansion of a procedure call is the replacement of the call by the code of the called procedure. As an optimization, inlining can reduce execution time for several reasons. First, it reduces the number of call and return instructions executed. Second, inlining produces larger blocks of code. This often benefits other optimization techniques, such as constant propagation[1], especially if inter-procedural optimizations are not attempted. Also, inlining can increase the locality of code, resulting in an improved instruction-cache access pattern[5, 9], and improved paging[4].

A contribution of this paper is a set of conditions under which inlining can transform all mutual recursion to direct recursion, resulting in fewer procedure calls. For instance, converting two mutually recursive procedures to one self-recursive procedure may halve the number of calls. Program locality is also improved, and further compiler optimizations may be possible that would be more difficult, or impossible, otherwise. For instance, inlining may remove all mutually recursive calls to a procedure, leaving only non-recursive or directly recursive calls. In this case, inter-procedural analysis is much simplified, and certain optimizations that can be performed only on directly recursive calls (or nonrecursive procedures, c.f. [12, 13]) may be performed. One example of this is the standard technique (with an explicit stack data structure and the use of local *goto* statements) for simulating recursion in nonrecursive languages[1].

Another example involves tail calls, and is examined later. We present a technique to eliminate a mutually recursive circuit if it consists of only tail calls.

For a second contribution, we consider whether heuristics for mutual-recursion elimination can continue forever. We show that this can happen, even if certain reasonable restrictions are placed on which arcs can be inlined. In fact, we provide a precise characterization of the programs for which such undesirable behavior is possible.

2 BACKGROUND

First, we describe our definitions and notation. Then we describe our model of inlining, and finally we consider the two different varieties of inlining we use.

2.1 Definitions

In this paper, we assume standard graph-theoretic definitions for directed graphs and multi-graphs, directed trees, nodes, arcs, circuits, leaves, and so forth. An arc from parent u to child v is denoted[2] by (u, v). It is an in-arc for v and an out-arc for u. If u and v are identical, this arc is a *self-loop*. In our multi-graphs, a circuit is viewed as a sequence of arcs. Thus, the same set of nodes may correspond to several circuits, if there are parallel arcs. A circuit is *simple* if it does not contain some node multiple times. Unless otherwise specified, all circuits are assumed to be simple. Furthermore, a circuit of length two is called a *2-circuit*. Without circuits, a directed (multi-)graph is a *(multi-)dag*. Its *roots* are nodes without in-arcs, and its *leaves* are nodes without out-arcs. Note that a dag may have several roots.

Rather than consider an actual program when deciding what to inline, we wish to make our decision based on a more compact input. For this reason, we use the call graph of the program. This is a directed multi-graph that has a node for each procedure and an arc for each call. For an arc $a = (\mathtt{q}, \mathtt{r})$, we refer to procedure `q` as the *caller* and to procedure `r` as the *callee*. Observe that a self-loop in the call graph indicates a directly recursive call, also known as a self-recursion, in the program. Note that the call graph may be a multi-graph, since `q` may contain several calls to `r`. Thus, we often refer to a specific *call site* within the code of `q`. For every call site there is an arc in the call graph, and vice versa. Note that we may choose to inline an arc in the call graph, but choose not to inline an arc parallel to it.

[1] We do not claim that this technique cannot be modified to handle mutual recursion.

[2] Since we deal with multi-graphs, this notation is somewhat ambiguous. A more precise notation, differentiating parallel arcs (u, v), is not required in this paper.

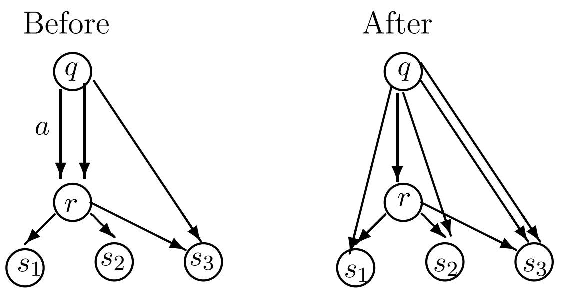

Figure 1: A call graph before and after inlining arc a.

Notation. Throughout this paper we adhere to a naming scheme. We use `q`, `r`, `s`, `f`, `g`, and `h` to represent procedures. Programs are named P, Q, and R, and the original program is P. The size of a program is σ. For call graphs, v, u, and w are nodes, and a, b, and c are arcs. C is a circuit in a call graph, and p is a path.

