Eliminating go to's while Preserving Program Structure

LYLE RAMSHAW

Digital Equipment Corporation Systems Research Center, Palo Alto, California

Abstract. Suppose we want to eliminate the local **go to** statements of a Pascal program by replacing them with multilevel loop **exit** statements. The standard ground rules for eliminating **go to**'s require that we preserve the flow graph of the program, but they allow us to completely rewrite the control structures that glue together the program's atomic tests and actions. The **go to**'s can be eliminated from a program under those ground rules if and only if the flow graph of that program has the graph-theoretic property named reducibility.

This paper considers a stricter set of ground rules, introduced by Peterson, Kasami, and Tokura, which demand that we preserve the program's original control structures, as well as its flow graph, while we eliminate its **go to**'s. In particular, we are allowed to delete the **go to** statements and the labels that they jump to and to insert various **exit** statements and labeled **repeat-endloop** pairs for them to jump out of. But we are forbidden to change the rest of the program text in any way. The critical issue that determines whether **go to**'s can be eliminated under these stricter rules turns out to be the static order of the atomic tests and actions in the program text. This static order can be encoded in the program's flow graph by augmenting it with extra edges. It can then be shown that the reducibility of a program's augmented flow graph, augmenting edges and all, is a necessary and sufficient condition for the eliminability of **go to**'s from that program under the stricter rules.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language constructs—*control structures*; F.3.3 [**Logics and Meanings of Programs**]: Studies of program constructs—*control primitives*

General Terms: Theory, Verification

Additional Key Words and Phrases: Block structure, **break** statement, **exit** statement, flow graph, **go to** statement, Mesa, Pascal, program transformations, reducibility, structured programming

1. *Introduction*

The **go to** statement was the center of much controversy 15 or 20 years ago [4, 11]. This controversy spurred several theoretical analyses of the power of the **go to** statement. Given a source program with **go to**'s, can we produce an equivalent target program that renounces **go to**'s in favor of more structured control constructs? The first step in tackling this question is to settle the ground rules: What is the precise meaning of "equivalent," and which control constructs are allowed in the target program? Different ground rules lead to different results, as summarized in Table I.

Some of this work was done while the author was employed by the Palo Alto Research Center of the Xerox Corporation and appears in the PARC technical report "Deftly Replacing **go to** Statements with **exit**'s," CSL-83-10 (Nov. 1983). The arrow-stretching technique for eliminating **go to**'s that underlies the sufficiency result was implemented at PARC as part of an effort to port the document compiler TEX from Pascal to Mesa.

Author's address: Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301.

TABLE I. NECESSARY AND SUFFICIENT CONDITIONS TO REPLACE THE SOURCE CONTROL CONSTRUCT WITH THE TARGET CONSTRUCT UNDER THE STATED POLICY

Policy	Source	Target	Condition
retain functional equivalence	**go to**'s	one **while** plus **if**'s	always
retain path equivalence	**go to**'s	multilevel **exit**'s	always
retain flow graph equivalence	**go to**'s	multilevel **exit**'s	when flow graph is reducible
retain structural equivalence	outward **go to**'s	multilevel **exit**'s	when augmented flow graph is reducible
use Elimination Rules only	outward **go to**'s	multilevel **exit**'s	when **go to** graph of every block has no head-to-heads

At the outset, let us agree to ignore the *nonlocal* **go to** statements of such languages as Pascal [8], where a **go to** may jump all the way out of a procedure body to a label in an enclosing procedure, hence causing the activation record of the inner procedure to be deallocated. In this paper, "program" means "procedure body," and all **go to**'s are local.

Eliminating **go to**'s is easy if we choose a lenient definition of equivalence. Two programs are called *functionally equivalent* [16] if, whenever they are given identical inputs, they produce identical outputs. In particular, the policy of functional equivalence does not rule out introducing new variables in the target program. By introducing one new variable that acts as a program counter, we can replace all of the control structures of any program, including its **go to**'s, with some conditional statements inside a single **while** loop. Harel discussed this result as an example of a folk theorem [5]; it is often credited to Böhm and Jacopini [3], somewhat inaccurately, as Harel points out.

Eliminating **go to**'s gets more challenging if we adopt a stricter notion of equivalence. Two programs are called *path equivalent* [2, 16] or *strongly equivalent* [15] if, given identical inputs, the dynamic sequences of atomic tests and actions that the two programs perform are identical. Knuth and Floyd [14] showed that there are source programs whose **go to**'s cannot be eliminated if we demand path equivalence and allow, in the target, only the three control constructs **begin-end**, **if-then-else**, and **while-do**. Some programming languages, such as Modula-2 [22], supplement these three with a "do forever" loop, for which we use the keywords **repeat-endloop**, and a single-level **exit** or **break** statement, a primitive that forces the immediate termination of the innermost enclosing **repeat** loop. (A tricky case: In Modula-2, an **exit** statement inside a **while** loop inside a **repeat** loop terminates the entire **repeat**, not just the **while**.) Unfortunately, as Kosaraju showed [15], there are still programs whose **go to**'s cannot be eliminated under path equivalence, even if we allow **repeat** loops and single-level **exit**'s in the target. But we can eliminate all **go to**'s under path equivalence if we allow ourselves a multilevel **exit** in the target, a primitive that can terminate any specified enclosing **repeat** loop [15, 18]. Ada[1] is an example of a language that includes multilevel **exit**'s [20]. Note that, once we allow **repeat** loops with multilevel **exit**'s, we no longer need to retain **while-do** as a separate primitive, since we can simulate a **while-do** by using a **repeat** loop whose first statement is a conditional **exit**. (This simulation does not always

[1] Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

work with the single-level **exit**'s of Modula-2 because of the tricky case mentioned in the previous parentheses.)

For completeness, we should note that some researchers chose to study a policy called *semantic equivalence* [16] or *weak equivalence* [15], which is just a bit more lenient than path equivalence. Semantic equivalence makes the assumption that the atomic tests are free of side effects and then loosens the rules by allowing the target program to perform redundant or useless evaluations of tests. From our point of view, adopting the policy of semantic equivalence gives the same results as adopting path equivalence: multilevel **exit**'s are as powerful as **go to**'s, but single-level **exit**'s are not [15].

The policy of path equivalence is permissive in the sense that it allows us to include, in the target program, as many copies of the tests and actions of the source program as we like; that is, it allows replicating code. The policy of *flow-graph equivalence* [2] or *very strong equivalence* [16] closes this loophole. Two programs are flow-graph equivalent if their flow graphs are the same, that is, there is a one-to-one correspondence between the atomic tests and actions of the two programs that respects control flow. (A fine point: Tests and actions in dead code should be ignored.) Under this policy, even multilevel **exit**'s are not powerful enough to replace all **go to**'s. A flow graph is called *reducible* if no simple cycle in it can be entered for the first time at two different places [6]. Reducibility precisely characterizes the power of the multilevel **exit** statement; that is, every program with **exit**'s has a reducible flow graph [6], and every reducible flow graph is the flow graph of some program with **exit**'s [1, 9, 18]. Hence, under flow graph equivalence, **go to**'s can be eliminated from a program if and only if that program's flow graph is reducible.

Two programs that are flow graph equivalent are essentially identical, from the point of view of the back end of a compiler. But the structurings of the tests and actions with **begin-end**, **if-then-else**, and the like in the two programs can be quite different. Most of the early eliminators of **go to**'s had no interest in trying to preserve this type of structure. In fact, many of them hoped that eliminating the **go to**'s from a program would be an automatic way to improve its structure [1]. It is an interesting theoretical challenge, however, to understand the interactions between **go to**'s and the other local control constructs. In this paper, we study the extent to which the original structure of a source program can be preserved while eliminating its **go to**'s.

Peterson et al. made a start on this problem [18]. They introduced a stricter notion of equivalence, one that demanded the preservation of the program's structure, as well as of its flow graph. We adopt their notion of equivalence for our investigations, christening it *structural equivalence.* A target program is structurally equivalent to a source program if the source and target have the same flow graph and we can convert the text of the source program into the text of the target simply by deleting all **go to** statements and the labels that they jump to and inserting various **exit** statements and appropriately labeled **repeat-endloop** pairs for them to jump out of, without rearranging or altering any other statements in any way. (We might have to delete or insert various colons and semicolons also, to keep the program syntactically valid.)

Note that the policy of structural equivalence allows us to bracket an existing sequence of statements with a new **repeat** and **endloop**, thus forming a new loop and, in some sense, adding more structure to the program—more levels to the parse tree at least. On the other hand, there is not much hope of eliminating

go to's in general if we are not allowed to create new **repeat** loops. Once **go to**'s are eliminated, **repeat** loops are the only construct that can effect a backward transfer of control, and the source program might not have any **repeat** loops. Thus, structural equivalence is about the strictest policy that leaves much chance of widespread success. To what extent can **go to**'s be eliminated under the policy of structural equivalence?

It is clear at the outset that some **go to** statements are less respectful of program structure than others. The bad ones are the **go to**'s that jump into the middle of some structured statement from outside it: those that jump into a compound statement or loop, those that jump into a branch of a conditional, and the like. Call such **go to** statements *inward*, and call the rest *outward*. Some languages, such as C [10], permit both inward and outward **go to**'s; others, such as Pascal [8] and Ada [20], permit only the outward ones. There is no hope of eliminating inward **go to**'s in general under structural equivalence. All of the structured control constructs, including multilevel **exit**'s, enforce the restriction that control can enter a statement only at its beginning; but inward **go to**'s allow control to enter a statement anywhere. We give up on inward **go to**'s right away.

Peterson et al. also gave up on inward **go to**'s, of course. Among outward **go to**'s, they focused on the forward ones—call a **go to** statement *forward* if the destination label follows the **go to** statement itself in the text of the program; else, call it *backward*. Peterson et al. gave a transformation that replaces all of the forward, outward **go to**'s to a particular label with **exit**'s by introducing one new **repeat** loop; we call this transformation the Forward Elimination Rule. By using this rule repeatedly, they were able to eliminate all forward, outward **go to**'s. (To be precise, they also handled certain backward, outward **go to**'s, but only a few of them: only unconditional backward **go to**'s whose destination label is at the same level of block nesting. Such **go to**'s can be replaced with **repeat-endloop** pairs in a trivial way.) We pick up where they left off.

Sections 2 through 4 develop a sufficient condition for the eliminability of **go to**'s under structural equivalence. In Section 2, we introduce two idealized programming languages called GOTO and EXIT: GOTO has outward **go to**'s, while EXIT has multilevel **exit**'s. Section 3 then presents the Forward Elimination Rule as a tool for translating from GOTO into EXIT under structural equivalence. Furthermore, it supplements the Forward Elimination Rule with an analogous Backward Elimination Rule, which handles backward, outward **go to**'s. Both of the Elimination Rules involve introducing new **repeat** loops; in certain cases, these requests for new loops can conflict with each other. In Section 4, we define the concept of a "head-to-head crossing" in the "**go to** graph" of a block. We then show that the two Elimination Rules can eliminate all of the **go to**'s from a GOTO program if and only if every block in that program has a **go to** graph that is free of head-to-head crossings. Thus, the absence of head-to-head crossings in the **go to** graph of every block is sufficient to guarantee the eliminability of **go to**'s under structural equivalence.

