Semantics-Preserving Procedure Extraction [*]

Raghavan Komondoor and Susan Horwitz
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
Electronic mail: {raghavan, horwitz}@cs.wisc.edu

Abstract

Procedure extraction is an important program transformation that can be used to make programs easier to understand and maintain, to facilitate code reuse, and to convert "monolithic" code to modular or object-oriented code. Procedure extraction involves the following steps:

1. The statements to be extracted are identified (by the programmer or by a programming tool).
2. If the statements are not contiguous, they are moved together so that they form a sequence that can be extracted into a procedure, and so that the semantics of the original code is preserved.
3. The statements are extracted into a new procedure, and are replaced with an appropriate call.

This paper addresses step 2: in particular, the conditions under which it is possible to move a set of selected statements together so that they become "extractable", while preserving semantics. Since semantic equivalence is, in general, undecidable, we identify sufficient conditions based on control and data dependences, and define an algorithm that moves the selected statements together when the conditions hold. We also include an outline of a proof that our algorithm is semantics-preserving.

While there has been considerable previous work on procedure extraction, we believe that this is the first paper to provide an algorithm for semantics-preserving procedure extraction given an arbitrary set of selected statements in an arbitrary control-flow graph.

1 Introduction

Legacy code can often be improved by extracting out code fragments to form procedures (and replacing the extracted code with procedure calls). This operation is useful in several contexts:

[*]This work was supported in part by the National Science Foundation under grants CCR-9625656 and CCR-9970707, by the Microsoft Corporation, and by IBM.

- Legacy programs often have monolithic code sequences that intersperse the computations of many different tasks [LS86, RSW96]. Such code becomes easier to understand and to maintain if it is replaced by a sequence of calls (one for each task) [CY79]. This decomposition may also facilitate better code reuse [SJ87, LV97].
- When the same code appears in multiple places, replacing each copy with a procedure call makes the code easier to understand and to maintain (since updates need only be performed on a single copy of the code).
- A code fragment can sometimes be recognized as performing a (conceptual) operation on a (conceptual) object. Making that idea explicit by extracting the fragment into a procedure (or method) can make the code easier to understand, and can be an important part of the process of converting poorly designed, "monolithic" code to modular or object-oriented code [Par72].

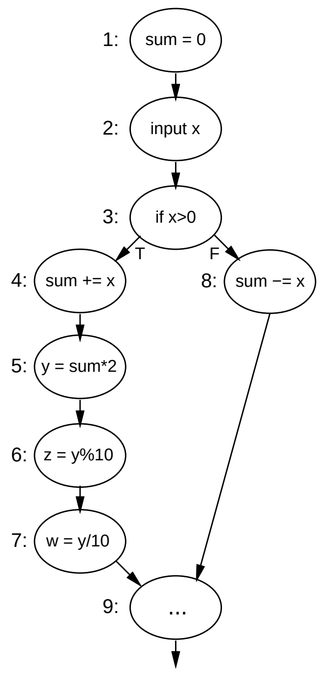
For the purposes of this paper, extracting a procedure is defined by the following three steps:

1. The statements to be extracted are identified.
2. If the statements are not contiguous, they are moved together so that they form a sequence that can be extracted into a procedure, and so that the semantics of the original code is preserved.
3. The statements are extracted into a new procedure, and are replaced with an appropriate call.

Although step 1 is very interesting, it is not the subject of this paper; we assume that the set of statements has been identified, either by the programmer or some kind of restructuring tool, such as those described in [LV97, BG98].

Step 3 involves deciding what the parameters to the procedure should be, which parameters should be passed by reference, and whether the procedure should return a value. These are straightforward issues (e.g., discussed in [LD98]).

Our interest is in step 2; in particular, we investigate the conditions under which it is possible to move a set of selected statements together so that they become "extractable", and so that semantics are preserved. To illustrate that this is a non-trivial problem, consider the following code fragment, represented by a control-flow graph (CFG):



Example 1: Although nodes 3 and 4 are contiguous in the CFG, they cannot be extracted into a procedure because of structural concerns: node 3 has an outgoing edge to node 8, and node 4 has an outgoing edge to node 5; if the two nodes are replaced by a call node (with a single successor in the CFG), either node 8 or node 5 will become unreachable, leading to a malformed CFG.

Example 2: Nodes 2 and 4 cannot be extracted into a procedure because of control dependence concerns. Moving node 4 before the *if* node would cause the wrong value to be assigned to variable *sum* when x is not positive; moving node 2 after the *if* node would cause the wrong value of x to be used to evaluate the condition. In either case, semantics would not be preserved.

Example 3: Nodes 4 and 6 cannot be extracted into a procedure because of data dependence concerns. This situation is similar to the one for nodes 2 and 4, discussed above: Moving node 6 before node 5 causes the wrong value of y to be used in the assignment to z, and moving node 4 after node 5 causes the wrong value of *sum* to be used in the assignment to y.

Example 4: Nodes 5 and 7 *can* be extracted. Node 7 can be moved up before node 6, which makes 5 and 7 an extractable sequence.

The chief contribution of this paper is an algorithm that moves a selected set of CFG nodes together so that they become extractable while preserving semantics. We believe that this algorithm is the first to handle an arbitrary set of selected nodes in an arbitrary (possibly unstructured) CFG. Since semantic equivalence is, in general, undecidable, it is not possible to define an algorithm for this problem that succeeds iff semantics-preserving procedure extraction can be performed for the given set of nodes. Therefore, we identify conditions based on control and data dependence that are *sufficient* to guarantee semantic equivalence; our algorithm succeeds iff those conditions hold. We also include an outline of a proof of correctness for our algorithm (that it performs only semantics-preserving procedure extraction).

A limitation of the algorithm is that it only moves CFG nodes; no duplication is performed. In Example 1, for instance, the indicated extraction can be done if the predicate is duplicated and the code restructured as follows:

```
if x>0  then sum += x
if x>0  then y = sum*2; z = y%10; w = y/10
        else sum -= x
```

Nevertheless, we feel that the algorithm is useful as is; in particular, it can be used as part of an automatic restructuring tool, and in that context failure of the algorithm can provide feedback indicating that code duplication is needed.

The remainder of the paper is organized as follows: Section 2 presents basic assumptions and terminology. Section 3 presents our algorithm for performing step 2 of procedure extraction. Section 4 discusses the algorithm's theoretical complexity, and Section 5 presents experimental results that give some insight into how well the algorithm will perform in practice. Section 6 states two theorems: the first theorem shows that the property we require to preserve control dependence is reasonable, and the second shows that the algorithm is correct (only performs semantics-preserving procedure extraction). Proof outlines for these theorems are included in an appendix; detailed proofs are available in [KH99]. Section 7 discusses related work. Finally, conclusions are presented in Section 8.

2 Assumptions and Terminology

We assume that each procedure in a program is represented by a control-flow graph (CFG) that includes unique Enter and Exit nodes. Other CFG nodes represent predicates or simple statements (assignment, input, output, unconditional branch, or procedure call). A return statement is modeled as an unconditional branch to the Exit node. The Enter node is a special pseudo-predicate; it has two outgoing edges: one labeled "true" to the first statement or predicate in the procedure, and one labeled "false" to the Exit node (these edges are included so that all nodes in the body of the procedure are control dependence descendents of the Enter node). Normal predicate nodes also have two outgoing edges, one labeled "true" and the other labeled "false"; other nodes have one, unlabeled outgoing edge (for the purposes of this paper, there is no need to represent call/return connections among procedures; thus, a call node has one outgoing edge whose target is the statement or predicate that follows the call). Every node is reachable from the Enter node, and the Exit node is reachable from every node. $\mathcal{N}(G)$ denotes the nodes of a CFG G and $\mathcal{E}(G)$ denotes the edges.

We assume that the program includes no uses of uninitialized variables, and no assignments to dead variables. We also assume that appropriate static analyses (e.g., pointer analysis and interprocedural may-use, may-mod analysis) have been done so that the may-use and may-define sets are known for each CFG node (including call nodes).

For the purposes of this paper, two procedures are semantically equivalent iff when they are called in the same state (i.e., with the same mapping of variables – including the special stream variables *input* and *output* – to values), they finish in states that agree on the values of all variables that are (interprocedurally) live at Exit (with *output* considered to be live at all points in the program). A procedure that does not terminate, or that causes an exception – e.g., a division by zero – is considered to finish in the state in which all variables are mapped to $\perp$. Two CFGs are semantically equivalent iff the procedures that they represent are semantically equivalent.

We provide definitions of some standard concepts used in this paper:

Definition(*domination*) : CFG node p dominates node q iff all paths from Enter to q go through p. By definition, no node dominates itself. □

Definition(*postdomination*) : Node p postdominates node q iff all paths from q to Exit go through p. By definition, every node postdominates itself. □

Definition(*control dependence*) : Node p is C-control dependent on node q, where C is either "true" or "false", iff q is a predicate node, p postdominates the C-successor of q but it does *not* postdominate q itself.□

Definition(*flow dependence*) : Node p is flow dependent on node q due to a variable v iff v is used by p and defined by q and there is a CFG path from q to p that includes no node that must define v.□

Definition(*anti dependence*) : Node p is anti dependent on node q due to variable v iff v is used by p and defined by q and there is a CFG path from p to q.□

Definition(*output dependence*) : Node p is output dependent on node q due to variable v iff both nodes define v and there is a CFG path from q to p.□

Definition(*def-order dependence*) : Node p is def-order dependent on node q due to variable v iff both nodes define v, there is a node u that is flow dependent on both p and q due to v, and there is a CFG path from q to p.□

The definitions for flow, anti and output dependences are based on the definitions in [KKP$^+$81], while the definition of def-order dependence is from [BH93]. The following additional terms are also used in the paper:

Definition(*control dependence set*) : The *control dependence set* of node q in CFG G is the set of predicate-node, truth-value pairs (p, C) such that q is C-control dependent on p in G. □

Definition(*hammock*) : A *hammock* is a subgraph of a CFG that has a single *entry node*, and from which control flows to a single *outside exit node*. More formally: A hammock in CFG G is the subgraph of G induced by a set of nodes $H \subseteq \mathcal{N}(G)$ such that:

1. There is a unique entry node e in H such that: $(m \in \mathcal{N}(G) - H) \wedge (n \in H) \wedge ((m,n) \in \mathcal{E}(G)) \Rightarrow (n = e)$.
2. There is a unique outside exit node t in $\mathcal{N}(G) - H$ such that: $(m \in H) \wedge (n \in \mathcal{N}(G) - H) \wedge ((m,n) \in \mathcal{E}(G)) \Rightarrow (n = t)$.