Inlining. Inlining `r` into `q` is the process of inserting `r`'s code into `q`, at some particular call site in `q` where it calls `r`. Refer to Figure 1. The call to `r` is then removed, and various small changes (e.g., simulating parameter passing) are made. To update the call graph after inlining, note that inlining will introduce call sites into `q`, for every call site in `r`. Thus, for every arc leaving `r` in the call graph, a new arc is created leaving `q`. Next, the arc representing the inlined call is removed. Finally, the other data in the call graph must be updated: the size of `q` is incremented by the size of `r`.

Inlining Model. Next, we give several assumptions we have made about inlining, and briefly discuss each. The first assumption is that procedures cannot be removed, even when no calls are made to them. Also, we assume that interactions between inlining and other optimizations can be ignored. Finally, we assume the entire program is available, and that all mutually recursive calls can be detected by examining the call graph.

If all calls to a procedure have been inlined, the procedure itself can sometimes be removed, resulting in reduced code size. We do not consider this possibility, since in an interactive system like EQUALS[7], every procedure may be called directly by the user. Moreover, the variant of inlining discussed in Section 2.2 may create a new call to a procedure that currently has none. Even when inlining in C, Hwu and Chang [2, 6] are only able to remove procedures under very limited conditions; for instance, if the procedure is declared to be static, and its address has not been made available to others.

As another simplification, we usually assume that the inlining process does not optimize tail recursions that may be created. We are primarily interested in improvement obtained by inlining, and not with its interactions with other optimizations. Despite this, in Section 3.1 we specifically consider tail-recursion removal in the context of inlining.

Finally, we assume that if the call graph of a program contains no circuits (except possibly self-loops), then there is no mutual recursion. This is not necessarily the case if all call sites are not explicitly provided. For instance, it is possible to create mutual recursion by calling through a pointer, in C. Alternatively, if the entire call graph of the program is not provided, it is possible that a separately-compiled module may contain procedures inducing mutual recursion.

2.2 Current- versus Original-Version Inlining

We next discuss two different methods of inlining, stating results that have been shown in [8]. There, we conclude that *original-version inlining*, or *ov-inlining*, is superior to the usual method, called *cv*-inlining, where the current versions are inlined.

Suppose a call to procedure `r` is inlined into procedure `q`, and then a call to `q` is inlined into procedure `s`. Which code is substituted in place of the call to `q`— the original code for `q` or the most recent code for `q`? As inlining progresses, there may be several versions of `q`, and any of them can be chosen: they are all functionally identical[3]. Previous approaches [1, 2, 3, 6, 9, 11] always use the current version (most recent). In [8], though, we show that the best policy is always to substitute the original version. *Ov*-inlining can obtain any program that *cv*-inlining can, and it can also obtain any program obtained by intermediate-version inlining. We also showed there are programs that can be obtained by *ov*-inlining, but not by *cv*-inlining. Moreover, some of these programs are superior to *any* program obtainable by *cv*-inlining.

Finally, we consider the changes made to the call graph after either form of inlining. For *cv*-inlining, a multi-arc is effectively collapsed, bringing copies of the callee's out-arcs into the caller. *Ov*-inlining is slightly more complicated. When the original code of a procedure is inserted, the arc is collapsed and copies of the callee's *original* out-arcs are given to the caller. Note that the current out-arcs may be much different from the original ones.

3 ELIMINATING MUTUAL RECURSION

To eliminate mutual recursion by transformation to self-recursion, we classify arcs as *legal* or *illegal*. An arc is illegal if it is to a node that has a self-loop. Inlining an illegal arc will not result in any benefit, because it creates a new arc with the same endpoints as the one being inlined. Thus, the resulting call graph contains the original call graph as a subgraph, and there

[3] One of the desirable properties of inlining is that it does not affect the functional behavior of any procedure.