Sections 5 through 9 sharpen the analysis to yield a condition that is both necessary and sufficient. The static order of the atomic tests and actions in the text of the program turns out to be the key issue. After reviewing the standard notion of a flow graph in Section 5, we define in Section 6 an augmented type of flow graph in which there are edges encoding the static order of the atomic elements as well as the usual edges representing the dynamic flow of control. Section 7 then proves that the augmented flow graphs of EXIT programs are always reducible, augmenting edges and all. Since structurally equivalent programs have the same augmented flow graphs, this implies that the reducibility of the augmented flow graph is a necessary condition for the eliminability of **go to**'s under structural equivalence. Sections 8 and 9 show that the reducibility of the augmented flow graph is also a sufficient condition by presenting an algorithm for eliminating **go to**'s that first uses four phases of simple cleanups to get rid of any spurious head-to-head crossings and then applies the Elimination Rules as discussed in Section 4.

The last two sections address dangling issues. In Section 10, we point out that the graph-theoretic property of reducibility simplifies, in the special case of augmented flow graphs, to a rule prohibiting certain "conflicting" pairs of edges. Finally, Section 11 discusses how the ideas in this paper took root during a project to translate the sources for Donald E. Knuth's document compiler TEX from Pascal to Mesa. Williams and Chen also found that the desire to mechanically translate Pascal programs into other languages led them into developing an algorithm for eliminating **go to**'s [21]. But their algorithm is quite different from the one in this paper: theirs preserves only functional equivalence.

2. *The Languages GOTO, EXIT, and JUMP*

Local control structure is only one small part of a programming language. In order to avoid dealing with the other complexities of real languages, we work with two idealized languages called GOTO and EXIT, which distill the essence of outward **go to**'s and multilevel **exit**'s. It is convenient to define GOTO and EXIT as sublanguages of a single language called JUMP, which includes both **go to**'s and **exit**'s. In particular, a GOTO program is a JUMP program that has no **exit** statements, while an EXIT program is a JUMP program that has no **go to**'s.

Figure 1 gives the syntax of JUMP in Extended Backus–Naur Form, the extensions being the use of curly braces to denote indefinite repetition (zero or more times) and square brackets to denote optionality (zero or one times). For example, a block in JUMP is a sequence of zero or more statements, terminated with semicolons, in which each gap between statements, including the gaps before the first and after the last, is labeled with zero or more labels. We refer to the statements that form this sequence as the *top-level statements of* the block, to distinguish them from any smaller statements that might be nested inside them. (The semicolon after the last top-level statement may be omitted when the final gap has no labels.)

In a real programming language, the atomic actions are explicit commands to do something, such as assignment statements, input/output statements, and procedure calls. But JUMP is not, strictly speaking, a programming language; it is a language in which to write uninterpreted program schemata. Hence, the atomic actions in JUMP are uninterpreted action symbols, drawn from the set {action_1, action_2, . . .}. Similarly, the tests in JUMP are symbols from the set {test_1, test_2, . . .}. The tests in JUMP are not assumed to be free of side effects; thus, evaluating the same test twice in a row may give two different results.

JUMP has two kinds of jump statements—**go to**'s and **exit**'s—each with its own flavor of labels. If B is a block in a JUMP program, let us define the *gaps of B* to be the spaces between the top-level statements of B, including the space just before the first top-level statement and the space just after the last. A gap label in a JUMP program names the gap in which it appears, and a **go to** statement specifies its destination by giving a gap label. A loop label names the loop at the end of which

⟨program⟩ ::= **start** ⟨block⟩ **stop**
⟨block⟩ ::= {{ ⟨gap label⟩ : } ⟨statement⟩ ; }{ ⟨gap label⟩ : }[⟨statement⟩]
⟨statement⟩ ::= ⟨action⟩ | ⟨jump⟩ | ⟨conditional⟩ | ⟨compound⟩ | ⟨loop⟩
⟨action⟩ ::= action$_1$ | action$_2$ | ···
⟨jump⟩ ::= **go to** ⟨gap label⟩ | **exit** ⟨loop label⟩
⟨conditional⟩ ::= **if** ⟨test⟩ **then** ⟨block⟩ **endthen** [**else** ⟨block⟩ **endelse**] **fi**
⟨compound⟩ ::= **begin** ⟨block⟩ **end**
⟨loop⟩ ::= **repeat** ⟨block⟩ **endloop** { : ⟨loop label⟩ }
⟨test⟩ ::= test$_1$ | test$_2$ | ···
⟨gap label⟩ ::= A | B | C | ···
⟨loop label⟩ ::= $\mathscr{A}$ | $\mathscr{B}$ | $\mathscr{C}$ | ···

FIG. 1. The syntax of JUMP in Extended Backus–Naur Form.

it appears, and an **exit** statement specifies its destination by giving a loop label. Note that, with this postfix convention for loop labels, the two types of jumps behave the same: They both transfer control to the textual location of the corresponding label.

The nonstandard keywords **endthen** and **endelse** in a conditional statement are not essential, but they will make our construction of the "step graph" of a JUMP program in Section 5 work out more neatly.

The language JUMP has two context-sensitive syntactic restrictions, associated with labels. First, all of the gap labels that label gaps of a single block must be distinct. Second, every jump statement must be within the scope of a matching label, where the scopes of labels are defined as follows. If "G:" labels a gap of the block B in a JUMP program, the *scope* of that G is the entire block B, minus any blocks nested inside of B that also have a gap labeled G. Similarly, if ":$\mathscr{L}$" labels a **repeat** loop R in a JUMP program, the *scope* of that $\mathscr{L}$ is the block that forms the body of R, minus the bodies of any loops nested inside of R that are also labeled $\mathscr{L}$.

Based on these syntactic restrictions, we define the semantic effect of jump statements simply by saying that each jump transfers control to the matching label in whose scope it lies. Thus, the destination of "**go to** G" is the unique gap labeled G in the innermost enclosing block that includes any such gap, and there must be such a block. Note that JUMP allows only outward **go to**'s, not inward ones. Similarly, the destination of "**exit** $\mathscr{L}$" is the end of the innermost enclosing loop labeled $\mathscr{L}$.

There is a concept about **go to**'s and their destinations that will be helpful in what follows. Suppose that S is a statement in a JUMP program and that T is a **go to** statement contained in S. We say that T is *bound in S* if the destination label of T is also included within S; else, T is *free in S*. For example, in the nonsensical statement

begin go to G; **go to** H; G: **end**,

"**go to** G" is bound but "**go to** H" is free.

3. *The Forward and Backward Elimination Rules*

Suppose that only one of the gaps of a block B in a JUMP program is labeled, say by "G:", and that this gap is only gone forward to. The *Forward Elimination Rule*

```
                                            action1;
     action1;                               action2;
     action2;                               repeat
     if test3 then go to G endthen fi;        if test3 then exit ℒ endthen fi;
     action4;                                 action4;
     if test5 then go to G endthen fi;        if test5 then exit ℒ endthen fi;
     action6;                                 action6;
  G: action7;                                 exit ℒ;
                                            endloop :ℒ;
                                            action7;
```

FIG. 2. An example of the Forward Elimination Rule.

is a transformation, introduced by Peterson et al. [18], that eliminates all of the **go to**'s to G; Figure 2 gives an example. If JUMP allowed compound statements to be labeled and exited as well as loops, we could make the Forward Elimination Rule a little simpler by inserting **begin-end** instead of **repeat-endloop** and omitting the final "**exit** $\mathscr{L}$". But it is traditional to restrict **exit** to exiting only loops, and we stick with that tradition.

The end of the new loop in the Forward Elimination Rule has to go in the gap of B labeled G. But we have some choice about where to put the beginning of the new loop. In Figure 2, for example, we could have started the loop either one or two statements earlier. In general, we can place the keyword **repeat** in any gap of B that precedes the first top-level statement of B containing a **go to** whose destination is the gap G. If S is a top-level statement of B, note that an instance of "**go to** G" in S that is bound in S is irrelevant at the moment, since its destination is a different gap labeled G, a gap of some block nested inside of S. The relevant instances of "**go to** G" are those that are free in the top-level statements of B in which they appear. Thus, we apply the Forward Elimination Rule as follows: We choose a new loop label, say $\mathscr{L}$; we insert "**exit** $\mathscr{L}$; **endloop** :$\mathscr{L}$;" in the gap G; we insert "**repeat**" in some gap of B that precedes the first top-level statement containing a free "**go to** G"; and we replace with "**exit** $\mathscr{L}$" each "**go to** G" that is free in the top-level statement of B in which it appears.

If S is a top-level statement of B containing a free "**go to** G", note that it does not matter where that free **go to** lies inside of S. The new loop that we introduce must contain all of S in order to contain any part of S.

Suppose next that a block B has only one gap that is labeled, say by "G:", and that this gap is only gone backward to. We can eliminate all of the **go to**'s to G in an analogous way, as shown by example in Figure 3. We call this technique the *Backward Elimination Rule.* We could get away with only one new loop in this rule instead of two if JUMP had a multilevel **next** or **continue** statement, a primitive that could force a new iteration of a specified enclosing loop to begin immediately. But, once again, we shall avoid cluttering up JUMP with unnecessary features.

In the Backward Elimination Rule, it is the location of the beginning of the two new loops that is precisely determined: We must insert "**repeat repeat**" in the gap G. The terminating boilerplate, the phrase "**exit** $\mathscr{M}$; **endloop** :$\mathscr{L}$; **endloop** :$\mathscr{M}$;", can be placed in any gap of B that follows the last top-level statement of B containing a free "**go to** G".

In the general case, a block B in JUMP will have several labeled gaps and those gaps will be gone to from various places, both by forward **go to**'s and by backward ones. If possible, we would like to eliminate all of the **go to**'s to all of the gaps of B at once by using the Forward and Backward Elimination Rules. That is, we would like to group the top-level statements of B into new loops, some possibly nested

```
   action1;
G: action2;
   if test3 then go to G endthen fi;
   action4;
   if test5 then go to G endthen fi;
   action6;
   action7;
```

```
action1;
repeat
   repeat
      action2;
      if test3 then exit 𝓛 endthen fi;
      action4;
      if test5 then exit 𝓛 endthen fi;
      exit 𝓜;
   endloop :𝓛;
endloop :𝓜;
action6;
action7;
```

FIG. 3. An example of the Backward Elimination Rule.

inside of others, and then replace each **go to** to a gap of B with an **exit** of an appropriate new loop. If we apply the two Elimination Rules naively to an arbitrary block, the resulting requests to introduce new loops may conflict with one another. Two simple cases of conflict are shown in Figures 4 and 5.