□

Definition(*hammock chain*) : A *hammock chain* $(H_1, H_2, \ldots, H_m)$ in CFG G is a sequence of hammocks with no incoming edges from outside the sequence to any node other than H_1's entry node. That is, $\forall i \in [2 \ldots m]$:

1. the entry node of H_i is the outside exit node of H_{i-1}, and
2. $(n \in H_i) \wedge (m \in \mathcal{N}(G) - H_i) \wedge ((m,n) \in \mathcal{E}(G)) \Rightarrow (m \in H_{i-1})$

□

It can be seen that any hammock chain is itself a hammock. The entry node of the chain is the entry node of the first hammock H_1 and the outside exit node of the chain is the outside exit node of the last hammock H_m.

Definition(*atomic hammock*) : An *atomic hammock* is a hammock that is itself not a chain of smaller hammocks. □

It can be shown that a hammock H with entry node e is atomic iff for each hammock H_1 that is strictly contained in H and also has entry node e, there exists a node n in $(H - H_1)$ such that there is an edge from n to e (see [KH99] for a proof of this characterization).

3 Semantics-Preserving Procedure Extraction

In this section we define an algorithm for reordering a given set of CFG nodes so that they can be extracted into a procedure while preserving semantics.

We assume the following inputs to the algorithm:

1. P, the control-flow graph of a procedure.
2. the set $\mathcal{M}$ of nodes in P that have been chosen for extraction ($\mathcal{M}$ is a subset of $\mathcal{N}(P)$ − {Enter, Exit})

The goal of the algorithm is to produce a CFG $P_{\mathcal{M}}$ that includes exactly the same nodes as P, so that:

- the nodes in $\mathcal{M}$ are extractable from $P_{\mathcal{M}}$, and
- $P_{\mathcal{M}}$ is semantically equivalent to P.

A very high-level description of our algorithm is as follows:

Step 1: Check whether the nodes in $\mathcal{M}$ are part of a chain of atomic hammocks in P; if not, then fail (P cannot be reordered to make the $\mathcal{M}$ nodes extractable while preserving control dependences).

Step 2: Create a polygraph that represents the ordering constraints imposed on the hammocks in the chain by data dependence considerations. (A polygraph is a graph with both "normal" edges and "either-or" edges. This will be clarified in Section 3.2 below.)

Step 3: Create the set of acyclic graphs defined by the polygraph created in Step 2.

Step 4: If any of the graphs created in Step 3 has a simple property (to be defined in Section 3.2), produce the corresponding CFG $P_{\mathcal{M}}$; otherwise, fail.

Subsection 3.1 explains the notion of extractability, and then step 1 of the algorithm, which concerns extractability and preserving control dependence. Subsection 3.2 elaborates on Steps 2–4, which have to do with preserving data dependence.

3.1 Extractability and Control Dependence

As stated earlier, a requirement is that the nodes in $\mathcal{M}$ be extractable from $P_{\mathcal{M}}$. What this means is that step 3 of procedure extraction – extracting the $\mathcal{M}$-nodes from $P_{\mathcal{M}}$ and replacing them with a procedure call node – does not result in a malformed CFG. In the example in the Introduction, nodes 5 and 7 can be moved together to result in a new CFG from which they are extractable, but this is not true for nodes 3 and 4 although they are already together. In essence, since the extracted procedure will have a single entry point, and the new procedure call node will have a single CFG successor, the set of $\mathcal{M}$-nodes must have the same two properties in $P_{\mathcal{M}}$ for the replacement to make sense: there must be a single $\mathcal{M}$-node that has incoming edges from outside the set, and all edges leaving the set must go to the same CFG node. It is thus easy to see that the nodes in $\mathcal{M}$ are extractable from $P_{\mathcal{M}}$ iff they form a hammock in $P_{\mathcal{M}}$.

Example 2 in the Introduction illustrates that a part of a sufficient condition to guarantee the semantic equivalence of P and $P_{\mathcal{M}}$ is that each node in $P_{\mathcal{M}}$ have the same control dependence set as in P.

We have shown in [KH99] that both of these objectives for the new CFG $P_{\mathcal{M}}$ – extractability and control dependence preservation – can be achieved iff in the original CFG P the nodes in $\mathcal{M}$ are part of a chain $\mathcal{C}$ of atomic hammocks (this result is stated as Theorem 1 in the appendix, and a proof outline is given).[1]

Every hammock in $\mathcal{C}$ must be either an $\mathcal{M}$-hammock – a hammock in which all nodes are in $\mathcal{M}$ – or an $\mathcal{O}$-hammock – one that has no nodes in $\mathcal{M}$. Figure 1(a) illustrates this structure; the $\mathcal{M}$-hammocks are shaded. If we look back at Example 4 in the Introduction, $(4, 5, 6, 7)$ is the chain that contains nodes 5 and 7 with each of the nodes being an atomic hammock.

Algorithm Step 1 (finding chain $\mathcal{C}$)
Step 1 of our algorithm determines whether there is such a chain $\mathcal{C}$ in P as follows:

i. Identify the set of all hammocks in P:

```
for each node $p$ in $P$
    for each postdominator $q$ of $p$
        (a) do a depth-first search starting from $p$
            and not going past $q$; let $H$ be the set of
            nodes visited by the depth-first search
        (b) $H$ is a hammock iff all edges coming into
            $H$ from outside $H$ have $p$ as their target.
```

ii. Eliminate from the set all hammocks that are neither $\mathcal{M}$-hammocks nor $\mathcal{O}$-hammocks (i.e., all hammocks that contain both $\mathcal{M}$-nodes and non-$\mathcal{M}$-nodes).

iii. Eliminate all non-atomic hammocks (see Section 2).

iv. Eliminate all non-maximal hammocks (a hammock is non-maximal if all of its nodes are contained in another hammock in the current set).

[1] A part of the proof also shows that P has such a chain iff the following two control-dependence conditions are met:

1. For every predicate node p in $\mathcal{M}$, all nodes that are control dependent on p are also in $\mathcal{M}$.
2. All nodes in $\mathcal{M}$ that are (directly) control-dependent on some node outside $\mathcal{M}$ have the *same* control dependence set outside $\mathcal{M}$.

We say that $\mathcal{M}$ is *well-formed in control dependence in P* in this case.

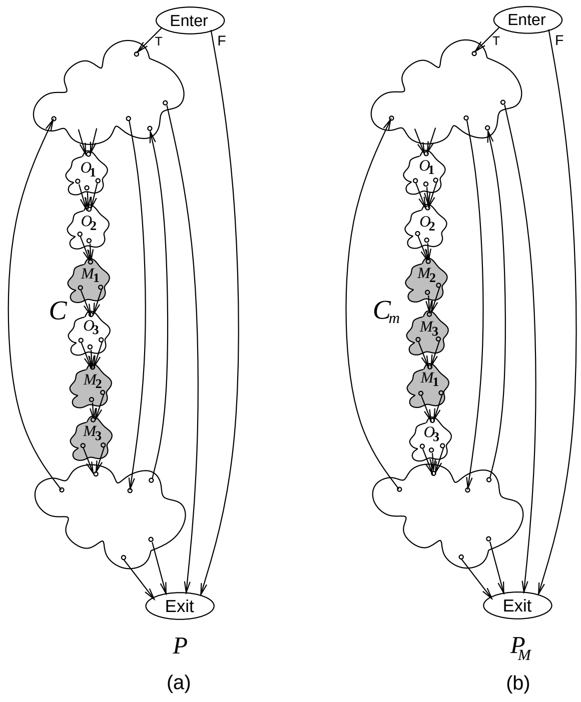


Figure 1: Chain containing $\mathcal{M}$ nodes

v. Check whether all $\mathcal{M}$-nodes are included in the final set of hammocks; if not, fail.

vi. Find the longest chain of atomic hammocks starting with any $\mathcal{M}$-hammock M_s as the current hammock: if the outside exit node of the current hammock is the entry node of some hammock H in the set, and all edges to this entry node from outside H come from the current hammock, then add H to the chain and make H the current hammock.

vii. Extend the chain backwards from M_s as far as possible: start with M_s as the current hammock; if the entry node of the current hammock is the outside exit node of some hammock H in the set, and all edges into the current hammock are from H, then add H to the beginning of the chain, and make H the current hammock.

viii. If all of the $\mathcal{M}$-hammocks are in the chain then chain $\mathcal{C}$ has been identified; if not, fail.

Example: For the CFG shown in Figure 1(a), step (ii) would eliminate from the set all hammocks that are neither $\mathcal{M}$-hammocks nor $\mathcal{O}$-hammocks (such as the entire CFG). It would leave in the set any O-hammocks that occur in the upper and lower "clouds" of code, and hammocks O_1, O_2, M_1, O_3, M_2, and M_3, as well as the non-atomic hammocks (O_1, O_2), and (M_2, M_3). Those non-atomic hammocks would be eliminated in step (iii). Any non-maximal hammocks inside the M_is or O_is or in the "clouds" would be eliminated in step (iv). Such hammocks arise, for example, in the context of a loop: the whole loop can be an atomic hammock, as well as the individual statements inside the loop; however, the individual statements are not maximal hammocks. Continuing with the example, step (v) would succeed, and step (vi) could start with M_1, M_2, or M_3. If it started with M_2, it would find the chain (M_2, M_3). Step (vii) would then extend that chain backward; the final chain would be: $(O_1, O_2, M_1, O_3, M_2, M_3)$.