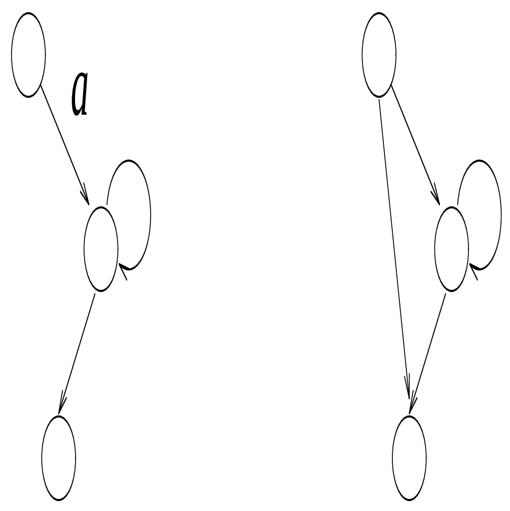

Figure 2: Arc a is illegal for *cv*-inlining in the above call graph.
Left: Before inlining. Right: After inlining. Note that a new arc has replaced a.

is no benefit in having more arcs. (See Figure 2.) An arc is legal for *cv*-inlining only if it is to a node with no self-loops. For *ov*-inlining, legal and illegal arcs can be defined similarly, but the distinction is not required for this paper. A sequence of (legal) inlinings that results in a call graph with no mutual recursion is called a *mutual-recursion elimination sequence*.

It turns out that we cannot always eliminate all mutual recursions, whether *cv*-inlining or *ov*-inlining is used. The following two questions raise interesting issues that are the main focus of this paper.

Question 1 *When can we eliminate* all *mutual recursion?*

Question 2 *Is there a call graph for which we can keep doing legal inlinings indefinitely?*

For the first question we can precisely describe conditions on call graphs. This also yields an algorithm for mutual recursion elimination. We shall answer the second question later. First, we provide two lemmas describing when any version of inlining *cannot* remove all mutual recursion.

Consider a path p which is $v(= v_0), v_1, v_2, \ldots, v_l(= u_0)$ (in the original call graph) from a node v to a circuit[4] $C = u_0, u_1, \ldots, u_m$. Figure 3 illustrates this structure, used in the following lemma, proved in the Appendix.

Lemma 1 *After any number of inlinings of any version, each node v_i, $0 \le i \le l-1$, has an out-arc either to a node originally on C or to v_j, where $j > i$. Furthermore, after any number of inlinings, every node $u_i, 0 \le i \le m$ has an out-arc to some node $u_j, 0 \le j \le m$.*

[4] Only the final node of p may be a node of C.

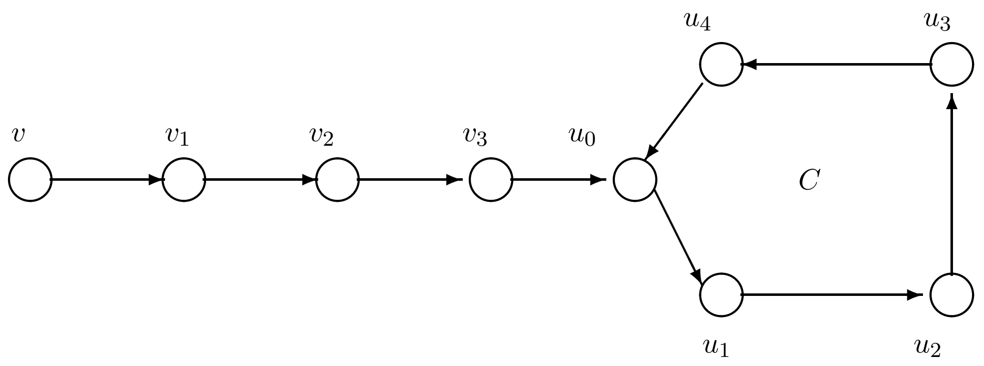

Figure 3: A path to a circuit.

Corollary *After any sequence of inlinings, there is a path from any node that could originally reach C, to at least one of the nodes originally on C.*

Recall that the strongly connected components (SCCs) of a call graph identify groups of mutually recursive procedures. Now suppose that we have two circuits C_1 and C_2 that are node disjoint but are in the same SCC. Thus, every node of C_1 has a path to C_2, and vice versa. This can be used to demonstrate that after any sequence of inlining, a circuit is always present in the call graph, implying the presence of mutual recursion.