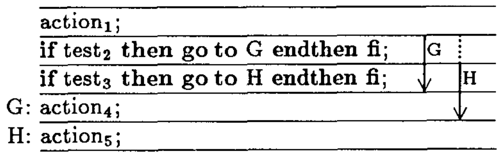
In Figure 4, we have two forward **go to**'s, the first of which jumps into the interior of the second, that is, into the middle of the region that the second jumps over. A naive attempt to apply the Forward Elimination Rule to both of these labels runs into a conflict. But this conflict can be easily resolved by making the new loop introduced for the label H longer than it would otherwise have to be—in particular, just long enough so that it completely contains the new loop for the label G. We call this process *stretching* the H loop.

Figure 5 shows a more stubborn case. The two **go to**'s here jump into each other's interiors. Remember that only one end of each new loop can be stretched. In this case, the stretchable ends are both on the outside. Whether we stretch them or not, these two requests for new loops will always remain in conflict.

Diagrams with lines and arrows, as in Figures 4 and 5, are quite helpful in studying conflicts between requests for new loops; we shall call them **go to** *graphs*. More formally, suppose that B is a block in JUMP, some of whose gaps are labeled. The **go to** *graph of B* is a directed graph (multiple edges allowed, but no self-loops). The vertices of the **go to** graph correspond to the gaps of B; they are drawn as horizontal lines in Figures 4 and 5. The edges, drawn as vertical arrows, are derived as follows. For each gap label G that is gone forward to, we add an arrow whose head is the gap labeled G and whose tail is the gap just before the first top-level statement of B containing a free "**go to** G". Similarly, for each gap label G that is gone backward to, we add an arrow whose head is the gap labeled G and whose tail is the gap just after the last top-level statement of B containing a free "**go to** G". Each arrow represents the extent of the shortest new loop that we could introduce to handle **go to**'s of that direction to that label.

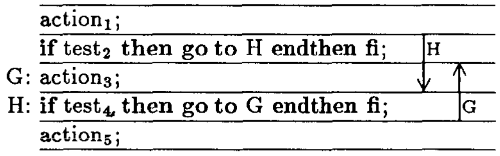
Stretching an arrow in a **go to** graph means moving its tail from one vertex to another so as to make the arrow longer. We draw a stretched arrow by a dotted extension of the arrow's shaft. The various stretched versions of an arrow correspond precisely to the various stretched loops that we could introduce to handle the corresponding **go to**'s. If there is some way to stretch the arrows that gets rid of all conflicts, we can use that stretching as a recipe for applying the Elimination Rules to eliminate all of the **go to**'s to gaps of B.

In general, a JUMP program has lots of blocks. Note that different blocks, whether disjoint or nested, don't interfere with each other at all while we are applying the Elimination Rules. For each block B, we build a **go to** graph, and we



```
   action1;
   if test2 then go to G endthen fi;
   if test3 then go to H endthen fi;
G: action4;
H: action5;
```

FIG. 4. A resolvable conflict.



```
   action1;
   if test2 then go to H endthen fi;
G: action3;
H: if test4 then go to G endthen fi;
   action5;
```

FIG. 5. An irresolvable conflict.

try to stretch its arrows to get rid of all conflicts. If we succeed in finding such a stretching, we group the top-level statements of B into new loops as specified by the stretched arrows, and we replace each **go to** whose destination is a gap of B by an equivalent **exit** of some new loop. This process does not affect any **go to**'s whose destinations are gaps of blocks other than B. It also does not affect any preexisting **exit**'s, unless we are foolish enough to choose, as the label for one of our new loops, an identifier that was already in use as a loop label. Therefore, we can translate an entire GOTO program into EXIT under structural equivalence, as long as we do not get stumped in stretching the arrows of the **go to** graph of some block. It behooves us to consider the combinatorial problem of stretching arrows.

4. *Stretching Arrows to Get Rid of Crossings*

Abstractly, a **go to** graph is a directed graph together with a total ordering on its vertices. Since self-loops are forbidden, each edge in a **go to** graph can be classified as either *forward* or *backward*. Each pair of edges can be classified as either *disjoint*, *nested*, or *crossing*, as shown in Figure 6. (In this section, we draw our **go to** graphs rotated 90 degrees.) There are four types of crossing pairs, depending upon the directions of the two arrowheads. We call them *forward-forward*, *backward-backward*, *head-to-head*, and *tail-to-tail*. Note that, if two edges share an endpoint, that pair should count as either disjoint or nested; in order for a pair to cross, four distinct vertices must be involved. To see that shared endpoints are not a problem, consider the stretched version of the **go to** graph in Figure 4: It is easy to begin both the G loop and the H loop in the same statement gap, as long as we begin them in the right order—first H and then G.

We want to stretch the edges of a **go to** graph so as to eliminate crossings. As we noted above, stretching moves cannot eliminate head-to-head crossings. But it turns out that stretching moves are powerful enough to eliminate all other crossings.

PROPOSITION 4.1. *Given a* **go to** *graph that is free of head-to-head crossings, it is possible to stretch its arrows so as to eliminate all crossings of any kind.*

To see that Proposition 4.1 is not trivial, consider the situation in Figure 7. If we were to stretch the tail of the forward arrow f back past the head of the backward arrow b, we would turn a nested pair into a head-to-head crossing. No further stretching would ever be able to eliminate that head-to-head. Thus, we cannot eliminate all crossings just by stretching arrows heedlessly.

On the other hand, Proposition 4.1 is not very subtle either. As long as we are careful not to introduce any head-to-head crossings, stretching arrows heedlessly

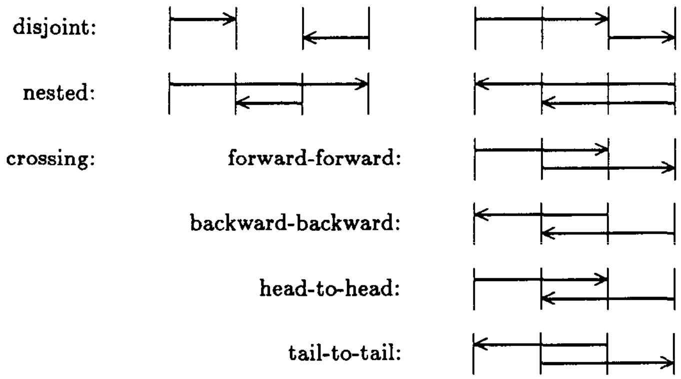


FIG. 6. Classifying pairs of edges in a **go to** graph.

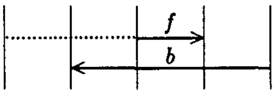


FIG. 7. A head-to-head crossing caused by careless stretching.

does just fine. This observation is due to Susan Owicki (personal communication), and it forms the basis of her simpler proof of my Proposition 4.1.[2]

LEMMA 4.2. *If a* **go to** *graph contains at least one crossing but no head-to-head crossings, then some arrow in it can be stretched without introducing any head-to-head crossings.*

If we can prove Lemma 4.2, Proposition 4.1 follows at once: just keep applying the lemma as often as possible. There are only a finite number of states that we could ever reach by stretching moves, since the **go to** graph contains only a finite number of vertices and edges. Furthermore, every stretching move increases the sum of the lengths of the edges; therefore, we will not be able to keep applying Lemma 4.2 forever. When it no longer applies, the graph must be free of all crossings.

To prove Lemma 4.2, we consider several cases. First, suppose that at least one of the crossings that remains is forward-forward, like the crossing of f and g in Figure 8. In such a situation, we stretch the tail of the arrow f back to meet the tail of g. Could this stretching introduce any head-to-head crossings? Any arrow that crossed the stretched f but not the original f would have to have one endpoint in the interval $(u, w]$ and the other either less than u or greater than y. In order for such a hypothetical arrow to end up head-to-head with the stretched f, it would have to be a backward arrow b with its tail at some vertex z with $z > y$ and its head at some vertex v with $u < v \leq w$. But no such arrow b can exist, since such an arrow would have already been in head-to-head conflict with g. Thus, even though stretching f may introduce new crossings of various kinds, it can not possibly introduce any head-to-head crossings.

If any backward-backward crossings remain, we proceed symmetrically.

If there are crossings, but none of them are head-to-head, forward-forward, or backward-backward, they must all be tail-to-tail. In this case, we have a lot of choice about what to do. To keep things simple, we shall pick one tail-to-tail pair

[2] Susan Owicki also deserves credit for the excellent idea of drawing the vertices of a **go to** graph as parallel lines rather than as dots.

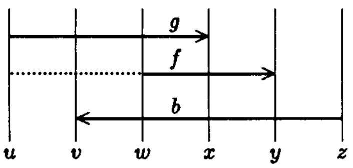
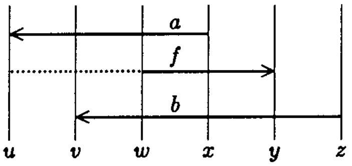


FIG. 8. The case of a forward-forward crossing in the proof of Lemma 2.

FIG. 9. The case of a tail-to-tail crossing in the proof of Lemma 2.



of arrows, call them a and f as in Figure 9, and stretch the tail of f back to the head of a. If we wanted, we could stretch a instead, or we could stretch both a and f—it doesn't matter. The advantage of stretching f is that it makes this case very similar to the forward-forward case. Could stretching f introduce any head-to-heads? By the same argument as before, the only type of arrow that might cause trouble is a backward arrow like b, with its tail at a vertex z with $z > y$ and its head at a vertex v with $u < v \le w$. But, if such an arrow b existed, it would form a backward-backward pair with a, and we would have employed the backward-backward case instead. Note that stretching f might very well introduce new crossings of other types, such as forward-forward crossings between the stretched f and arrows like the reverse of b. But stretching f cannot introduce any head-to-heads, and that is enough to complete the proofs of Lemma 4.2 and of Proposition 4.1. □

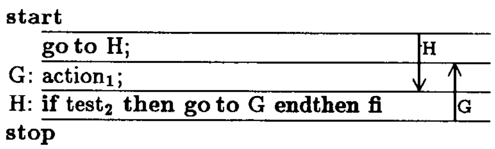
COROLLARY 4.3. *Let P be a program in GOTO or, more generally, in JUMP. The Forward and Backward Elimination Rules suffice to eliminate all of the* **go to***'s from P, producing a structurally equivalent EXIT program, if and only if the* **go to** *graphs of all of the blocks of P are free of head-to-head crossings.*

Disclaimer: The presence or absence of head-to-head crossings is not the key to writing well-structured programs in GOTO. In fact, it seems quite clear that backward-backward pairs are at least as bad as head-to-head pairs in terms of program structure—probably worse: Jumping backward into the middle of a loop seems at least as ill-advised as jumping forward into the middle of one. Good program structure is too subtle a property to be captured by any simple test.