3.2 Data Dependence

The goal of Steps 2–4 of our algorithm is to determine whether it is possible to permute the chain $\mathcal{C}$ into a chain $\mathcal{C}_m$ by reordering its hammocks so that:

- the $\mathcal{M}$-hammocks occur contiguously in $\mathcal{C}_m$, and
- the semantics of P are preserved.

If this can be done, then the CFG obtained by replacing $\mathcal{C}$ with $\mathcal{C}_m$ – shown in Figure 1(b) – is our desired CFG $P_{\mathcal{M}}$.

To preserve the semantics, we must ensure that on every execution, each node in P executes the same number of times and in the same states as in $P_{\mathcal{M}}$. It can be shown that permuting $\mathcal{C}$ in any way cannot alter the control dependence set of any node (see Part 3 of the proof outline of Theorem 1 in the appendix). Intuitively then, a permutation preserves semantics if it preserves the flow of values in the program: for each execution, if a node n in P uses a value defined at node m, then node n in $P_{\mathcal{M}}$ must also use the value defined at node m.

These observations lead us to define six kinds of ordering constraints (imposed by chain $\mathcal{C}$) that must be satisfied by the permutation. Some of the constraints are simple constraints of the form "hammock A must come before hammock B in the permutation." Others are "either-or" constraints of the form "either hammocks $A_1 \ldots A_j$ must all come before hammock B in the permutation, or hammock B must come before hammock C."

We can show that these 6 kinds of ordering constraints are sufficient: any permutation of the chain $\mathcal{C}$ that satisfies all of the constraints imposed by $\mathcal{C}$ preserves semantics. Theorem 2 in the appendix states this result formally, and a proof outline is included (see [KH99] for a more detailed version).

Algorithm Step 2 (building the polygraph)
Step 2 of our algorithm involves building a *polygraph* that represents the ordering constraints imposed by the chain $\mathcal{C}$. Each node of the polygraph corresponds to one atomic hammock in $\mathcal{C}$; each edge in the polygraph represents one ordering constraint. A polygraph has two kinds of edges: *normal* edges (e.g., $A \to B$), which represent simple constraints, and *either-or* edges (e.g., $\{A_1, A_2, \ldots A_j\} \to B \parallel B \to C$), which represent either-or constraints. A polygraph defines a *set* of (normal) graphs. Each graph in the set has the same nodes as the polygraph, and includes all of the polygraph's normal edges. For each either-or edge $\{A_1, A_2, \ldots A_j\} \to B \parallel B \to C$ in the polygraph, a graph in the set either includes the edges $A_1 \to B$, $A_2 \to B$, $\ldots$ $A_j \to B$, or it includes the edge $B \to C$. A polygraph that has k either-or edges thus defines a set of 2^k normal graphs.

The six kinds of ordering constraints are induced by instances of flow, def-order, anti, and output dependences between the hammocks in $\mathcal{C}$. The constraints are defined below and are illustrated using the chain shown in Figure 2, in which each node represents one atomic hammock, and the shaded nodes are $\mathcal{M}$-hammocks. Note that because of our assumptions that there are no uninitialized uses and no dead assignments, there must be a definition of variable v outside the chain that reaches chain entry, and v must be live at chain exit. For the purposes of this example, we will

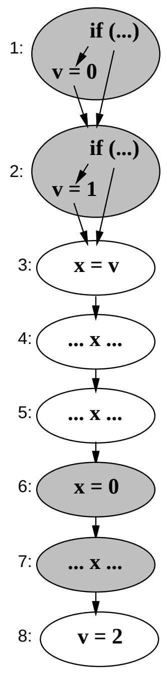


Figure 2: Chain used to illustrate ordering constraints (each node represents one atomic hammock)

assume that the definition of v in hammock 8 does not reach the chain entry, and that x is not live at chain exit.

1. *Constraints induced by flow dependence*: Normal edge $A \to B$ is in the polygraph for chain $\mathcal{C}$ if
   (a) hammock A comes before hammock B in $\mathcal{C}$, and
   (b) there is a definition of a variable v in A that reaches a use of v in B.

   Example: For the chain in Figure 2, flow dependences induce the normal polygraph edges $1 \to 3$, $2 \to 3$, $3 \to 4$, $3 \to 5$, and $6 \to 7$.

2. *Constraints induced by def-order dependence*: Normal edge $A \to B$ is in the polygraph for chain $\mathcal{C}$ if
   (a) A comes before B, and
   (b) there are definitions of a variable v in both A and B, and
   (c) there is a use of v somewhere in the program that is reached by the definitions in both A and B.

   Example: Def-order dependences induce the normal polygraph edge $1 \to 2$ (because of the use of v in hammock 3).

3. *Constraints induced by anti dependence*: Normal edge $A \to B$ is in the polygraph for chain $\mathcal{C}$ if
   (a) A comes before B, and
   (b) there is a use of a variable v in A that is reached by a definition outside $\mathcal{C}$, and there is a definition of v in B.

Example: Anti dependences induce the normal polygraph edge $3 \rightarrow 8$ (because of the definition of v outside the chain that reaches chain entry).

4. *More constraints induced by anti dependence*: Either-or edge $\{A_1, A_2, \ldots, A_j\} \rightarrow B \parallel B \rightarrow C$ is in the polygraph for chain $\mathcal{C}$ if
   (a) C comes before any of the A hammocks, which all come before B, and
   (b) there is a non-empty set of variables V that are defined in both B and C, and
   (c) every A_i includes at least one use of a variable $v \in V$ that is reached by a definition in C, and
   (d) the set $A_1, ..., A_j$ is maximal: every hammock U that comes after C and before B, and that includes a use of a variable $v \in V$ that is reached by a definition in C is in the set.

   Example: Anti dependences induce the either-or polygraph edges $(\{4,5\} \rightarrow 6 \parallel 6 \rightarrow 3)$, $(\{3\} \rightarrow 8 \parallel 8 \rightarrow 1)$, and $(\{3\} \rightarrow 8 \parallel 8 \rightarrow 2)$. However, note that the second and third edges are redundant, since normal edge $3 \rightarrow 8$ is also included in the polygraph because of a type 3 constraint.

5. *Constraints induced by output dependence*: Normal edge $A \rightarrow B$ is in the polygraph for chain $\mathcal{C}$ if
   (a) A comes before B, and
   (b) there is a definition of a variable v in both A and B, and
   (c) v is live at the exit of the chain $\mathcal{C}$, and
   (d) the definition in B reaches the exit of the chain but the definition in A does not reach the exit of the chain.

   Example: Output dependences induce the normal polygraph edges $1 \rightarrow 8$ and $2 \rightarrow 8$ (because v is live at chain exit).

6. *More constraints induced by output dependence*: Either-or edge $\{A_1, A_2, \ldots, A_j\} \rightarrow B \parallel B \rightarrow C$ is in the polygraph for chain $\mathcal{C}$ if
   (a) B comes before C, which comes before any of the A hammocks, and
   (b) there is a non-empty set of variables V that are defined in both B and C, and
   (c) every A_i includes at least one use of a variable $v \in V$ that is reached by a definition in C but is not reached by any definition in B, and
   (d) the set $A_1, ..., A_j$ is maximal: every hammock U that comes after C and that includes a use of a variable $v \in V$ that is reached by a definition in C but is not reached by any definition in B is in the set.

   Example: Output dependences induce the either-or polygraph edge $\{7\} \rightarrow 3 \parallel 3 \rightarrow 6$.

Note that because we have assumed that input and output are implemented using stream variables, there is no need to include special cases for constraints induced by I/O. For example, the statement *input x* is treated as both a use and a definition of the stream variable *input* (as well as a definition of x); therefore, if two hammocks in the chain each include an input statement, there will be a flow dependence from one hammock to the other, and the constraints defined above will ensure that the order of the input statements in the chain is maintained.

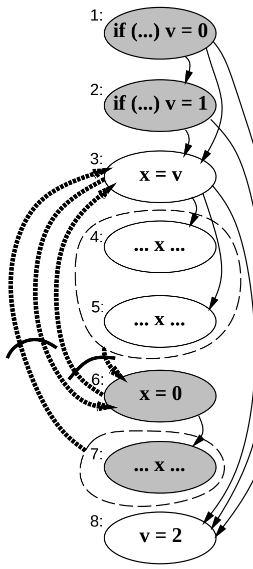


Figure 3: Polygraph built for the chain of Figure 2

Example: The polygraph built for the example chain in Figure 2 is shown in Figure 3 (only the two non-redundant either-or edges are included). Normal edges are shown using plain arrows; either-or edge $\{A_1, A_2, ..., A_j\} \rightarrow B \parallel B \rightarrow C$ is indicated by enclosing the A_i nodes in a dashed circle, connecting that circle to node B with a heavy dashed arrow, connecting node B to node C with a second heavy dashed arrow, and linking the two dashed arrows with an arc.