Lemma 2 *If the call graph contains two node-disjoint circuits in an SCC, then there is no mutual-recursion elimination sequence, regardless of whether cv- or ov-inlining is performed.*

Proof: Let C_1 and C_2 be the two disjoint circuits of the SCC. Suppose that any sequence of inlining is performed. Then by the corollary to Lemma 1, any node in C_1 can reach one of the nodes in C_2, and vice versa. Consider any node u in C_1. From u we can reach some node v in C_2. (Since C_1 and C_2 are disjoint, $v \neq u$) From v we can reach another node, say w, in C_1. (Again, $v \neq w$.) If $w = u$, we have a circuit. If not, a new node in C_2 can next be reached, and so forth. When we reach an already-visited node, we complete a circuit. The process of finding unvisited nodes cannot be repeated indefinitely, since C_1 and C_2 are finite. $\Box$

Finally, we arrive at a theorem that precisely identifies those call graphs for which inlining can remove all mutual recursion. It is valid for both *ov*- and *cv*-inlining. It also assumes that a self-loop is a circuit.

Theorem 1 *For any call graph, there exists a mutual-recursion elimination sequence iff no SCC contains two node-disjoint circuits.*

Proof: **(If:)** First note that no inlining can produce a circuit involving nodes in different SCC's. Therefore, consider some SCC. Since its circuits are not node disjoint, there exists a shared node v that is on all circuits. Note that when v is deleted from the SCC, we obtain a dag. (Otherwise there was a circuit that did not have v on it.) Consider a *cv*-inlining sequence that completely inlines that dag, bottom-up. If this inlining sequence is used on the original SCC (before the deletion of v), observe that we will not obtain any self-loops. The only arcs introduced are in-arcs of v. Thus, after this phase has been completed, all arcs are either in-arcs or out-arcs for v. Now we can inline all out-arcs of v that are not self-loops. Any new arc added in this phase is a self-loop on v. Note that we have eliminated all arcs, except some in-arcs of v. The result has no mutual recursion because no other node has any in-arcs.

Since this result was obtained by *cv*-inlining, the same result can be obtained by *ov*-inlining.

Only If. If the condition is false, then there is an SCC with two node-disjoint circuits. Thus, by Lemma 2 no amount of inlining within that SCC can remove all mutual recursion. □

This concludes our discussion for Question 1. Next we address Question 2, which asked if an inlining program can keep inlining forever. Now, recall that, for *cv*-inlining, it is not legal to inline an arc to a node with a self-loop. This restriction arises since inlining such an arc can clearly be repeated forever. Likewise, we might suspect that legal arcs, in the case of *ov*-inlining, could be defined as arcs to nodes that do not *initially* have self-loops. Nonetheless, even if the nodes being inlined did not originally have self-loops, we can keep inlining forever. For instance, consider the following program, where there is no initial self-recursion.

```
f = .... g()....;                    g = .... f()....
```

We can inline the call to `g` into `f`, to get

```
f = .........f().....                g = .... f()....
```

and then inline the recursive call, to get

```
f = .............g().....            g = .... f()....
```

Clearly this process can be repeated forever. (This case is uninteresting and is included for completeness.) The interesting case involves *cv*-inlining and is considered next.

The answer to Question 2 is still *yes*, if only legal *cv*-inlining is permitted. We say that a program has an *infinite inlining sequence* if there is some way to apply any number of inlining operations to it. If the operations are all legal *cv*-inlinings, we say it has an infinite *cv*-inlining sequence. An easy case, where the call graph is a dag, is considered in the next lemma, stated without proof.

Lemma 3 *If the call graph is a multi-dag, there is no infinite cv-inlining sequence.*

Let an *unlooped* circuit be a simple circuit on which none of the nodes has a self-loop. The necessary and sufficient conditions for the existence of an infinite *cv*-inlining sequence, for a general call graph, are established by the following theorem. Its proof is in the Appendix.

Theorem 2 *A program has an infinite cv-inlining sequence iff either of the two following conditions hold:*

1. *Its call graph has an unlooped circuit of length three or more.*
2. *Its call graph has an unlooped circuit of length two, and there is a third node with an arc to either of the two nodes of the circuit.*

Before we proceed with the elimination of mutual recursion, we must test the conditions the theorem. If these are met, we avoid a non-terminating sequence by keeping *generation counts* for new arcs. (Without this several of our most natural heuristics actually did fail.) This count is zero for all original arcs. Each time an arc of generation count i is inlined, any new arcs created have a generation count of $i + 1$. We limit the maximum allowable generation count to a constant. This prevents us from inlining forever, even if no code-size limit is given. We note that it is easy to efficiently test for the conditions of the theorem.