5. *Step Graphs, Flow Graphs, and Reducibility*

It is time for a major change of focus. Corollary 4.3 gives a sufficient condition for the eliminability of **go to**'s under structural equivalence. In this section, we begin the groundwork for a condition that is both necessary and sufficient by constructing the flow graph of a JUMP program and reviewing some standard results about the reducibility of flow graphs.

Warning: There is some risk of confusion between the **go to** graphs that we have been using so far and the step graphs and flow graphs that we are about to define, since they are all directed graphs somehow associated with a program. But there are important differences. First, every block in a JUMP program has its own **go to** graph; but the program as a whole has just one step graph and one flow graph—



```
start
   go to H;
G: action₁;
H: if test₂ then go to G endthen fi
stop
```

FIG. 10. The GOTO program While.

not one per block. Second, a vertex in a **go to** graph represents a gap of a block; but a vertex in a step graph or flow graph represents a token of the program—not a space between two adjacent tokens. To avoid confusion, we reserve the words *vertex* and *edge* for use only when talking about **go to** graphs. When talking about step graphs and flow graphs, we use the words *node* and *arc* instead, along with *path* and *cycle.*

A *flow graph* is a labeled, directed graph (multiple arcs and self-loops both allowed) with certain properties. The nodes come in three classes: initial, atomic, and final. Every flow graph must have precisely one *initial* node, which must be labeled **start**. This node must have in-degree zero and out-degree one; the unique arc leaving the node labeled **start** must be unlabeled. A flow graph may have one or more *atomic* nodes. Each atomic node must be labeled with a symbol from the set {action_1, action_2, . . . , test_1, test_2, . . . }. An atomic node labeled "action_i" must have precisely one outgoing arc, which must be unlabeled. An atomic node labeled "test_i" must have precisely two outgoing arcs, one labeled "true" and the other labeled "false". A flow graph may have a node labeled **stop**, and it may have a node labeled **spin** (at most one of each). These *final* nodes, if they exist, must have out-degree zero. Finally, every node in a flow graph must be accessible by a path from the initial node, the node labeled **start**.

In what follows, we write a node by putting square brackets around its label. Thus, [**start**] denotes the initial node, and [action_1] denotes some node labeled "action_1".

Our definition of flow graphs is standard in most respects. In particular, we are following standard practice by demanding that all nodes of the flow graph be accessible from [**start**], that is, that any dead code in a program is not represented in that program's flow graph. Our treatment of [**start**] itself is rather unusual, however. Most definitions of flow graphs indicate where the computation is to begin by marking one of the atomic nodes as initial, rather than by adding a separate initial node. We need a separate initial node in the next section to anchor the beginning of the augmenting path. Our final node [**spin**] is also nonstandard. We use it to represent a computation that has become trapped in an infinite loop without any atomic tests or actions, like the loops in the statements "**begin** G: **go to** G **end**" or "**repeat endloop**". We refer to such loops as *spin cycles.*

The semantics of JUMP associates a unique flow graph with each JUMP program, and there are no particular surprises in the rules for constructing that flow graph. We formalize those rules as a two-step process. Given a JUMP program, we first build a more detailed graph called the *step graph* of the program; the step graph is to the flow graph as an interpreter is to a compiler. We then collapse the step graph into the flow graph by a process of path compression.

Figure 10 shows an example GOTO program, Figure 11 shows its step graph, and Figure 12 shows its flow graph. We name this example program *While,* since its flow graph is the same as that of the non-JUMP program "**while** test_2 **do** action_1". Figure 10 also shows the **go to** graph of the outer block in While; for future reference, note that it includes a head-to-head crossing.

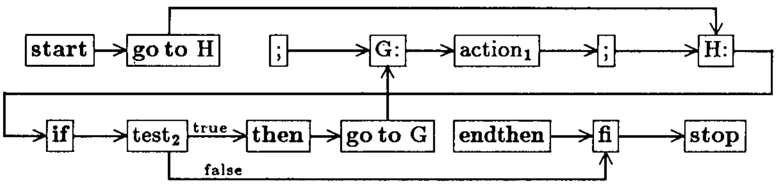


FIG. 11. The step graph of the GOTO program While.

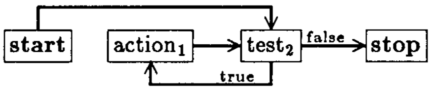


FIG. 12. The flow graph of the GOTO program While.

The *step graph* of a JUMP program is a labeled, directed graph whose nodes are almost exactly in one-to-one correspondence with the tokens of the JUMP program. The only exception to this one-to-one correspondence is that labels do not have nodes of their own: They are grouped together either with the adjacent colon or with the preceding **go to** or **exit**.

The nodes of the step graph are partitioned into four classes: initial, atomic, final, and glue. The nodes [**start**] and [**stop**], corresponding to the first and last tokens, are the unique *initial* and *final* nodes. In particular, step graphs do not have [**spin**] nodes. The nodes [action$_i$] and [test$_i$] are *atomic* nodes. All the other nodes are *glue* nodes: [**if**], [**then**], [**endthen**], [**else**], [**endelse**], [**fi**], [**begin**], [**end**], [**repeat**], [**endloop**], [;], [X:], [**go to** X], [:$\mathscr{X}$], and [**exit** $\mathscr{X}$], where X stands for any gap label and $\mathscr{X}$ for any loop label.

The arcs of the step graph also come in four types: sequential, conditional, loopback, and jump. *Sequential arcs* are unlabeled, and they go from one node to the node for the following token. Most nodes in a step graph have a single outgoing arc, which is sequential. This rule applies to the initial node [**start**], to all atomic action nodes, and to all glue nodes of the form [**if**], [**then**], [**else**], [**fi**], [**begin**], [**end**], [**repeat**], [;], [X:], or [:$\mathscr{X}$]. *Conditional arcs* are those that take control to the proper branch of a conditional statement and then collect it again when that branch is done. Each atomic test node has two outgoing arcs, which are conditional: one is labeled "true" and goes to the matching [**then**], the other is labeled "false" and goes to the matching [**else**]. If the **else-endelse** branch of the conditional is missing, the false arc goes to the matching [**fi**] instead. Nodes of the form [**endthen**] and [**endelse**] have one outgoing arc: an unlabeled, conditional arc to the matching [**fi**]. Each node of the form [**endloop**] has a single unlabeled, outgoing arc to the matching [**repeat**]; such arcs are called *loopback arcs*. Nodes corresponding to jump statements have one unlabeled, outgoing arc to the destination label node: either from [**go to** X] to [X:] or from [**exit** $\mathscr{X}$] to [:$\mathscr{X}$]; these arcs are called *jump arcs*. Which instance of [X:] or [:$\mathscr{X}$] is the correct destination of the jump is determined by the scoping rules in Section 2. Finally, the node [**stop**] has no outgoing arcs.

Unlike a flow graph, a step graph may contain nodes that are not accessible from [**start**]. The accessible nodes are called *live*, and the rest are called *dead*. The step graph in Figure 11 has two dead nodes, both of which are glue nodes: the first [;] and the [**endthen**].

In our future work, it is important to know how the arcs of a step graph interact with the syntactic structure of the associated program. If S is a set of nodes in some graph, an arc *enters S* if its head is in S but its tail is not; an arc *leaves S* if its tail is in S but its head is not.

LEMMA 5.1. *Let A be either a statement or a block in a JUMP program P. An arc in the step graph of P that enters A must be a sequential arc. An arc that leaves A must be either a sequential arc or a jump arc.*

PROOF. Conditional arcs and loopback arcs neither enter nor leave any statement or block. By the scoping rules of JUMP, jump arcs may leave statements and blocks, but they cannot enter them. □

By compressing paths through glue nodes in the step graph of a JUMP program into single arcs, we can build a flow graph for that JUMP program. The nodes of the compressed graph are precisely the live, nonglue nodes of the step graph, plus possibly a node [**spin**]. The arcs of the compressed graph correspond to paths in the step graph from one live nonglue node to another through a sequence of glue nodes. Since all glue nodes have out-degree one, there is never any choice about how to build such a path. There is the possibility, however, that we might get caught in a *spin cycle*, a cycle in the step graph that consists entirely of glue nodes. If this ever happens, we add a single node labeled **spin** to the compressed graph, and we represent any path in the step graph that enters any spin cycle as an arc to [**spin**] in the compressed graph. Given a JUMP program, it is straightforward to verify that the result of compressing its step graph is a valid flow graph; we refer to this compressed graph as *the flow graph of* the JUMP program.

The following easy proposition verifies that **go to**'s are powerful enough to encode any flow graph.

PROPOSITION 5.2. *Every flow graph is the flow graph of some GOTO program.*

PROOF. Given any flow graph, label all of its nodes other than [**start**] with distinct gap labels. For each node, we can write a fragment of GOTO program that captures the computation at that node. If the node [action$_i$] is labeled G and its outgoing arc goes to the node labeled H, we write "G: action$_i$; **go to** H;". If the node [test$_i$] is labeled G and its successors are labeled H and K, we write "G: **if** test$_i$ **then go to** H **endthen else go to** K **endelse fi**;". If a node [**spin**] is present and labeled S, we write "S: **repeat endloop**;". Finally, we concatenate these fragments in any order and make them into a GOTO program by preceding them with "**start go to** A;" and following them with "Z: **stop**", where A is the label on the successor of [**start**] and Z is the label on [**stop**] itself. (If the flow graph does not contain a node [**stop**], we omit the label "Z:".) □

Unlike programs in GOTO, programs in EXIT cannot have arbitrary flow graphs. As we noted in Section 1, previous research has shown that the flow graphs of EXIT programs are precisely those that have the graph-theoretic property called *reducibility*. The concept of reducibility is blessed with many equivalent definitions [7]. We shall use a definition that restricts the ways in which cycles in the graph can be first entered. A cycle in a flow graph is called *simple* if it does not visit any node more than once. If v is a node on a simple cycle C, we call v a *gateway to C* if there is some path from [**start**] to v that avoids all other nodes of C; that is, it is possible to enter C for the first time at v. A flow graph is called *reducible* if no simple cycle has more than one gateway.

Beware: A gateway to a cycle is not the same thing as the head of an arc that enters that cycle. Every gateway is also the head of an entering arc, but not every

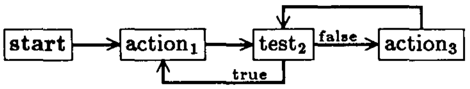


FIG. 13. A reducible flow graph with a two-entry cycle.

head of an entering arc is a gateway. As Hecht and Ullman noted [6], reducible flow graphs, like the example in Figure 13, can have cycles with multiple entry nodes: Both of the nodes in the cycle consisting of [action$_1$] and [test$_2$] are heads of entering arcs. But only [action$_1$] is a gateway.