Algorithm Step 3 (creating the acyclic graphs defined by the polygraph)
It is easy to see that a permutation $\mathcal{C}'$ of $\mathcal{C}$ satisfies all the constraints imposed by $\mathcal{C}$ iff $\mathcal{C}'$ is consistent with the edges of (at least) one of the graphs defined by the polygraph for $\mathcal{C}$ (which was created in step 2). Moreover, no permutation of $\mathcal{C}$ can be consistent with the edges of a cyclic graph defined by the polygraph. Therefore, to find permutations of $\mathcal{C}$ that satisfy all constraints imposed by $\mathcal{C}$, step 3 of our algorithm creates all acyclic graphs defined by the polygraph. This can be done with a recursive routine `CreateGraphs` whose inputs are a graph G and a set of either-or edges E. The inputs to the top-level call of this routine are: a graph that contains all the polygraph's normal edges, and the set of all non-redundant either-or edges defined by the polygraph. The steps of the routine are:

1. If E is empty, return $\{G\}$ (G is one of the acyclic graphs defined by the polygraph).
2. Select an either-or edge $e = \{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$ from E.
3. Let G' be equal to G augmented with edge $B \rightarrow C$; if G' is acyclic then call `CreateGraphs` recursively with

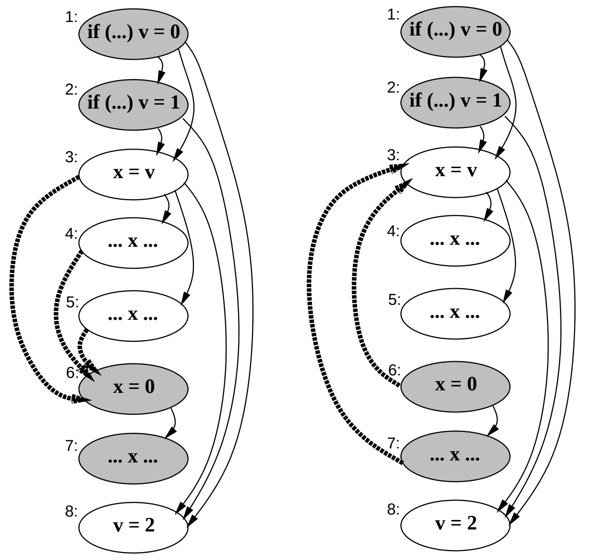


Figure 4: Acyclic graphs defined by the polygraph of Figure 3

G' and $E - \{e\}$ as inputs. Let $S1$ be the set returned by this recursive call.

4. Let G'' be equal to G augmented with edges $A_1 \to B, A_2 \to B, \ldots, A_j \to B$. If G'' is acyclic then call `CreateGraphs` recursively with G'' and $E - \{e\}$ as inputs. Let $S2$ be the set returned by this recursive call.

5. Return $S1 \cup S2$.

Example: Two acyclic graphs are defined by the polygraph shown in Figure 3. They are shown in Figure 4. Heavy dashed arrows indicate the edges that were added to account for the polygraph's either-or edges; plain arrows correspond to the polygraph's normal edges.

Algorithm Step 4 (creating the goal CFG $P_{\mathcal{M}}$)
Step 4 of our algorithm determines whether it is possible to move the nodes in $\mathcal{M}$ together so that they are extractable, without violating any of the constraints imposed by data dependence. This is accomplished by determining whether there is a permutation of the chain $\mathcal{C}$ in which all of the $\mathcal{M}$-hammocks are contiguous, and that is consistent with the edges of (at least) one of the acyclic graphs created in step 3. If such a permutation $\mathcal{C}_m$ is found, we can clearly produce $P_{\mathcal{M}}$ (in which all the $\mathcal{M}$ nodes form a hammock, and are thus extractable) by replacing $\mathcal{C}$ with $\mathcal{C}_m$ in P.

It is easy to see that permutation $\mathcal{C}_m$ exists iff there is an acyclic graph G created in step 3 such that there are no paths in G that run from an $\mathcal{M}$-node to an $\mathcal{O}$-node to an $\mathcal{M}$-node[2] (the existence of such a path would preclude moving the endpoint $\mathcal{M}$-hammocks together).

This property can be easily checked for each graph G created in step 3 by using depth-first search as follows:

For each $\mathcal{O}$-node n in G:

1. Use depth-first search from n to determine whether there is a path to an $\mathcal{M}$-node.
2. If yes, use reverse depth-first search from n to determine whether there is also a path *from* an $\mathcal{M}$-node.
3. If yes, reject G

[2]Here we use $\mathcal{O}$-node to mean a node in G that represents an $\mathcal{O}$-hammock, and $\mathcal{M}$-node to mean a node in G that represents an $\mathcal{M}$-hammock.

Once an acyclic graph G has been identified that has no such "illegal" paths, a final ordering $\mathcal{C}_m$ of the hammocks of the chain $\mathcal{C}$ can be produced by finding topological orderings of three subgraphs of G:

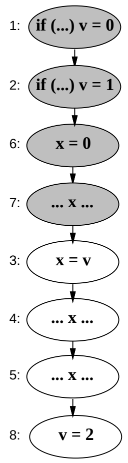
1. The subgraph induced by the set of $\mathcal{O}$-nodes from which there *is* a path to an $\mathcal{M}$-node.

2. The subgraph induced by the set of $\mathcal{M}$-nodes.

3. The subgraph induced by the set of $\mathcal{O}$-nodes from which there is *not* a path to an $\mathcal{M}$-node.

The final ordering is the concatenation of the three topological orderings. Since G is acyclic, and there are no paths from an $\mathcal{M}$-node to an $\mathcal{O}$-node to an $\mathcal{M}$-node, the orderings are guaranteed to exist, and the concatenation is guaranteed to be consistent with the edges of G, thus satisfying all of the constraints imposed by the original chain $\mathcal{C}$.

Example: The first graph shown in Figure 4 would be rejected by step 4 of the algorithm because $\mathcal{O}$-nodes 3, 4 and 5 are all reachable from $\mathcal{M}$-nodes 1 and 2, and can reach $\mathcal{M}$-nodes 6 and 7. The second graph has no such illegal paths; in fact, there are no paths from an $\mathcal{O}$-node to a $\mathcal{M}$-node. Therefore, there are only two topological orderings to concatenate to form the final ordering (the ordering of the $\mathcal{M}$-nodes, followed by the ordering of the $\mathcal{O}$-nodes). One such final ordering is shown below.



4 Algorithm Complexity

The space complexity of the algorithm is polynomial in the length of chain $\mathcal{C}$, provided we make a minor change to the algorithm: instead of creating all the acyclic graphs in step 3 and then checking each of them in step 4, we need to merge the two steps. Thus, when an acyclic graph is created, step 4 should be applied on it. If the graph is rejected for having a bad path then it can be thrown away; otherwise, the algorithm can generate final ordering $\mathcal{C}_m$ and stop.

The time complexity of the algorithm is dominated by the combined steps 3 and 4. To see why, we begin by observing that finding the chain $\mathcal{C}$ (step 1 of the algorithm) takes time polynomial in the size of the CFG. Building the polygraph for chain $\mathcal{C}$ (step 2 of the algorithm) takes time polynomial in n, where n is the number of hammocks in $\mathcal{C}$ (n itself is polynomial in the size of the CFG).

The total time required to generate the acyclic graphs defined by the polygraph and find one without a bad path (combined step 3 and 4 of the algorithm) is determined by the total number of calls to the recursive routine `CreateGraphs` (described in Section 3.2). An individual call to this routine can make up to two recursive calls of its own, and therefore the total number of activations at any given depth k from the top-level activation is $\leq 2^k$. Since the maximum depth of recursion is equal to the number of either-or edges, which we refer to as e, and since $k \leq e$, the total number of calls to routine `CreateGraphs` is bounded above by $e \times 2^e$. Thus the time requirement of combined steps 3 and 4 is bounded above by $((e \times 2^e) \times (\text{some polynomial in } n))$. e itself is $O(n^2)$ in the worst case, as for any either-or edge $\{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$, there are $O(n^2)$ distinct pairs of hammocks for B and C and the set $A_1, A_2, \ldots A_j$ is fixed for any particular B and C.

We can improve this upper bound by recognizing that the number of activations of `CreateGraphs` at any given depth k from the top-level activation is also bounded above by $n!$ ($n!$ could be smaller than 2^k as k could be equal to e which itself is $O(n^2)$). The reason for this can be seen by considering the set of acyclic graphs received as inputs by the activations at this depth: each acyclic graph in the set has (at least) one permutation of $\mathcal{C}$ that is consistent with its edges, no permutation of $\mathcal{C}$ can be consistent with more than one graph in the set (because any two graphs differ on at least one either-or edge and no permutation can be consistent with both alternatives of an either-or edge), and there are only $n!$ different permutations of $\mathcal{C}$. Therefore an improved upper bound on the total time requirement of steps 3 and 4 is $((e \times \min(n!, 2^e)) \times (\text{some polynomial in } n))$.

This upper bound on the algorithm's time requirement is not surprising, since we have been able to show that the problem of finding a permutation $\mathcal{C}_m$ of $\mathcal{C}$ (if one exists) such that the $\mathcal{M}$ hammocks occur contiguously in $\mathcal{C}_m$, and $\mathcal{C}_m$ satisfies all the constraints imposed by $\mathcal{C}$, is NP–Hard in n. We omit the NP–Hardness proof in this paper due to space constraints; it involves a reduction from the NP–Complete problem of determining whether a given schedule of database transactions is view-serializable (see [Pap86]).

Although this upper bound looks prohibitive, there is some evidence that the algorithm will work well in practice: We have measured n and e for all chains of atomic hammocks in a set of benchmark programs, and the results of the study (reported in the next section) are very encouraging. The bottom line is that in all of the programs, fewer than 1% of the chains have polygraphs with more than 5 either-or edges (i.e., have $e > 5$). Thus, it seems likely that the algorithm will usually have a reasonable running time.