This concludes the discussion of Question 2. The next section shows that the answer to Question 1 changes, if tail-recursion optimization is done while inlining progresses. In this case we can always eliminate all mutual recursion that involves only tail-calls.

3.1 Creating Tail Recursions

Next, we consider an important interaction between mutual-recursion elimination and the tail-recursion optimization performed by many compilers. Recall that a tail call is a procedure call that is performed as the final activity of the calling procedure. The caller makes the call and, after the called procedure has completed, immediately returns. A tail-recursive call is thus a directly recursive call that is also a tail call. A frequently performed optimization replaces such tail recursion with loops. Many compilers perform a more general optimization with tail calls that are not directly recursive; such a call can be replaced by instructions that first modify the current activation record and then perform a jump. The new sequence of instructions is usually faster than the construction of a new activation record, and it also reduces the stack-space requirement.

The EQUALS compiler[7] can remove tail recursion, but since it generates C code, it is unable to perform the general tail-call optimization. Despite this, selective inlinings of tail calls can create tail recursions that EQUALS can then optimize. Even if EQUALS optimized general tail calls, it would often be preferable to have new tail recursions, which then can be changed to loops. An optimized tail recursion (a loop) is faster than an optimized sequence of tail calls. Thus, the techniques we develop to expose tail recursions should be valuable to others.

Suppose a tail call is inlined, and the callee itself has a tail call to some procedure. A new tail call is created from the caller to that procedure. If it is a direct tail-recursive call, the compiler can immediately convert it into a loop. The simple algorithm described next can convert all circuits of tail calls (*tail-circuits*) into tail recursions that are then eliminated by the compiler. Note that if all mutually recursive calls are tail calls, then all mutual recursion can always be eliminated, unlike the situation in the previous section[5].

Consider the call graph when only the tail calls in the original call graph are retained. With *cv*-inlining, a new tail call can only be created from an inlined tail call and a tail call from the inlined node. (Informally, this is composing two tail calls.) Therefore, we use the modified call graph: the arcs we omit have no bearing on the final presence or absence of tail circuits. Also, since inlining can never introduce circuits involving nodes that are in different SCC's, we also delete arcs between SCCs from the modified call graph.

Thus, our algorithm processes the SCC's of the modified call graph independently. It first chooses some node u and *cv*-inlines all of its in-arcs, in the modified call graph. Any tail recursions created thereby are immediately removed by the compiler. Now that u has no calls to it, it will never receive any such calls (in-arcs): to create a new in-arc for u, *cv*-inlining would require that u had an in-arc. The algorithm then chooses another node, and the process repeats. Termination is guaranteed, since after each step the number of nodes without in-arcs increases.

4 CONCLUSION

We have examined several different issues that arise when inline expansion is attempted in a heavily recursive environment. We examined the issue of mutual recursion elimination, and

[5] We do not claim that our algorithm finds the *best* method of eliminating tail-circuits, though.

presented necessary and sufficient criteria for

- the existence of some sequence of inlinings that can completely remove mutual recursion.
- the existence of an unending sequence of *cv*-inlinings.

APPENDIX

Next, we prove Lemma 1 and Theorem 2, which are restated below.

A PROOF OF LEMMA 1

After any number of inlinings of any version, each node v_i, $0 \leq i \leq l-1$, has an out-arc either to a node originally on C or to v_j, where $j > i$. Furthermore, after any number of inlinings, every node $u_i, 0 \leq i \leq m$ has an out-arc to some node $u_j, 0 \leq j \leq m$.

Proof: We prove both statements simultaneously by induction on the number of inlining steps. Initially, every node v_i on p, except the last, has an out-arc to its successor (v_{i+1}) along p. Each node of C has an out-arc to its successor on C. Thus, the basis is established, for zero inlining steps. Assume that the lemma holds for k or fewer inlining steps. Consider a sequence of $k+1$ inlinings, and let the last inlining be of arc a, which leaves node w. Inlining will only cause the removal of one arc (that is, a), although it may cause several new arcs to be created. By the inductive hypothesis, after k inlinings the property was true for all nodes. Observe that w is the only node that might not satisfy the property after inlining. Nevertheless, we show that a new arc is always added such that w also satisfies the property. The argument considers three cases, depending on a.