6. *Augmented Flow Graphs*

The GOTO program While in Figure 11 exemplifies our current ignorance about eliminating **go to**'s under structural equivalence. The **go to** graph of the outer block in While has a head-to-head crossing; therefore, the Elimination Rules by themselves are not powerful enough to eliminate the **go to**'s from While. On the other hand, the flow graph of While, shown in Figure 12, is reducible. Therefore, we know from the standard theory that the **go to**'s of While can be eliminated under flow graph equivalence, that is, there do exist EXIT programs with the same flow graph as While. The simplest such program is

```
start repeat if test₂ then action₁ endthen
                        else exit 𝒮 endelse fi endloop :𝒮 stop.
```

But this EXIT program is not structurally equivalent to While. Can the **go to**'s be eliminated from While under structural equivalence, or not?

It turns out that they cannot. In the text of While, the atomic element "action$_1$" comes before "test$_2$". We prove in the next section that the test comes before the action in every EXIT program with the same flow graph as While. Since the static order of the atomic elements is one of the flavors of structure preserved by structural equivalence, the **go to**'s in While cannot be eliminated under that policy.

The main tool that we use to carry out this proof is a way of encoding a linear order on the atomic nodes of a flow graph into the graph structure itself. Define an *augmented flow graph* to be a labeled, directed graph whose arcs are partitioned into two classes: *dynamic* and *static*. If the static arcs are deleted, what remains must be an ordinary flow graph. In addition, the static arcs must form a path, called the *augmenting path*, that begins at the node [**start**] and then visits each atomic node exactly once. The augmenting path must not visit any final nodes. The augmenting path encodes a linear order on the atomic nodes of the flow graph by the order in which it visits them.

The particular linear order that we are interested in encoding is the static order of the corresponding atomic elements in the text of a program. If P is any JUMP program, we associate an augmented flow graph with P as follows, naming it *the augmented flow graph of P*. First, construct the normal flow graph of P by compressing paths in the step graph of P. Then, add static arcs to form an augmenting path that visits the atomic nodes of the flow graph in the same order that the corresponding live atomic elements occur in the text of P. For example, Figure 14 shows the augmented flow graph of the GOTO program While, with its two static arcs drawn dashed. Note that this graph is not reducible; both of the nodes in the cycle consisting of [action$_1$] and [test$_2$] are gateways. Our claim above that the test must precede the action in every EXIT program with the same flow graph as While will follow from the proof in the next section that every EXIT program has a reducible augmented flow graph.

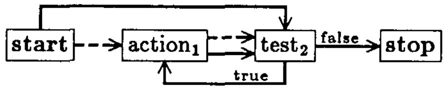


FIG. 14. The augmented flow graph of the GOTO program While.

```
start
  repeat
    repeat
      repeat
        exit $\mathcal{L}$;
        action$_1$;   } dead code
        exit $\mathcal{M}$;
      endloop :$\mathcal{L}$;
      action$_2$;
      exit $\mathcal{M}$;
    endloop :$\mathcal{M}$;
    action$_3$;
  endloop
stop
```

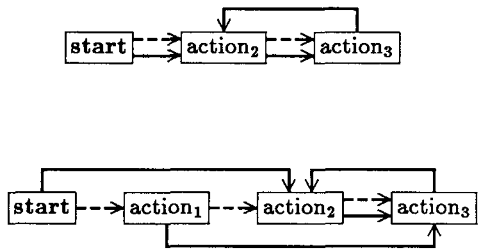


FIG. 15. An EXIT program, its augmented flow graph, and the nonreducible graph that would result if dead code were included in augmented flow graphs.

Note that the nonreducibility of the augmented flow graph of While depends upon the existence of [**start**] as a separate node. That is why we defined our flow graphs to have separate initial nodes, rather than having one of the atomic nodes marked as the place to start.

There is another fine point in our definitions that deserves comment: the treatment of dead code. Our definition of an augmented flow graph states that, if all the static arcs are deleted, the remaining graph must be an ordinary flow graph. In particular, this means that each node in an augmented flow graph must be accessible from [**start**] by a path consisting entirely of dynamic arcs. An alternative would have been to include all the dead code in the augmented flow graph, trusting to the augmenting path to make the dead nodes accessible from [**start**]. The problem with this alternative is that some EXIT programs would then have nonreducible augmented flow graphs; Figure 15 gives an example.

Even when we care about the static order of the atomic elements, **go to**'s remain an all-powerful control construct.

PROPOSITION 6.1. *Every augmented flow graph is the augmented flow graph of some GOTO program.*

Proof. Assemble the program fragments used in the proof of Proposition 5.2 in the order specified by the augmenting path. □

Just how powerful **exit**'s are when we care about the static order of the atomic elements is the issue addressed by the next three sections. We close this section with some remarks about the concept of *forward* and *backward.*

The nodes of the step graph of a JUMP program have a natural linear order, coming from the order of the corresponding tokens in the program text. Hence, just as the edges in a **go to** graph can be classified as either *forward* or *backward,* we can similarly classify the arcs in a step graph. Sequential arcs, conditional arcs, and the jump arcs that come from **exit**'s are all forward; loopback arcs are all backward; **go to** arcs can be either forward or backward.

If we have constructed a flow graph or an augmented flow graph from a JUMP program, we can use the static order of the live atomic elements in that program to define an analogous notion of forward and backward. Two technical points arise. First, we make the convention that any arc whose head is [**spin**] is *forward.* This case is special because **spin** is not a token in the program text. Second, we agree that all self-loops, that is, all arcs of the form $p \longrightarrow p$, are *backward.* This convention guarantees that every cycle in a flow graph or an augmented flow graph must include at least one backward arc.

On the other hand, we might have a flow graph or an augmented flow graph without any associated program. In the augmented case, we can still define forward and backward arcs by considering the order of their heads and tails along the augmenting path. For an ordinary flow graph with no associated program, however, there is no textual notion of forward and backward. It is worth mentioning that there is a related graph-theoretic notion, even though we do not need it in this paper. Call an arc in a flow graph *graph-backward* if its head dominates its tail, that is, if every path from [**start**] to its tail passes through its head; otherwise, call it *graph-forward.* In the flow graphs of EXIT programs, it turns out that an arc is graph-backward if and only if it goes backward in the program text, that is, the graph-theoretic and textual notions of forward and backward agree. The graph-theoretic notion is important because it also turns out that a flow graph is reducible if and only if it has no cycles consisting entirely of graph-forward arcs [7].

7. *Exploiting the Structure of* **exit**'*s*

Our goal in this section is to prove that the augmented flow graph of any EXIT program is reducible. We begin with two technical definitions about step graphs: *precursor* and *arrival node.*

Every block in a JUMP program is immediately preceded by either **start**, **then**, **else**, **begin**, or **repeat**; and every statement is immediately preceded either by one of those, by ";", or by "X:". If A is either a block or a statement in a JUMP program P, we refer to the node in the step graph of P that corresponds to the token immediately preceding A as the *precursor* of A. Note that precursor nodes are either initial nodes or glue nodes. From Proposition 5.1, we know that the only arc that can enter A is the sequential arc whose tail is the precursor of A. (This arc actually does enter A, except in the trivial case where A is an empty block.)

If u is an initial or glue node in the step graph of a JUMP program, consider the path leaving u. This path is uniquely determined as long as it travels through glue nodes, since all glue nodes have out-degree one. If the path leaving u visits some glue node twice before visiting any nonglue node, it will be trapped forever in a spin cycle; in this case, we say that *u never arrives.* Otherwise, we say that *u arrives at v*, or that v is the *arrival node* of u, where v is the first atomic or final node on the path leaving u.

The following lemma is more general than necessary for our current needs; the extra generality will be helpful when we use this lemma again in Section 9.

LEMMA 7.1. *Let S be a statement in a JUMP program P with the property that every* **go to** *statement included in S is free in S. If S contains any live atomic elements, then, in the step graph of P, the precursor of S arrives at the leftmost live atomic element in S.*

PROOF. By induction on the size of the statements in the program P, we may assume that the result of the lemma holds for all statements smaller than S, in particular, for all statements T nested inside of S.

Let u be the precursor of S, and let x be the leftmost live atomic element in S; our goal is to show that u arrives at x. The proof splits into five cases, depending upon the form of the statement S: jump, action, conditional, compound, or loop.

The first three cases are easy. Since S contains the live atomic element x, S cannot be a jump statement. If S is an action, we have $x =$ [action$_i$]; if S is a conditional, we have $x =$ [test$_i$]. In either case, the precursor u of S will arrive at x.

The last two cases are compounds and loops. Let $T_1, T_2, \ldots$ denote the top-level statements of the block that forms the body of S, and suppose that x lies in T_k. Since x is a live atomic element, there must be a path in the step graph of P from **[start]** to x; let α be such a path. At some point, the path α must enter S. By Lemma 5.1, each time α enters S, it must do so by traversing the sequential arc $u \rightarrow v$, where v is the leftmost node in S (either $v =$ **[begin]** or $v =$ **[repeat]**). If we partition α the last time that it traverses this arc, we have

$$\alpha = \textbf{[start]} \xrightarrow{*} u \rightarrow v \xrightarrow[\beta]{*} x,$$

where β remains entirely inside of S. We will be done if we can show that x is the first nonglue node on β.

The first thing that β does is to sequence through zero or more gap labels into T_1. Let us assume for the moment that $k > 1$. This means that T_1 has no live atomic elements; hence, β must leave T_1 again before it visits any nonglue node. By Lemma 5.1, control must leave T_1 either by a sequential arc or by a jump arc. Control cannot leave T_1 by a **go to** arc, because every **go to** statement in S is free in S and would hence take control outside of S. Control also cannot leave T_1 by an **exit** arc, because any **exit** arc that left T_1 would either leave S also or else, in the case that S itself is the loop being exited, would transfer control to a loop label at the very end of S, after which control would sequence out of S. Therefore, the only way that control can leave T_1 is by sequencing out. After passing through the semicolon and zero or more gap labels, the path β will then enter T_2. We can argue similarly for $T_2, \ldots, T_{k-1}$, until control reaches the precursor of T_k.