Furthermore, the values of both n and e will be known by the end of step 2 (creating the polygraph), after doing work only polynomial in the size of the CFG; if both values are large, heuristics can be used in place of step 3. Two possible heuristics are:

1. Instead of generating the acyclic graphs defined by the polygraph, generate the permutations of $\mathcal{C}$ in which the $\mathcal{M}$-hammocks occur contiguously, and in which the relative ordering of the $\mathcal{M}$-hammocks and the relative ordering of the $\mathcal{O}$-hammocks are the same as in $\mathcal{C}$ (there are only $O(n)$ such permutations). For each generated permutation, check whether it satisfies the constraints of the polygraph. If so, use that permutation in place of $\mathcal{C}$ to obtain the CFG $P_{\mathcal{M}}$.
2. Limit the number of graphs defined by the polygraph by arbitrarily converting some or all of its either-or edges to normal edges, then generate the corresponding acyclic graphs.

Of course, these heuristics will fail in some cases where the actual algorithm succeeds, but they may work well in practice.

5 Experimental Results

To provide some insight into the actual running time of our algorithm, we analyzed all chains of atomic hammocks in a set of benchmark programs. The steps carried out by the analysis for each program are listed below:

1. Use the SUIF compiler infrastructure front-end [WFW+94] to build an intermediate form for the program.
2. Build a CFG for each procedure from its intermediate form.
3. Perform pointer analysis on the entire program. This provides information on the variables that might be pointed to by pointer variables, which in turn helps us in determining the variables that might be defined/used by statements that dereference pointers.
4. Compute summary information for each procedure. This information consists of the set of variables that might be defined, and the set of variables that might be used, as a result of a call to the procedure. This summary information is used to compute data dependences between call nodes and other nodes.
5. For each procedure in the program:

(a) Identify all atomic hammocks. Compute the may-use-before-defined set – the set of variables that might be used in a hammock before being defined – for each hammock by performing a backward dataflow analysis within the hammock. Compute the may-define set – the set of variables that may be defined by a hammock – for each hammock by unioning the may-define sets of all nodes in the hammock. Compute the must-define set of each hammock by unioning the must-define sets of all nodes in the hammock that postdominate the entry node.

(b) Identify all maximal-length chains of atomic hammocks in the CFG, as described in Algorithm Step 1 (in Section 3.1).

(c) For each chain, use the may-use-before-defined, may-define and must-define sets of the hammocks to compute the normal edges and either-or edges in the chain's polygraph.

The normal edges in the polygraph induce a transitive precedence relation on the hammocks in the chain. If A and B are hammocks in a chain, then the presence of a normal edge $A \rightarrow B$ implies that A precedes B in the relation. An either-or edge $\{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$ is redundant if one of the A_i's precedes or is preceded by B, or if B precedes or is preceded by C. Every redundant either-or edge is eliminated

Program	Number of procedures	Program size	
		No. of lines of source	Total No. of CFG nodes
agrep	65	6220	20725
allroots	6	449	724
anagram	16	655	1348
bc	101	8576	16146
bison-1.2.2	150	7852	27769
flex-2.4.7	147	8459	21975
football	57	2327	19383
gzip-1.2.4	99	7624	17886
ispell-4.0	121	7768	16484
simulator	110	5307	12049

Figure 5: Information about benchmark programs

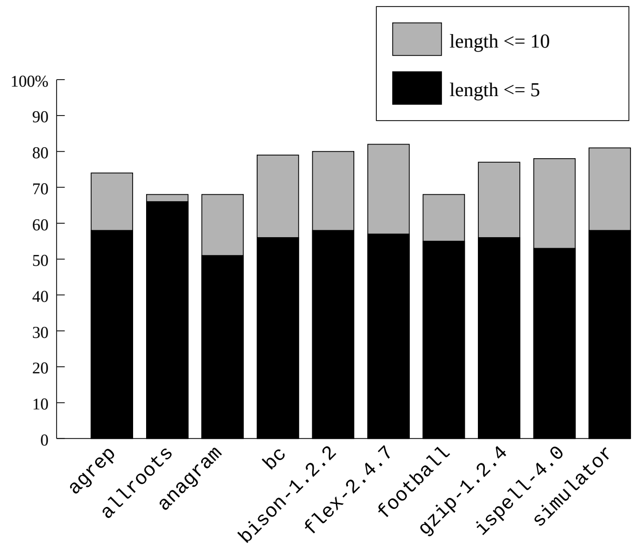


Figure 6: Distribution of chains by length

and replaced by a set of normal edges; for instance if C is known to precede B, then $\{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$ is replaced by the set of normal edges $\{A_1 \rightarrow B, A_2 \rightarrow B, \ldots, A_j \rightarrow B\}$.

Figure 5 gives some statistics for the benchmark programs we analyzed; *bc, bison, flex, gzip* and *ispell* are Gnu Unix utilities; *agrep* is described in [WM92]; *anagram* has been used in the experiments of [ABS94], while *allroots, football* and *simulator* were used in the experiments of [LRZ93].

Figure 6 gives the distribution of chains by length in each program. The taller bar gives the percentage of chains that have length ≤ 10, while the solidly shaded portion indicates the percentage of chains that have length ≤ 5. It can be observed that from 50 to 65% of chains have length ≤ 5. Figure 7 gives the cumulative number of chains over all programs for various chain lengths.

Figure 8 gives the distribution of chains by the number of either-or edges they have. The taller bar gives the percentage of chains that have ≤ 5 either-or edges, while the solidly shaded portion indicates the percentage of chains that have 0 either-or edges. It can be observed that from 91 to 99%

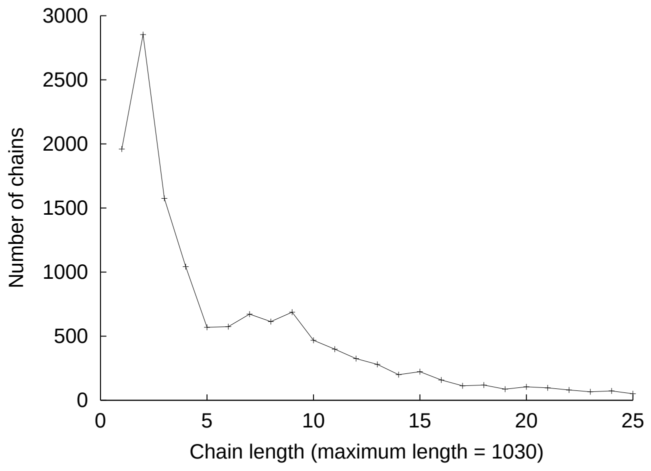


Figure 7: Chain lengths over all programs

of chains have no either-or edges, and virtually all the remaining ones have ≤ 5 edges. Figure 9 gives the cumulative number of chains over all programs that have a given number of either-or edges.

Recall that the algorithm's worst case time bound is proportional to $\min(n!, 2^e) \times$ (some polynomial in n) (where n is the length of the chain and e is the number of either-or edges in it). Our experimental results indicate that chain lengths tend to be short, and that the number of either-or edges tends to be very small; this is a strong indication that the running time of the algorithm will actually be polynomial in n for most chains in real programs, thus making it a feasible one in practice.

We also studied whether many either-or edges were made redundant by normal edges. The results were in the negative; in each program over 98% of the chains had ≤ 5 either-or edges even if redundant ones were *not* eliminated. We believe this further validates our conclusions.

Finally, Figure 10 gives for each program the total number of chains, length of the longest chain, and the maximum number of either-or edges in any single chain.

5.1 Factors Affecting the Experimental Results

Although our experimental results are quite encouraging, there are a number of factors that, if changed, might cause somewhat different results to be produced. These factors are discussed below.

Using the SUIF intermediate form: The SUIF intermediate form is a low-level representation. Since we build the CFG from the SUIF representation, each source level statement can correspond to many CFG nodes (this can be observed from the data in Figure 5). As an example, the single source level statement "*p++ = 0" becomes a series of CFG nodes that first save the original value of p into a temporary, then increment p, and finally store 0 into the location pointed to by the temporary. A result of this low-level representation is that we will tend to have longer chains than if we had worked at the source level, and we may have more normal and either-or edges per chain. On the other hand, the low-level representation has the advantage of flexibility; in the above example, it allows an extracted procedure to

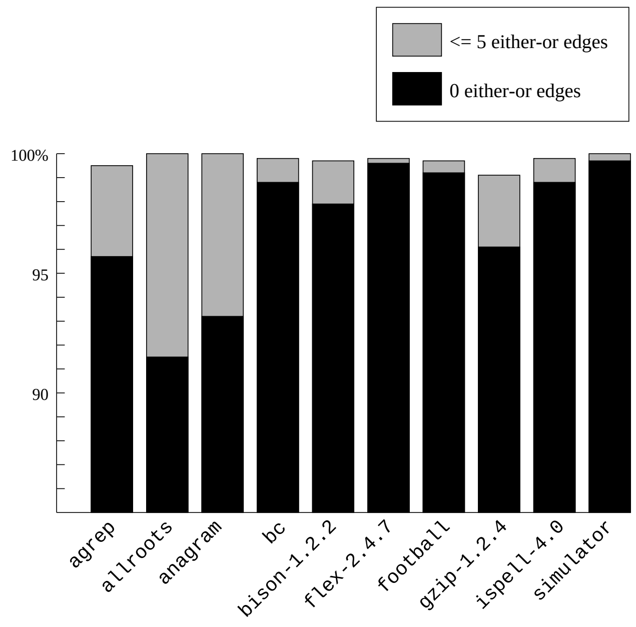


Figure 8: Distribution of chains by number of either-or edges (Note that the y-axis starts at 85%)

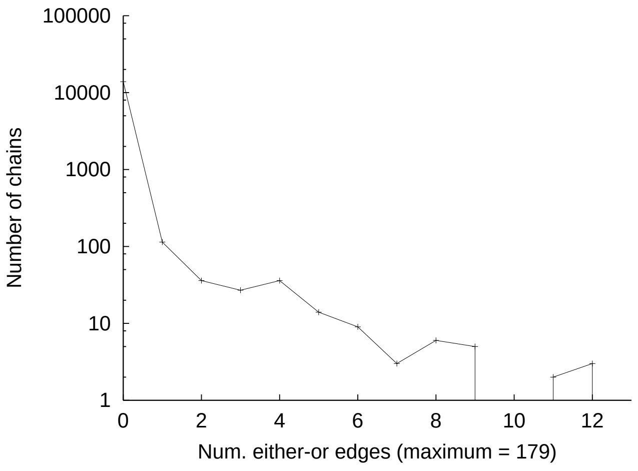


Figure 9: either-or edge counts over all programs

Program	Total number of chains	Longest chain	Max. num. of either-or edges
agrep	1821	99	86
allroots	47	93	3
anagram	104	70	4
bc	1514	169	8
bison-1.2.2	2487	80	35
flex-2.4.7	2151	194	179
football	1169	346	12
gzip-1.2.4	1680	127	133
ispell-4.0	1589	1030	6
simulator	1236	60	1

Figure 10: Measured statistics about benchmark programs

include just the increment of the pointer, or just the storing of the value 0.