Suppose that a is an arc (v_i, w), where either w is v_j $(j > i)$, or w is a node $u_{j'}$ of C. By the inductive hypothesis, whatever version of w (which corresponds to a procedure) we choose, it satisfies the property. If $w = u_{j'}$, it has an out-arc to some node of C. Otherwise, w has an out-arc to v_k $(k > j)$, or to a node previously on C. After inlining a, v_i will therefore obtain an arc to a suitable node in either case, satisfying the property.

Next, suppose that $a = (u_j, u_k)$: it is an arc between two of the nodes previously on C. Now, any version of u_k has an arc to some node u_l of C, by the inductive hypothesis. After inlining a, u_j will have an arc to u_l and thereby satisfy the property.

Finally, we consider the uninteresting case when none of the above conditions are satisfied. Note that after a has been inlined its absence will not affect the property that is claimed by the

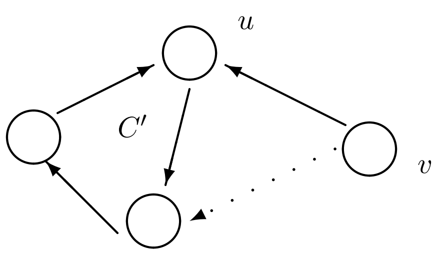

Figure 4: An unlooped circuit with a non-circuit node. The dotted arc is not initially present. It will be created when the arc from v to the circuit is inlined.

lemma. □

B PROOF OF THEOREM 2

A program has an infinite cv-inlining sequence iff either of the two following conditions hold:

1. *Its call graph has an unlooped circuit of length three or more.*
2. *Its call graph has an unlooped circuit of length two, and there is a third node with an arc to either of the two nodes of the circuit.*

Proof: *Sufficiency.* We show that if either condition is met, we can obtain an unlooped circuit C' (of length two or more) and a non-circuit node that has an arc to a node of C'. If condition 2 is met, then C' is simply the required circuit of length two (2-circuit).

Now suppose condition 2 is not met, but condition 1 is. Let C be the unlooped circuit implied by condition 1. Suppose its length is k. Then we can inline any of the nodes, say v, on C into its predecessor, say w. This introduces no self-loops. (If it did, then v must have had an arc to w, forming an unlooped circuit. Furthermore, since C was of length ≥ 3, the predecessor of w was neither v nor w and now has an arc to the unlooped circuit v, w, v. Hence, condition 2 is met, contrary to our assumption.) The required circuit C' is thus the result of this first inlining, and its length is $k - 1$. v is not on C', but it maintains an out-arc to its successor on C'.

Now we show that the circuit C', with an arc to it from non-circuit node v, can yield an infinite *cv*-inlining sequence. We can repeatedly inline circuit nodes into v, as follows. (See Figure 4.) If there are arcs from v to several circuit nodes, choose one of them, say u, arbitrarily. Inline u into v. It is unimportant if v obtains a self-loop. In any case, no node on C' gets any outgoing arcs from this inlining, since only v acquires outgoing arcs. Therefore, no node on C' gets a self-loop. Also, v gets a new arc to the successor of u (along C'). v will next inline this

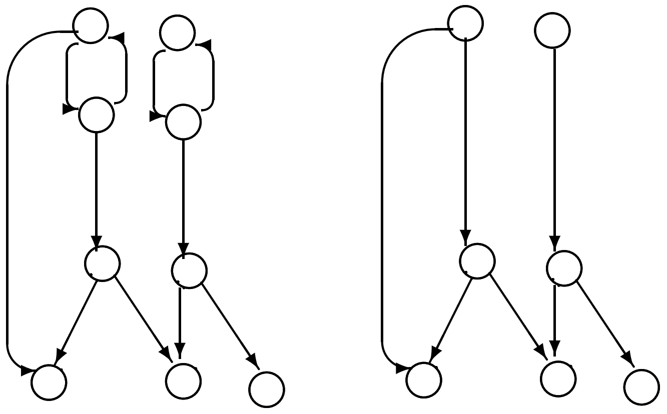

Figure 5: An idag (left) and its corresponding dag

arc. This process of inlining the nodes of C' into v, in their order of appearance on C', can continue indefinitely. Thus, we have shown that the conditions are sufficient.