We now apply the inductive hypothesis to T_k. Since every **go to** statement in S is free in S, it follows that every **go to** statement in T_k is free in T_k. Furthermore, we know that T_k contains at least one live atomic element, since T_k contains x. By induction, the precursor of T_k must arrive at the leftmost live atomic element in T_k, which is x. Therefore, u also arrives at x. □

LEMMA 7.2. *Let S be a statement in an EXIT program P, and suppose that S contains at least one live atomic element. Every arc in the augmented flow graph of P that enters S has the leftmost live atomic element in S as its head.*

PROOF. Let x denote the leftmost live atomic element in S. An arc in the augmented flow graph of P that enters S must be either static or dynamic. Precisely one static arc enters S, and its head is x. A dynamic arc that enters S is the result of collapsing a path through glue nodes in the step graph. By Lemma 5.1, this path must enter S by traversing the sequential arc leaving the precursor of S. Since S has no **go to** statements at all, Lemma 7.1 tells us that the precursor of S arrives at x. Hence, every dynamic arc that enters S has x as its head. □

PROPOSITION 7.3. *Every simple cycle in the augmented flow graph of an EXIT program contains precisely one backward arc, and the head of that arc is the unique* *gateway to the cycle. In particular, the augmented flow graph of every EXIT program is reducible.*

PROOF. Fix an EXIT program P, and consider some simple cycle C in the augmented flow graph of P. Since C is a cycle, it must contain at least one backward arc. We show that the head of any backward arc in C is the unique gateway to C. Since C is simple, this will imply that C has only one backward arc.

Let $v \longrightarrow u$ be any backward arc in C, which means that u and v are atomic nodes with $u \le v$. Since every backward arc is dynamic, the arc $v \longrightarrow u$ is the result of collapsing a path in the step graph of P from v to u through a sequence of glue nodes; let α denote this path. The path α must include at least one backward arc. Since the program P has no **go to**'s, the only backward arcs in the step graph of P are loopback arcs; α must include one or more of them. In fact, one of those loopback arcs must transfer control from an [**endloop**] to the right of v back to a [**repeat**] to the left of v. Let R denote the loop in P delimited by this **repeat-endloop** pair, and note that v lies in R.

Apply Lemma 7.1 to the loop statement R. Since v lies in R, the loop R does contain a live atomic element. Since R is a statement in an EXIT program, it has no **go to**'s at all. Hence, we may conclude that the precursor of R arrives at the leftmost live atomic element in R. The path from the precursor of R to the leftmost live atomic element in R passes through the node [**repeat**$_R$], and the path α from v to u also passes through [**repeat**$_R$]. Since both paths are paths through glue nodes, we deduce that u must be the leftmost live atomic element in R.

Furthermore, we claim that every node of the entire cycle C must lie in R. If not, let w be a node of C outside of R whose successor along the cycle C lies in R. Note that $w \neq v$, since v lies in R. The arc in the augmented flow graph from w to its successor along C enters R; by Lemma 7.2, its head must be the leftmost live atomic element in R, which is u. But u cannot be the successor of w, since u is already the successor of v and C is a simple cycle.

Since every node of C lies in R, any path from [**start**] to any node of C must enter R. By Lemma 7.2, the head of the first arc on such a path that enters R must be u. Therefore, u is the unique gateway to C. ☐

COROLLARY 7.4. *The reducibility of the augmented flow graph of a GOTO program (or, more generally, of a JUMP program) is a necessary condition for the eliminability of* **go to**'*s from that program under structural equivalence.*

PROOF. Two programs that are structurally equivalent must have the same augmented flow graph, and Proposition 7.3 showed that the augmented flow graph of every EXIT program is reducible. ☐

8. *Dealing with Circuitous* **go to**'*s*

Our next goal is to prove that the reducibility of the augmented flow graph is also a sufficient condition for the eliminability of **go to**'s under structural equivalence. Given a program in JUMP whose augmented flow graph is reducible, the most interesting part of the task of eliminating its **go to**'s will involve stretching arrows and applying the Elimination Rules, as discussed in Section 4. Unfortunately, that theory is not enough as it stands. There are various annoying ways that a program in JUMP can have head-to-head crossings in the **go to** graphs of certain of its blocks, despite having a reducible augmented flow graph.

The most obvious source of such spurious head-to-head crossings is jumps to jumps. Suppose that the destination of the jump statement "**go to** G" is itself an unconditional jump of the form "G: **go to** H;". We wouldn't change the augmented flow graph at all if we replaced the statement "**go to** G" with "**go to** H". But this change might easily get rid of a head-to-head.

There are at least three other, more devious sources of spurious head-to-head crossings. First, one of the **go to**'s involved in a head-to-head might be dead. Second, one of the **go to**'s in a head-to-head might jump to a label G that labels a spin cycle, as in "G: **repeat endloop**;". In this case, we might as well dive off the deep end right away, replacing the statement "**go to** G" with "**repeat endloop**". Third, the two labels involved in a head-to-head crossing might arrive at the same atomic element even though they label different gaps, as in the example

$$\cdots \textbf{go to } \text{H} \cdots \text{G: } \textbf{begin end}; \text{ H: action}_1; \cdots \textbf{go to } \text{G} \cdots .$$

We can get rid of this spurious head-to-head by replacing "**go to** G" with "**go to** H".

All spurious head-to-heads can be dealt with while preserving structural equivalence; unfortunately, the proof, in this section and the next, is a bit tedious. In this section, we describe four preprocessing phases that can be applied to a JUMP program. In the next section, we shall prove that those four phases get rid of all of the head-to-head crossings in any JUMP program whose augmented flow graph is reducible. Once the spurious head-to-heads are gone, the Elimination Rules can finish the job of translating into EXIT.

Phase 1 just cleans up the program somewhat, so that later phases will not be bothered by scoping problems in which a single identifier is used more than once as a gap label. To keep things as simple as possible, we begin Phase 1 by labeling every gap of every block with a brand-new gap label. (If the last statement of a block is not followed by a semicolon, we must insert one before we can label the gap following that statement; the policy of structural equivalence allows us to insert semicolons.) We then replace the old label in each **go to** statement with the corresponding new label. Finally, we delete all of the old labels. This leaves us with a GOTO program in which every gap has exactly one label and all of the labels are distinct. Strictly speaking, of course, the policy of structural equivalence does not allow us to insert new gap labels or to replace one **go to** statement with another. But we intend to delete all of the **go to**'s and all of the gap labels before we are done, so it does not matter what we do with them in the meantime.

The task of Phases 2 and 3 is to make **go to** statements arrive at their final destinations by a route that is reasonably direct. Recall that the *arrival node* of a glue node x in a step graph is the first nonglue node on the path leaving x. Call a **go to** statement *looping* if it never arrives, *halting* if it arrives at [**stop**], and *atomic* if it arrives at an atomic node. Phase 2 deals with the looping and halting **go to**'s, while Phase 3 handles the atomic ones.

Phase 2 is easy. Let Z be the label that Phase 1 gave to the rightmost gap in the outermost block, the gap immediately preceding **stop**. Note that the scope of Z is the entire program since we chose the new labels in Phase 1 to be distinct. If "**go to** G" is a halting **go to**, Phase 2 replaces it with "**go to** Z". If "**go to** G" is a looping **go to**, Phase 2 replaces it with "**repeat endloop**". Recall that the policy of structural equivalence does allow us to insert new **repeat-endloop** pairs.

Phase 3 is trickier, because the prohibition of inward **go to**'s in JUMP means that it cannot succeed as crisply as Phase 2. In particular, a **go to** can arrive at an atomic element that is hidden inside a nested block, as in the example

$\cdots$ **go to** G $\cdots$ G: **begin** action$_1$; $\cdots$ **end**; $\cdots$.

The only way to jump directly to [action$_1$] would be to use an inward **go to**, which the rules of JUMP forbid. Thus, Phase 3 has to settle for enforcing some weaker standard of directness.

Call an atomic **go to** statement "**go to** G" *concise* if its arrival node lies somewhere inside the statement immediately following the gap G; Phase 3 adopts conciseness as its standard. In some cases, this standard is even weaker than it would have to be. Note that the statement "**go to** G" in the example

$\cdots$ G: **begin** H: action$_1$; $\cdots$ **go to** G $\cdots$ **end**; $\cdots$

is concise, even though replacing it with "**go to** H" would make control arrive at [action$_1$] more directly, without violating the rules of JUMP. Note also that making all atomic **go to**'s concise does not necessarily eliminate all instances of jumps to jumps. Both of the **go to**'s in the following example are concise:

$\cdots$ **go to** G $\cdots$ G: **begin go to** H; $\cdots$ H: action$_1$; $\cdots$ **end**; $\cdots$.

It turns out, however, that making all atomic **go to**'s concise does suffice to get rid of all the spurious head-to-head crossings.

By convention, let us say that a halting **go to** is *concise* only if it jumps directly to the rightmost gap in the outermost block, and let us say that no looping **go to** is ever *concise*. With these conventions, Phase 2 made all the nonatomic **go to**'s concise. Phase 3 will make all the atomic **go to**'s concise by repeating a reduction process as often as necessary. Given an atomic **go to** that is not concise, Phase 3 will *reduce* it by replacing it with another **go to** that arrives at the same atomic node and passes through fewer glue nodes along the way.

How do we reduce a **go to**? Suppose that the jump "**go to** G" arrives at the atomic node g. Let B_G be the block one of whose gaps is labeled G. Note that the statement "**go to** G" must lie in B_G, since B_G is the scope of G. The gap labeled G is either the last gap in B_G or it's not.

Suppose first that the gap labeled G is not the last gap in B_G. Let S_G be the statement immediately following the gap labeled G, and let γ be the path through glue nodes in the step graph from the node [**go to** G] to its arrival node g. If the arrival node g lies in S_G, the jump "**go to** G" is already concise. If not, then γ must leave S_G somehow, in order to get to g. By Lemma 5.1, γ must leave S_G by traversing either a sequential arc or a jump arc. If γ leaves S_G by traversing a sequential arc, we can reduce the jump "**go to** G" by replacing it with "**go to** H", where H is the label of the gap immediately following S_G. Since we chose the new labels in Phase 1 to be distinct, the scope of H will be all of B_G, the same as the scope of G, so it will include the jump that we are reducing. If γ leaves S_G by traversing a **go to** arc, say [**go to** K] $\rightarrow$ [K:], we can reduce the jump "**go to** G" by replacing it with "**go to** K". Since performing a "**go to** K" inside of S_G causes control to leave S_G, the label K must label a gap either of the block B_G or of some larger block containing B_G; in either case, the **go to** that we are reducing will lie in the scope of K. Finally, if γ leaves S_G by traversing an **exit** arc, say "**exit** $\mathscr{L}$", we can reduce the jump "**go to** G" by replacing it with "**go to** L", where L is the label of the gap immediately following the loop labeled $\mathscr{L}$. As before, the jump "**go to** G" must lie inside the scope of L. (One might be tempted to rush ahead and simply replace "**go to** G" with "**exit** $\mathscr{L}$" in this case; but that might not work because of name conflicts among the loop labels. Note that Phase 1 only cleaned up the gap labels, not the loop labels.)