Handling of pointers: For the sake of efficiency, we perform the *flow-insensitive* pointer analysis defined by Andersen [And94]. This in general gives less accurate results than a flow-sensitive analysis, which in turn could increase or decrease the number of normal and either-or edges in a chain.

Live Variable Analysis: Computing the normal edges induced by output dependence (the fifth type of constraint described in Step 2 of the algorithm, Section 3) for a chain requires knowing which variables are live at chain exit. Rather than performing a whole-program live-variable analysis we assume that no variables are live at chain exit. This assumption can only increase the number of either-or edges reported for any chain.

Computing summary information for procedures: As mentioned earlier, we need to compute may-define and may-use sets for each procedure so that data dependence information for call nodes is known. We let the may-define (may-use) set of a procedure be equal to the set of variables that may be defined (used) by the procedure itself and by other procedures that could be (directly or indirectly) called by it.

We might have gotten different sets of normal and either-or edges if we had computed may-use-before-define summary information rather than just may-use information, but that would require an expensive whole-program analysis which we left out for the sake of efficiency.

Another issue is which variables are included in the may-define and may-use sets for a call node. Including all variables in the called procedure's may-define and may-use sets – even variables that are not visible to the calling procedure – can cause extra normal and either-or edges to be included in the polygraph (e.g., if a hammock chain includes two calls to the same procedure, and that procedure both defines and uses a local variable x, then there will be a flow dependence, and therefore a normal edge, between the two call nodes, since the definition of x at the first call will be considered to reach the use at the second call).

In our implementation we include in a call node's may-define (may-use) set all variables in the called procedure's may-define (may-use) set that are global, static, arrays, or structures, or whose addresses are taken. Including (local) structures is probably overly conservative; however, SUIF provides a single boolean function that identifies arrays, structures, globals, and variables whose addreses are taken; using that function provides a safe approximation to the set of variables in the called procedure's may-define and may-use sets that should be in the call node's sets.

Handling calls to library functions: Since source code for library functions is unavailable for analysis, we provided summary functions for most library functions. These summaries simulate definitions and uses of all variables that are not local to the library function (our summaries were based on the summaries that were used in the experiments reported in [WL95]). Use of different summary functions could lead to different sets of polygraph edges.

6 Key Theorems

Here we state two key theorems; proof outlines are given in the Appendix. The first theorem shows that it is possible to move the nodes in $\mathcal{M}$ together so that they are extractable (they form a hammock) while preserving control-dependence sets for all nodes in P iff the nodes in $\mathcal{M}$ are part of a chain $\mathcal{C}$ of atomic hammocks in P. This demonstrates that Step 1 of our algorithm (which fails if the nodes in $\mathcal{M}$ are not part of a chain of atomic hammocks) is reasonable. The second theorem demonstrates that our algorithm is correct, by showing that any permutation of the hammocks in $\mathcal{C}$ that satisfies the ordering constraints imposed by $\mathcal{C}$ (as defined in Section 3.2) preserves semantics.

Theorem 1 *: Given CFG P and set of nodes $\mathcal{M}$, it is possible to create a new CFG P_e with the same nodes as P, with each node having the same set of control dependences as in P, and in which the $\mathcal{M}$ nodes form a hammock iff there is a chain $\mathcal{C}$ of atomic hammocks in P such that:*

1. *$\mathcal{C}$ includes all of the nodes in $\mathcal{M}$, and*
2. *every hammock in $\mathcal{C}$ is either an $\mathcal{M}$-hammock – a hammock in which all nodes are in $\mathcal{M}$ – or an $\mathcal{O}$-hammock – one that has no nodes in $\mathcal{M}$.*

Theorem 2 *: Given:*

1. *Chain $\mathcal{C}$ of atomic hammocks in CFG P, such that all nodes in $\mathcal{M}$ are in the chain, and every hammock is either an $\mathcal{M}$-hammock (containing only $\mathcal{M}$ nodes) or an $\mathcal{O}$-hammock (containing no $\mathcal{M}$ nodes), and*
2. *$\mathcal{C}'$, a permutation of $\mathcal{C}$ such that $\mathcal{C}'$ satisfies the ordering constraints imposed by $\mathcal{C}$ (as defined in Section 3.2), and*
3. *P', the CFG obtained by replacing chain $\mathcal{C}$ with the chain $\mathcal{C}'$ in P*

then: P and P' are semantically equivalent.

7 Related Work

Related work falls into two main categories: work related to procedure extraction (including [GN93, LD98, LV97, BG98]), and work on semantics-preserving transformations (including [BD77, LMW79, Ram88, PP96, Fea82, LV97, BG98, BDFH97, CLZ86, FOW87]).

[GN93] describes a tool that supports a set of meaning-preserving transformations on Scheme programs, including one that extracts a given contiguous sequence of expressions into a new function and replaces the sequence by a call to the function. If the user wishes to extract non-contiguous expressions into a function, the expressions must first be moved together using a transformation that moves a given expression to a given point if this does not change the program's meaning. The user is respnsible for identifying the correct program point and the correct ordering for the expressions. Our algorithm automates the aspect of bringing non-contiguous code together while preserving meaning. Additionally, our algorithm works on general unstructured programs whereas their transformations are limited to structured programs.

[LD98] also addresses the problem of procedure extraction, but their goal and approach are quite different from ours. Their goal is to discover and extract a meaningful computation surrounding a programmer-specified "seed" set of statements, within a programmer-specified bounding hammock. Their approach is to take the backward slice of the seed within the hammock, and then attempt to extract the slice as a procedure. The use of slices implicitly imposes the second part of our control dependence well-formedness condition. Their data dependence condition is quite restrictive: there can be no data flow from the extracted computation to the remaining computation in the bounding hammock, and vice versa, and no variable can be defined in both computations if the definitions reach uses outside the bounding hammock. In effect, they perform extraction only when there is no data dependence interaction between the code to be extracted and the rest of the code in the bounding hammock. In our approach, code can be extracted successfully in many situations where there are complex data dependence interactions between the extracted code and the remaining code. This flexibility comes at the price of a high worst-case time requirement; however, our experimental results indicate that this may not be a problem in practice. On the other hand, their approach allows predicate nodes to be duplicated, which may allow some procedure extractions that would not be possible using our approach. It may be possible to combine the advantages of both approaches: applying our procedure extraction algorithm on the backward slice of the seed will allow it to be extracted in some situations where their approach fails. Conversely, we might be able to make our data dependence condition more restrictive to reduce the time complexity of our algorithm, and to extend our algorithm to permit automatic code duplication when that is necessary to prevent the algorithm from failing.

Both [LV97] and [BG98] describe tools that identify sets of statements to be extracted into a procedure based on some user input. In [LV97], the programmer identifies a set of program variables as input variables, and another set of program variables as output variables. The tool then identifies the statements that make up the computation that defines the output variables using the values of the input variables. In the approach described in [BG98], the programmer specifies a set of variables, and the tool provides a visualization of the data flow graph of the computations that depend on or define any of the specified variables. The programmer then uses the visualization to select a "root" node, and the tool identifies the set of nodes on which the root node depends as a candidate for procedure extraction. Both of these tools can identify statements that are widely separated from each other or are not extractable unless the program is restructured to some extent. They do not address the extractability issue in general, however, and presumably leave actual extraction to be performed by the programmer. Our work is complementary to theirs in the sense that our procedure-extraction algorithm could be applied after one of these tools identifies the statements to extract.

Automatic transformations on programs are discussed in [BD77, Fea82, PP96, BDFH97]. [Fea82] proposes a system that accepts a set of recursion equations, a starting expression and a goal pattern. The system determines if there is a way to rewrite the starting expression into an expression that satisfies the goal pattern by a series of simple transformation steps, where each transformation step is a use of the recursion equations to perform an operation like folding, unfolding, instantiation or abstraction on the expression in hand. Although procedure extraction was not an explicit goal of these transformation systems, it is likely that similar techniques could be used to move the statements in $\mathcal{M}$ together via a series of simple meaning-preserving transformation steps.

Much has been reported in the literature about converting unstructured programs to structured programs by eliminating goto's (e.g.,[LMW79, Ram88]). Our work is related to this work in the sense that it involves meaning-preserving program transformations, although the goals of the transformations are different. In a similar sense, our work is related to work done on optimizing transformations such as code motion out of loops and other program regions (e.g., [CLZ86]). A notable similarity in fact exists between the necessary condition checked by the "strict" approach (one that guarantees an improvement in execution time) described in [CLZ86] and the condition checked by step 1 of our algorithm; they too stipulate that a node be moved out only if its control dependence ancestors can also be moved out.