Necessity. We now show that mutual-recursion elimination must terminate at some point, if the call graph does not meet conditions 1 and 2. Therefore, assume that any circuit that it does have either is of length two, without an outside node pointing to it, or is a circuit of length greater than two, with a node that has a self-loop. Recall that it is illegal to inline a node with a self-loop.

We next show that in-arcs to all such nodes can be deleted without affecting any mutual-recursion sequence. Note that once a node has a self-loop at some stage of mutual-recursion elimination, it will always have the self-loop. Thus, it will *never* be legal to *cv*-inline this node into another, after it acquires a self-loop. Consider a node v that has self-loops initially. For any in-arc (u, v), its copies created during inlining[6] are all in-arcs of v. All such copies of (u, v) are illegal for *cv*-inlining, as are their copies, and so forth.

Hence, we delete all arcs, except self-loops, to nodes that initially have self-loops. This will break all circuits of length three or more, resulting in a structure that we call an *idag.* An idag is essentially a dag with 2-circuits present in some of its roots. See Figure 5 for an example. Also, certain nodes may have self-loops on them, but they are all isolated. The 2-circuits are present only in the roots since we assume that condition 2 has not been met, thereby disallowing in-arcs. We also define the regular (multi-)dag corresponding to an idag, derived by contracting each 2-circuit to a single node and deleting any self-loops present on isolated nodes. Any idag arc that is not on a 2-circuit is called a *dag-arc.*

[6] (u, v) gives rise only to new arcs that are copies of it, and this occurs when u is inlined into some other node.

Next, we show that an idag cannot have an infinite inlining sequence. Assume, for a contradiction, there does exist an infinite inlining sequence. Any arc on any such sequence is either a 2-circuit arc or is a dag-arc. We next explain why there can be only a finite number of 2-circuit arcs. When we inline some 2-circuit node v_1 into its mate v_2, it obtains a self-loop. All in-arcs, whether existing or later created, of v_2 are thereafter illegal for inlining, and thus cannot appear in the remainder of the sequence. Now consider v_1. It may still have in-arcs from (only) v_2, but it has no self-loops. Therefore, each time it is inlined, the number of its in-arcs drops. Since inlining any other arc in the idag cannot affect the number of in-arcs of v_1, the number of times v_1 can be inlined is bounded by the number of in-arcs it originally had. Thus, although an idag may have several roots with 2-circuits in them, the infinite sequence must have fewer 2-circuit arcs than existed before inlining. Since we have only a finite number of 2-circuit arcs, there must exist an infinite sequence of dag-arcs. Hence, by Lemma 3 we obtain the desired contradiction. □

References

[1] John Eugene Ball. *Program Improvement by the Selective Integration of Procedure Calls.* PhD thesis, University of Rochester, 1982.

[2] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 25:349–369, 1992.

[3] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software—Practice and Experience*, 18:775–790, 1988.

[4] Stephen J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. *IEEE Transactions on Software Engineering*, 14:1640–1644, 1988.

[5] Wen-mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings, 16th Annual Symposium on Computer Architecture*, pages 242–251, 1989.

[6] Wen-mei W. Hwu and Pohua P. Chang. Inline function expansion for compiling C programs. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 246–255, 1989.

[7] Owen Kaser, Shaunak Pawagi, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Fast parallel implementations of lazy languages— the EQUALS experience. In *Proc. Conf. on LISP and Functional Programming*, pages 335–344, 1992.

[8] Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. *A New Approach to Inlining.* Technical Report 92/06, SUNY at Stony Brook, 1992.

[9] Scott McFarling. Procedure merging with instruction caches. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 71–79, 1991.

[10] Thomas P. Murtagh. An improved storage management scheme for block structured languages. *ACM Transactions on Programming Languages and Systems*, 13:372–398, 1991.

[11] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, 1977.

[12] Steven S. Skiena. Compiler optimization by detecting recursive subprograms. In *1985 ACM Annual Conference*, pages 403–411, 1985.

[13] Kenneth G. Walter. Recursion analysis for compiler optimization. *Communications of the ACM*, 19:514–516, 1976.