On the other hand, the gap labeled G might be the last gap in the block B_G. We then consider the context that surrounds B_G. It cannot be the case that B_G is surrounded by **start-stop**, since [**go to** G] arrives at the atomic node g, not at [**stop**]. If B_G is surrounded by **begin-end**, we can reduce the jump "**go to** G" by replacing G with the label of the gap immediately following this entire compound statement. Similarly, if B_G is surrounded by **then-endthen** or by **else-endelse**, we can jump to the gap following the entire conditional. Finally, if B_G is surrounded by **repeat-endloop**, we can jump to the first gap in B_G instead of the last. We conclude that every nonconcise atomic **go to** can be reduced, which shows that Phase 3 can make all of the atomic **go to**'s concise.

Phase 4 is quite simple: We get rid of the dead **go to**'s by replacing each dead **go to** statement in the program with "**repeat endloop**". Why "**repeat endloop**"? We have to insert some statement in order to keep the program syntactically valid, and the policy of structural equivalence does not give us much choice about what to insert. We waited until Phase 4 to get rid of the dead **go to**'s because the rewriting involved in making **go to**'s concise might cause formerly live **go to**'s to become dead.

9. *Proving the Absence of Spurious Head-to-Heads*

In this section, we prove that the rewriting done by the four phases described in Section 8 is enough to eliminate all spurious head-to-head crossings.

PROPOSITION 9.1. *If all of the* **go to**'*s in a JUMP program P are live and concise, and if the augmented flow graph of P is reducible, then every block in P has a* **go to** *graph that is free of head-to-head crossings.*

PROOF. Suppose that all of the **go to**'s in P are live and concise, but that the **go to** graph of some block in P has a head-to-head crossing. We show that the augmented flow graph of P must then have a simple cycle C with two gateways.

Among the blocks of P that include head-to-head crossings, choose a block B that is minimal in the sense that all of the blocks nested inside of B are free of head-to-heads. Let G and H be the labels of gaps of B that are involved in a head-to-head crossing of the form

$$\cdots \textbf{go to } \mathrm{H} \cdots ; \mathrm{G}\!: \cdots ; \mathrm{H}\!: \cdots \textbf{go to } \mathrm{G} \cdots .$$

Neither [**go to** G] nor [**go to** H] can be a looping **go to**, since they are concise by assumption. Neither of them can be halting, since a concise halting **go to** jumps to the rightmost gap of the outermost block and neither G nor H is the rightmost gap in B. Thus, both [**go to** G] and [**go to** H] must be atomic **go to**'s. Let γ and η denote the paths through glue nodes in the step graph of P from the nodes [**go to** G] and [**go to** H] to the corresponding atomic arrival nodes g and h. Let S_G be the top-level statement of B immediately following the gap G, and similarly for S_H; by conciseness, the atomic element g lies in S_G and h lies in S_H.

Strategy: The cycle C is going to consist of the static arcs along the augmenting path from g to k, where k is an atomic element just before [**go to** G], followed by the dynamic arc from k back to g via the jump [**go to** G] $\rightarrow$ [G:]. The node h is going to be the second gateway to C. We begin by trying to find k.

Since the node [**go to** G] is live, we can choose a path α in the step graph of P from [**start**] to [**go to** G]. The path α must enter the block B at some point. By

Lemma 5.1, it must do so by traversing the sequential arc from the precursor u of B to the leftmost token v in B. If we decompose α the last time that it traverses this arc, we have

$$\alpha = [\textbf{start}] \xrightarrow{*} u \rightarrow v \underset{\beta}{\xrightarrow{*}} [\textbf{go to } \mathrm{G}],$$

where β lies entirely within B.

The path β might traverse **go to** arcs of the form [**go to** K] $\rightarrow$ [K:]; we claim, however, that β must visit some atomic node after traversing any **go to** arc. Suppose, on the contrary, that β had the form

$$\beta = v \xrightarrow{*} [\textbf{go to } \mathrm{K}] \rightarrow [\mathrm{K}{:}] \underset{\delta}{\xrightarrow{*}} [\textbf{go to } \mathrm{G}],$$

where δ is free of atomic nodes. From the node [**go to** G], the path γ takes us through glue nodes to the atomic node g; hence, the node [**go to** K] arrives at g. By conciseness, the label K must label the gap immediately preceding some statement S_{K} containing g. The statements S_{K} and S_{G} cannot be disjoint, since they both contain g. Could we have $S_{\mathrm{K}} \subseteq S_{\mathrm{G}}$? If so, the node [G:] would lie outside of S_{K}. This would mean that the path starting at [**go to** K] would jump to [K:], then enter S_{K}, then leave S_{K} somehow to get to [G:], then reenter S_{K} again to get to g. But a path through glue nodes cannot enter the same statement twice without being caught in a spin cycle, which would preclude ever arriving at g. Therefore, we must have $S_{\mathrm{K}} \supset S_{\mathrm{G}}$. But this cannot happen either. Since S_{G} is a top-level statement of B, the relation $S_{\mathrm{K}} \supset S_{\mathrm{G}}$ would imply [K:] $\notin B$, contradicting the fact that β lies entirely in B.

Our next claim is that the path β must visit at least one atomic node and that the last atomic node it visits must lie after [H:] within B. To see this, note that β manages to get from v to [**go to** G] while remaining within B. In particular, it starts to the left of [H:] and ends to the right of [H:]. Consider the last time that β crosses from the left of [H:] to its right; it is enough to show that β must visit some atomic node after this last cross. To achieve this cross, β must either sequence through [H:] or else jump forward across [H:] by traversing a **go to** arc, since a conditional arc or an **exit** arc that jumped forward across a gap of B would have its head outside of B. If β visits [H:] itself on the last cross from its left to its right, it must then follow the path η through glue nodes to the atomic node h before continuing on to [**go to** G]. Thus, in this case, β does visit an atomic node after the last cross. Suppose, on the other hand, that β achieves the last cross by jumping over [H:] with a forward **go to**. By the result in the last paragraph, β must visit some atomic node after traversing this **go to** arc, so our claim holds once again.

Let k denote the last atomic node that β visits; we have just shown that [H:] $< k$. Let C denote the simple cycle in the augmented flow graph of P that consists of the static arcs from g to k followed by the dynamic arc $k \rightarrow g$. The augmenting path shows that g is one gateway to the cycle C. We intend to show that the node h lies on C and is a second gateway to C.

Since the cycle C consists exactly of the nodes from g through k inclusive, the problem of proving that h lies on C reduces to the problem of showing that $g \leq h \leq k$. We know, in fact, that $g < h$, so the first inequality holds. We also know that [H:] $< k$, which means that k lies either in the statement S_{H} or in some succeeding top-level statement of B. If k does not lie in S_{H}, we must have $h < k$. Suppose, on the other hand, that both h and k lie in S_{H}. By the choice of B, any blocks nested inside of S_{H} have **go to** graphs that are free of head-to-head crossings. Hence, we can apply the Elimination Rules to S_{H} to replace all of the **go to**'s in S_{H}

that are bound in S_H with **exit**'s, without changing the augmented flow graph. After doing so, the only **go to**'s remaining in S_H will be free in S_H, so we can apply Lemma 7.1 to conclude that the node h, which is the arrival node of the precursor [H:] of S_H, is also the leftmost live atomic element in S_H. Since k is a live atomic element that lies in S_H by assumption, we deduce that $h \leq k$ in this case also. Thus, in either case, the node h lies on the cycle C.

It remains to show that h is a second gateway to C, for which we shall use the fact that [**go to** H] is live. Let ϵ be a path in the step graph of P from [**start**] to [**go to** H], and, as before, partition ϵ the last time that it enters the block B; we have

$$\epsilon = [\textbf{start}] \xrightarrow{*} u \longrightarrow v \xrightarrow[\zeta]{*} [\textbf{go to } \mathrm{H}],$$

where ζ remains entirely in B. Recall that the path η leads through glue nodes from [**go to** H] to h. The collapsed version of the combined path $\epsilon\eta$ in the augmented flow graph of P is almost enough to show that the node h is indeed a second gateway to C. The only problem is that ζ might visit atomic nodes of C. We consider two cases. If ζ does not visit any atomic nodes at all, it certainly does not visit any nodes of C, and we are done. So suppose ζ visits at least one atomic node, and let f be the last atomic node that ζ visits. By the same argument that worked for β, we can see that ζ must visit some atomic node after traversing any **go to** arc. Hence, after visiting its last atomic node f, the path ζ cannot traverse any **go to** arcs. We claim that $f <$ [G:]. Otherwise, the path ζ would have to move backward across [G:] to get from f to [**go to** H]. The only backward arcs in a step graph other than **go to** arcs are loopback arcs. But looping back around a loop nested inside of B could not carry us back across the gap label [G:], and looping back around a larger loop in which B was nested would involve leaving B. Thus, we must have $f <$ [G:]. This means that the augmenting path from [**start**] to f avoids all nodes of C. Hence, we can show that h is a second gateway to C in this case by following the augmenting path from [**start**] to f and then the dynamic arc from f to h. At long last, the proof of Proposition 9.1 is complete. □

COROLLARY 9.2. *The reducibility of the augmented flow graph of a GOTO program (or, more generally, of a JUMP program) is sufficient to ensure that all* **go to**'*s can be eliminated from that program under structural equivalence.*

COROLLARY 9.3. *Every reducible augmented flow graph is the augmented flow graph of some EXIT program.*

PROOF. First produce a GOTO program with the given augmented flow graph by the construction in Proposition 6.1; then apply Corollary 9.2 to that GOTO program. □

10. *Testing the Reducibility of Augmented Flow Graphs*

Our theoretical investigations are almost complete, but there is one more point worth making. We have been studying two regularity conditions for programs in JUMP: the reducibility of the augmented flow graph (RAFG) and the absence of head-to-head crossings (AHHC) in the **go to** graphs. These two conditions differ a little in strength: RAFG is both necessary and sufficient for the eliminability of **go to**'s under structural equivalence, whereas AHHC is sufficient but not necessary. But they also differ in another way. AHHC is a first-order condition—that is, it

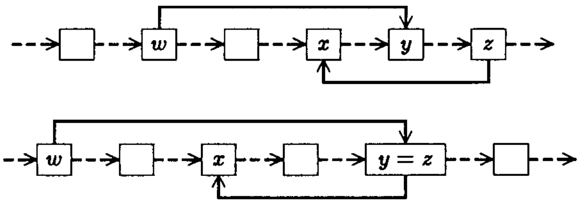


FIG. 16. Two examples of conflicting pairs of arcs in an augmented flow graph.

can be defined simply by quantifying over edges. But RAFG seems more subtle and higher order: All the standard definitions of reducibility talk about complicated things like cycles or reduction algorithms or arc partitions. In fact, this latter difference is an illusion. The notion of reducibility in general is subtle, but the linear order inherent in an augmented flow graph allows us to rephrase the reducibility of those graphs as a first-order condition, in fact, as the prohibition of certain "conflicting" pairs of arcs.