[FOW87] mention the use of chains of hammocks in an approach to enable easy generation of sequential code from a Program Dependence Graph. They suggest factoring the control dependence subgraph of the PDG into hierarchical chains of hammocks, and say that code can be generated for a chain in any order consistent with the data dependences between the hammocks. Their concern however being program optimization and vectorization, they do not talk about procedure extraction or how the main problem therein of moving together a given set of nodes can be mapped to the problem of finding a certain chain and permuting it under a certain set of constraints.

8 Conclusions and Future Work

We have defined an algorithm that moves a selected set of nodes in a CFG together so that they become extractable while preserving program semantics. The algorithm places no restrictions on the structure of the CFG or on the selected set of nodes. Although the algorithm has a worst-case exponential time complexity, experimental results indicate that it may work well in practice.

The algorithm succeeds in moving the selected nodes together if and only if certain data and control dependence properties hold. These properties guarantee that semantics will be preserved; however they do not necessarily hold in all instances where semantics preserving extraction is possible. Our future research plans include studying how often the algorithm succeeds in extracting meaningful methods from real programs.

A Appendix: Correctness Proofs

This appendix includes proof outlines for the two theorems stated in Section 6. (The full proofs are available in [KH99].) The first proof shows that it is possible to move the nodes in $\mathcal{M}$ together so that they are extractable (they form a hammock) while preserving control-dependence sets for all nodes in P iff the nodes in $\mathcal{M}$ are part of a chain $\mathcal{C}$ of atomic hammocks in P. This demonstrates that Step 1 of our algorithm (which fails if the nodes in $\mathcal{M}$ are not part of a chain of atomic hammocks) is reasonable. The second proof demonstrates that our algorithm is correct, by showing that any permutation of the hammocks in $\mathcal{C}$ that satisfies the ordering constraints imposed by $\mathcal{C}$ (as defined in Section 3.2) preserves semantics.

A.1 Necessity of a chain containing $\mathcal{M}$

Theorem 1 *: Given CFG P and set of nodes $\mathcal{M}$, it is possible to create a new CFG P_e with the same nodes as P, with each node having the same set of control dependences as in P, and in which the $\mathcal{M}$ nodes form a hammock iff there is a chain $\mathcal{C}$ of atomic hammocks in P such that:*

1. *$\mathcal{C}$ includes all of the nodes in $\mathcal{M}$, and*
2. *every hammock in $\mathcal{C}$ is either an $\mathcal{M}$-hammock – a hammock in which all nodes are in $\mathcal{M}$ – or an $\mathcal{O}$-hammock – one that has no nodes in $\mathcal{M}$.*

Proof Outline:

The proof is in three parts. First we show that if CFG P_e with the specified properties exists, then the following two conditions must hold for P:

1. For every predicate node p in $\mathcal{M}$, all nodes that are control dependent on p are also in $\mathcal{M}$.
2. All nodes in $\mathcal{M}$ that are (directly) control-dependent on some node outside $\mathcal{M}$ have the *same* control dependence set outside $\mathcal{M}$. In other words, if q and s belong to $\mathcal{M}$, and both are control dependent on nodes outside $\mathcal{M}$, and p is some predicate node outside $\mathcal{M}$, then q is C-control dependent on p if and only if s is C-control dependent on p, where C is either "true" or "false".

We say that $\mathcal{M}$ is *well-formed in control dependence in P* in this case.

Next we show that if $\mathcal{M}$ is well-formed in control dependence in P, then a chain $\mathcal{C}$ with the specified properties exists in P.

Finally, we show that if the chain $\mathcal{C}$ exists in P then the CFG P_e can be obtained from P by permuting $\mathcal{C}$. Clearly the three parts together prove the theorem.

A.1.1 Proof Part 1

We assume there exists a CFG P_e with the same nodes as P, with each node having the same set of control dependences as in P, and in which the $\mathcal{M}$ nodes form a $\mathcal{M}$-hammock $H_{\mathcal{M}}$. The strategy is to first show that $\mathcal{M}$ is well-formed in control dependence in P_e. In P_e, all the $\mathcal{M}$ nodes are inside $H_{\mathcal{M}}$ and all the nodes not in $\mathcal{M}$ are outside $H_{\mathcal{M}}$. It can be shown that no node outside a hammock can be control dependent on any node inside a hammock, and hence the first part of the control dependence well-formedness condition follows. For the second part, let q and s be nodes in $\mathcal{M}$ such that both are control dependent on nodes outside $\mathcal{M}$, and let p be a predicate outside $\mathcal{M}$ such that q is C-control dependent on p. It can then be shown that both q and s postdominate the entry node of $H_{\mathcal{M}}$, and that the entry node of $H_{\mathcal{M}}$ is C-control dependent on p. These in turn lead to the result that s too is C-control dependent on p, which gives us the second part of the condition for $\mathcal{M}$ being well-formed in control dependence in P_e.

Since control dependence well-formedness is a property based only on the control dependence sets of nodes, and we know that each node has the same control dependence set in P_e as in P, we can infer that $\mathcal{M}$ is also well-formed in control dependence in P.

A.1.2 Proof Part 2

Let $\mathcal{M}$ be well-formed in control dependence in P. We show that there then exists a chain of atomic hammocks $\mathcal{C}$ in P such that $\mathcal{C}$ includes all of the nodes in $\mathcal{M}$, and every hammock in $\mathcal{C}$ is either an $\mathcal{M}$-hammock or an $\mathcal{O}$-hammock.

We define a node $e \in \mathcal{M}$ to be an *entry node* of $\mathcal{M}$ if there is an edge into e from a node outside $\mathcal{M}$. We define *Region*(e) – the region of e – to be the set of nodes in $\mathcal{M}$ that can be reached from e along CFG paths that include only nodes in $\mathcal{M}$. By definition, $e \in$ *Region*(e). Our goal is to show that the region of each entry node is an $\mathcal{M}$-hammock in P, that these $\mathcal{M}$-hammocks belong to a chain, and that every $\mathcal{M}$ node belongs to one of the regions. First, we make the following observation about $\mathcal{M}$ and P, which follows from $\mathcal{M}$ being well-formed in control dependence: If e is an entry node and d is any of the predecessors of e not in $\mathcal{M}$, then any path from Enter to d includes a predicate node p not in $\mathcal{M}$ (p could be equal to d) such that e is control dependent on p. An implication of this is that all entry nodes have the same non-empty set of control dependences outside $\mathcal{M}$, which in turn leads to a result that there is a total ordering *Ord*$(\mathcal{E})$ on the set of all entry nodes $\mathcal{E}$ such that an entry node dominates all entry nodes after itself in *Ord*$(\mathcal{E})$ and is also postdominated by them.

Existence of $\mathcal{M}$-hammocks:
Let e be an entry node of $\mathcal{M}$. Any node outside *Region*(e) that has an edge coming into it from a node in *Region*(e) is said to be a first node outside *Region*(e). It can be shown that any first node outside *Region*(e) postdominates all nodes in *Region*(e) (this is because the first node outside *Region*(e) cannot be control dependent on any node in *Region*(e) – an implication of control dependence well-formedness). If there were more than one first node outside *Region*(e) each of them would have to postdominate the other which is not possible. As a result, there is a unique first node outside *Region*(e), which we will refer to as *outside*(e).

The regions of different entry nodes cannot intersect (and this implies that all edges into the region of an entry node from outside the region go into the entry node). For the sake of argument assume that two different entry nodes e_1 and e_2 have intersecting regions. Since any region has a unique first outside node, intersecting regions must have a common first outside node. Let d_2 be a predecessor of e_2 outside $\mathcal{M}$. Say e_1 precedes e_2 in *Ord*$(\mathcal{E})$, which means e_2 postdominates e_1 and e_1 dominates e_2. This implies the existence of a path S from e_1 to d_2 that does not pass through e_2 and all of whose nodes are postdominated by e_2. One of the nodes t on this path is surely equal to *outside*(e_1) (and hence equal to *outside*(e_2)), which implies a contradiction, namely t and e_2 postdominate each other.

We have thus shown that if e is an entry node of $\mathcal{M}$, then *Region*(e) is an $\mathcal{M}$-hammock with e being its entry node and *outside*(e) being its outside exit node. Since each $\mathcal{M}$ node is either an entry node itself or is reachable from an entry node along a path that goes through only nodes in $\mathcal{M}$, each $\mathcal{M}$ node belongs to the region of some entry node.

Consecutive regions and the nodes in between them form a chain:
Let e_i and e_j be two distinct nodes in $\mathcal{E}$ such that e_j immediately follows e_i in *Ord*$(\mathcal{E})$. *InBetween*(e_i, e_j) is defined to be the set of all nodes that are outside $\mathcal{M}$, and that can be reached from *outside*(e_i) along CFG paths that include only nodes that are outside $\mathcal{M}$. By definition, *outside*$(e_i) \in$ *InBetween*(e_i, e_j). Our goal is to show that *InBetween*(e_i, e_j) is an $\mathcal{O}$-hammock.

An important property that is a consequence of the fact that all entry nodes have the same non-empty set of control dependences outside $\mathcal{M}$ is that:

1. there exists a *direct path* (explained below) from *outside*(e_i) to e_j, and
2. there does *not* exist a direct path from *outside*(e_i) to any entry node other than e_j, nor does one exist from any first outside node other than *outside*(e_i) to e_j, and
3. *outside*(e_i) dominates e_j

where a direct path is one that does not include any entry nodes other than e_j. From this property we can derive three lemmas. The first is that any edge from a node in *InBetween*(e_i, e_j) to a node outside it goes into e_j (otherwise there would be a direct path from *outside*(e_i) to an entry node that is not e_j); the second is that all edges coming into e_j are either from *Region*(e_j) or from *InBetween*(e_i, e_j) (otherwise there would exist a direct path from *outside*(e_k), $k \neq i$, to e_j – this would be because *outside*(e_i) dominates the node that has the edge to e_j but any path from *outside*(e_i) to that node will have to pass through *outside*(e_j)). The third lemma says that any edge coming into a node in *InBetween*(e_i, e_j) from outside *InBetween*(e_i, e_j) comes from a node in *Region*(e_i) (the reason is similar to the one given for the second lemma).

From the first and third lemmas it is clear that *InBetween*(e_i, e_j) is an $\mathcal{O}$-hammock with its entry node being the same as *Region*(e_i)'s outside exit node, namely *outside*(e_i), and its outside exit node being e_j (which is the entry node of *Region*(e_j)). (*Region*(e_i), *InBetween*(e_i, e_j), *Region*(e_j)) in fact form a hammock chain, as the second and third lemmas say that all edges coming into e_j and *outside*(e_i) are from their respective previous hammocks in the chain.

Finding the chain $\mathcal{C}$:
Since any two consecutive regions and the nodes in between form a chain, we may conclude that (*Region*(e_1), *InBetween*(e_1, e_2), *Region*$(e_2), \ldots,$ *InBetween*(e_{m-1}, e_m), *Region*(e_m)) is the chain we seek, where *Ord*$(\mathcal{E}) = e_1, e_2 \ldots e_m$. (If any of the hammocks in the chain are non-atomic, they can be decomposed into chains of atomic hammocks.)

A.1.3 Proof Part 3

We assume that CFG P has a chain $\mathcal{C}$ such that each atomic hammock of $\mathcal{C}$ is either an $\mathcal{M}$-hammock or an $\mathcal{O}$-hammock and all the $\mathcal{M}$ nodes are contained in $\mathcal{C}$. Let $\mathcal{C}_m$ be a permutation of $\mathcal{C}$ such that the $\mathcal{M}$-hammocks of $\mathcal{C}$ occur contiguously in $\mathcal{C}_m$. Let P_e be the CFG obtained by replacing $\mathcal{C}$ in P with $\mathcal{C}_m$. We show that P_e has the properties that we seek:

1. Since P_e is obtained from P by permuting the chain $\mathcal{C}$ in P, clearly the node sets of the two CFGs are the same.

2. All the $\mathcal{M}$ nodes are contained in the $\mathcal{M}$-hammocks of $\mathcal{C}$ (that is given), and these $\mathcal{M}$-hammocks form their own contiguous subchain in $\mathcal{C}_m$. Since any chain of hammocks is itself a hammock, the $\mathcal{M}$ nodes form a hammock in P_e.

3. We show informally that each node has the same control dependence set in P_e as in P. Let p be a predicate node and q be any node such that q is C-control dependent on p in one of the CFGs P or P_e. We consider two cases for p. If p is inside $\mathcal{C}$ (and hence inside $\mathcal{C}_m$), then q must belong to the same atomic hammock as p as no node outside a hammock can be control dependent on a node inside the hammock. Since each atomic

hammock in $\mathcal{C}$ appears as such in $\mathcal{C}_m$ and vice versa, q will be C-control dependent on p in both CFGs. If p is outside the chain, then there are two subcases: if q is inside the chain then it must postdominate the entry nodes of both chains; if q is outside the chain then permuting the chain should have no effect on paths from p to q or p to Exit. Therefore in either subcase q will be C-control dependent on p in both CFGs.

A.2 Ordering constraints guarantee semantics preservation

Theorem 2 *: Given:*

1. *Chain $\mathcal{C}$ of atomic hammocks in CFG P, such that all nodes in $\mathcal{M}$ are in the chain, and every hammock is either an $\mathcal{M}$-hammock (containing only $\mathcal{M}$ nodes) or an $\mathcal{O}$-hammock (containing no $\mathcal{M}$ nodes), and*
2. *$\mathcal{C}'$, a permutation of $\mathcal{C}$ such that $\mathcal{C}'$ satisfies the ordering constraints imposed by $\mathcal{C}$ (as defined in Section 3.2), and*
3. *P', the CFG obtained by replacing chain $\mathcal{C}$ with the chain $\mathcal{C}'$ in P*

then: P and P' are semantically equivalent.

Proof Outline:

The proof depends on two key lemmas. The first says that the sets of live variables at chain exit and at chain entry are the same for P and P'. The second says that if control enters $\mathcal{C}$ in P and $\mathcal{C}'$ in P' in the same state (as defined in Section 2), then:

- at the entry point of every hammock H in $\mathcal{C}$ and $\mathcal{C}'$, the states in the two programs will be identical with respect to all variables that are upwards-exposed in H (variables that might be used in H before being defined), and
- at the chain exits, the states in the two programs will be identical with respect to all live variables.

Since P and P' differ only in the chains, these two lemmas can be shown to ensure identical semantics.

The proof of the first lemma involves first showing that $\mathcal{C}$ and $\mathcal{C}'$ have the same set of upwards-exposed uses (this follows from the constraints imposed by the data dependences). Given that fact, we argue that the sets of live variables at chain exit and chain entry must be the same: A variable is live at chain exit because there is a definition-free path to a use either outside the chain (and in that case, the same path exists in both P and P') or to a use inside the chain (and in that case, the fact that the two chains have the same sets of upwards-exposed uses ensures that such a path exists in both CFGs). A variable is live at chain entry because there is a definition-free path to a use either inside the chain (again, the fact that the two chains have the same sets of upwards-exposed uses ensures that such a path exists in both CFGs), or after the chain (in which case the variable is live at chain exit, and that case has already been covered).

The first part of the second lemma is proved using induction on the position of hammock H in $\mathcal{C}$. The second part is proved as follows: Consider constructing a new dummy hammock F such that F's upwards-exposed uses include all variables that are live just before the exit node t of the two chains. Given that $\mathcal{C}'$ satisfies the constraints imposed by $\mathcal{C}$, the chain $\mathcal{C}' + F$ satisfies the constraints imposed by chain $\mathcal{C} + F$. $\mathcal{C}' + F$ is obtained by appending hammock F to the end of the chain $\mathcal{C}'$, and $\mathcal{C} + F$ is obtained analogously. Now consider appending hammock F to the end of chain $\mathcal{C}$ in P and to the end of $\mathcal{C}'$ in P'. Using the property just stated and using part 1 of this lemma we infer that control enters F in both CFGs with the same values for all variables with upwards-exposed uses in F; i.e., all variables that are live at chain exit. This inference must clearly hold even if F were not there at the end of the two chains, and thus we have shown that control leaves $\mathcal{C}$ in P and in $\mathcal{C}'$ in P' with the same values for all variables that are live at that point.

The proof of the theorem concludes by arguing that the two key lemmas guarantee that P and P' are semantically equivalent; i.e, starting execution of P and P' in the same state, both procedures end with the same values of the variables that are live at the Exit node. The argument is as follows: It is clear that if for some starting state the path of execution does not flow through the chain $\mathcal{C}$ in P, then the path of execution would not flow through $\mathcal{C}'$ in P'. That is because the two procedures are identical except for the chain. Therefore the state when control reaches the Exit node will be identical in the two CFGs.

If the path of execution from Enter to Exit in P does flow through $\mathcal{C}$ for the given starting state S, then we can decompose the path into a sequence of subpaths as follows:

1. a path from Enter to e, the entry node of $\mathcal{C}$
2. a path through $\mathcal{C}$ from e to t, the outside exit node of $\mathcal{C}$
3. zero or more occurrences of the following sequence:
   (a) a path from t to e
   (b) a path through $\mathcal{C}$ from e to t
4. a path from t to Exit

Consider the execution path in P' for the same starting state S. It is clear that P' will initially follow a subpath from Enter to e', where e' is the entry node of $\mathcal{C}'$. Moreover this subpath is identical to subpath 1 above with identical changes resulting to the state. Control then follows a path through $\mathcal{C}'$ and this could be different from subpath 2 above. But the first lemma says that the set of live variables at chain entry is the same for both CFGs, and the same is true at chain exit. Using this fact and the second lemma, it is clear that although subpath 2 through the chain could be different in the two CFGs, in both cases control leaves the chain with the same values for the variables that are live at chain exit. Therefore, by repeating the argument on the rest of the execution path to Exit, it follows that execution reaches Exit in P and P' with identical values for variables live at that point.

References

[ABS94] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, January 1977.

[BDFH97] J. A. Bergstra, T. B. Dinesh, J. Field, and J. Heering. Toward a complete transformational toolkit for compilers. *ACM Transactions on Programming Languages and Systems*, 19(5):639–684, September 1997.

[BG98] R. W. Bowdidge and W. G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology*, 7(2), April 1998.

[BH93] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *ACM Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[CLZ86] R. Cytron, A. Lowry, and K. Zadeck. Code motion of control structures in high-level languages. In *ACM Symposium on Principles of Programming Languages*, pages 70–85, 1986.

[CY79] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice–Hall, Englewood Cliffs, New Jersey, 1979.

[Fea82] M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GN93] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[KH99] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. Technical Report TR-1407, Computer Sciences, University of Wisconsin-Madison, 1999.

[KKP$^+$81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[LD98] A. Lakhotia and J-C. Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(11–12):677–689, November 1998.

[LMW79] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice*. Addison–Wesley, Cambridge, Mass., 1979.

[LRZ93] W. Landi, B. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[LS86] S. Letovski and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, pages 198–204, May 1986.

[LV97] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–258, April 1997.

[Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PP96] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.

[Ram88] L. Ramshaw. Eliminating go to's while preserving program structure. *J. ACM*, 35(4):893–920, October 1988.

[RSW96] S. Rugaber, K. Stirewalt, and L. M. Wills. Understanding interleaved code. *Automated Software Engineering*, 3(1–2):47–76, June 1996.

[SJ87] H. M. Sneed and G. Jandrasics. Software recycling. In *Proc. Conf. Software Maintenance*, pages 82–90, 1987.

[WFW$^+$94] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29(12), pages 31–37, December 1994.

[WL95] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[WM92] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, January 1992.