We say that two arcs $w \to y$ and $z \to x$ in an augmented flow graph *conflict* if the nodes involved occur on the augmenting path in the order $w < x < y \leq z$. Two examples of conflicting pairs of arcs are shown in Figure 16. When $y < z$, a conflicting pair in an augmented flow graph looks just like a head-to-head crossing in a **go to** graph. The case $y = z$ is special: It counts as a conflict even though it would not count as a head-to-head.

PROPOSITION 10.1. *An augmented flow graph is reducible if and only if it contains no conflicting pairs of arcs.*

PROOF. If there is a conflicting pair of arcs, it is easy to see that the augmented flow graph is not reducible: The simple cycle composed of the augmenting path arcs from x to z followed by the backward arc $z \to x$ has both x and y as gateways.

Conversely, suppose that C is a simple cycle in the augmented flow graph with more than one gateway; we must show that some conflicting pair of arcs exists. Note that no final node can appear on C itself, nor can any final node appear on any path from [**start**] to any node of C; hence, we may ignore the final nodes. The augmenting path imposes a linear order on the nonfinal nodes of the flow graph. Let u be the smallest node of the cycle C in this order, and let v be the largest nonfinal node of the entire graph from which it is possible to reach u without visiting any node smaller than u. All of the nodes of C must lie in the interval $[u, v]$. The node u will be one gateway to C. By assumption, C has some gateway other than u. The path from [**start**] to this second gateway must enter the interval $[u, v]$ by traversing an arc $t \to h$ with $h \in [u, v]$ and $t \notin [u, v]$. Could we have $t > v$? Note that we can get from t to u without visiting any node less than u as follows: traverse the assumed arc from t to h, follow the augmenting path from h to v, and then do whatever v does to get to u. Since v was chosen as the largest node with this property, the existence of such a node t with $t > v$ would be a contradiction. Hence, we must have $t < u$. Follow the path from v to u that does not visit any node less than u, and consider the first arc $p \to q$ on that path whose head q satisfies $q < h$; such an arc must exist since $u < h$. We have $t < u \leq q < h \leq p$; this implies that the arc $t \to h$ forms a conflicting pair with the arc $p \to q$. $\square$

11. *A Case Study of Arrow Stretching*

The results in this paper are mostly of theoretical interest. But it is possible to imagine unusual practical situations where the Elimination Rules and arrow stretching would come in handy. In fact, it was just such a situation that started me thinking about the problem of preserving structure while eliminating **go to**'s.

In 1982, when I was at the Computer Science Laboratory of the Xerox Palo Alto Research Center, I decided that I would like to import the Pascal version of Knuth's document compiler TEX [12] into the PARC/CSL computing environment. The easiest way to do so involved translating it from Pascal to Mesa [17], the systems programming language used in PARC/CSL. As it happened, Edward M. McCreight (personal communication) had already taken advantage of the family resemblance between Pascal and Mesa by building a Pascal to Mesa source translation tool. I decided to port TEX using McCreight's translator.

Unfortunately, Pascal allows outward **go to**'s—and Knuth used them fairly heavily in TEX—while Mesa does not. Ironically, Knuth is partly to blame for the absence of full-fledged outward **go to**'s in Mesa. The language designers at PARC/CSL were just deciding what local control structures to put into Mesa when Knuth's influential article [11] appeared. In this article, Knuth proposed a restricted form of the **go to** statement based on Zahn's "event indicator" scheme. The designers of Mesa adopted Knuth's proposal. In terms of expressive power, Zahn–Knuth **go to**'s [11] are essentially the same as multilevel **exit**'s; either construct can simulate the other while preserving program structure. Thus, I had a practical interest in replacing outward **go to**'s with multilevel **exit**'s.

The standard technique for doing so would involve computing flow graphs and testing them for reducibility; this looked unattractive for several reasons. First, most of the **go to**'s that Knuth used in TEX were part of simple, well-structured idioms—computing flow graphs seemed like overkill. Second, Knuth was going to a lot of work to make the sources of TEX of publishable quality. It seemed a shame not to preserve as much of Knuth's program structure as possible in the Mesa version. Third, I was planning to debug the resulting Mesa program by reading Knuth's beautifully typeset and indexed listings of the Pascal code [13]. This plan would be feasible only if the two programs corresponded quite closely.

So, I added code for stretching arrows and applying the Elimination Rules to McCreight's translator. Using the beefed-up translator, Michael F. Plass and I successfully translated version 0.8 of TEX into Mesa early in 1983. With the compile-time switch *debug* turned off, that version of TEX had 395 **go to**'s and 162 labels. There were 109 blocks whose **go to** graphs included at least one arrow. Of these, 69 had more than one arrow; twenty had at least one crossing pair of arrows; one included backward-backward crossings; but none had any head-to-heads. Later, when other circumstances caused us to turn on the *debug* switch, we discovered that a single head-to-head had been hiding in the debugging routine *debug_help*. The ordinary flow graph of *debug_help* was reducible, but its augmented flow graph was nonreducible. Its statement order had been contorted so that a label where the user was encouraged to set a breakpoint would appear at the beginning of the procedure body. In the next version of TEX, Knuth rewrote *debug_help* to get rid of this head-to-head.

Given any **go to** graph without head-to-heads, we proved in Section 4 that its edges can be stretched so as to eliminate all crossings; but we did not give an efficient stretching algorithm. Figure 17 shows the algorithm that I implemented. Each edge is represented as a record with the three fields *head*, *tail*, and *newtail*,

```
S ← the empty stack;
for each vertex v in increasing order do
    for each forward edge f with f.head = v in decreasing order of f.tail do
        while f.tail < top(S) do
            w ← pop(S);
            for each backward edge b with b.head = w in any order do
                if b.tail > v then error({f, b} are head-to-head) else b.newtail ← v;
        f.newtail ← top(S);
    push v onto S;
while S is not empty do
    w ← pop(S);
    for each backward edge b with b.head = w in any order do
        b.newtail ← the last vertex;
```

FIG. 17. An algorithm that stretches the edges of a **go to** graph.

the last of which stores the tail of the edge after it has been stretched. The variable *S* denotes an auxiliary stack, used to store those vertices where yet-to-be-processed forward edges could end without crossing previously-processed forward edges. This algorithm has antisymmetric preferences about stretching arrows: It stretches forward edges as little as possible and backward edges as much as possible. As far as efficiency is concerned, McCreight's translator parsed the Pascal source with recursive descent and built up the Mesa target as a tree of program fragments. The additional time needed to stretch arrows and apply the Elimination Rules in that context is linear in the length of the program text, if we ignore the cost of maintaining a symbol table in which to insert and lookup labels.

Another practical issue that deserves comment is the handling of the nonlocal **go to**'s of Pascal, the **go to**'s that jump out of a procedure body to a label defined in an enclosing procedure. Executing a nonlocal **go to** involves returning from one or more procedures and deallocating their activation records. As it happens, Mesa has an exception-handling mechanism that can achieve this effect. Unfortunately, nonlocal **go to**'s have another problem as well: It is hard to predict where they are going to come from. Think about a nonlocal **go to** from the point of view of the destination label G, which labels a gap of some block *B*. A nonlocal "**go to** G" does not occur textually within *B* at all. Instead, there is a call on some procedure *Q* within *B*, and the "**go to** G" appears in the body of *Q* or in the body of some procedure that *Q* calls. We say that a nonlocal **go to** is *forward* if the call on the procedure *Q* occurs before the label "G:" in *B*, else *backward*. By proper use of the Mesa exception machinery, we can prepare for either forward or backward nonlocal **go to**'s to G. In each case, we introduce a new block that is enabled to handle the exceptional condition. One end of this new block must be in the gap labeled G, while the other end must be far enough away so that no nonlocal **go to**'s to G will escape being handled. Unfortunately, we need at least global flow analysis and perhaps theorem proving to determine how large these new blocks have to be. If we try to duck the issue by making them as large as possible, we quickly generate head-to-head crossings.

I do not know of any good solution to the problem of nonlocal **go to**'s. In McCreight's translator, I chose to assume that no backward nonlocal **go to**'s would ever happen, but that forward nonlocal **go to**'s might come from anywhere. This worked fine for TEX, since TEX has only one nonlocal **go to** statement in it: a forward nonlocal jump from the routine that handles fatal errors out to the end of the program. But this simple strategy would clearly fail on programs that made more extensive use of nonlocal **go to**'s.

ACKNOWLEDGMENTS. I would like to thank John Hershberger and Greg Nelson for helpful conversations and Alfred Aho for revealing to me, early in this investigation, my embarrassing ignorance about flow graph reducibility.

REFERENCES

1. BAKER, B. S. An algorithm for structuring flowgraphs. *J. ACM 24*, 1 (Jan. 1977), 98–120.
2. BAKER, B. S., AND KOSARAJU, S. R. A comparison of multilevel **break** and **next** statements. *J. ACM 26*, 3 (July 1979), 555–566.
3. BÖHM, C., AND JACOPINI, G. Flow diagrams, Turing machines, and languages with only two formation rules. *Commun. ACM 9*, 5 (May 1966), 366–371.
4. DIJKSTRA, E. W. **Go to** statement considered harmful. *Commun. ACM 11*, 3 (March 1968), 147–148 (the second instance of pages 147–148).
5. HAREL, D. On folk theorems. *Commun. ACM 23*, 7 (July 1980), 379–389.
6. HECHT, M. S., AND ULLMAN, J. D. Flow graph reducibility. *SIAM J. Comput. 1*, 2 (June 1972), 188–202.
7. HECHT, M. S., AND ULLMAN, J. D. Characterizations of reducible flow graphs. *J. ACM 21*, 3 (July 1974), 367–375.
8. JENSEN, K., AND WIRTH, N. *Pascal User Manual and Report*, 3rd ed. revised for the ISO Pascal Standard by A. B. Mickel and J. F. Miner. Springer-Verlag, New York, 1985.
9. KEOHANE, J., CHERNIAVSKY, J. C., AND HENDERSON, P. B. On transforming control structures. *SIAM J. Comput. 11*, 2 (May 1982), 268–286.
10. KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
11. KNUTH, D. E. Structured Programming with **go to** Statements. *Comput. Surv. 6*, 4 (Dec. 1974), 261–301.
12. KNUTH, D. E. *The TEXbook*, vol. A of *Computers and Typesetting*. Addison-Wesley, New York, 1984.
13. KNUTH, D. E. *TEX: The Program*, vol. B of *Computers and Typesetting*. Addison-Wesley, New York, 1986.
14. KNUTH, D. E., AND FLOYD, R. W. Notes on avoiding "go to" statements. *Inf. Process. Lett. 1*, 1 (Feb. 1971), 23–31 and 177.
15. KOSARAJU, S. R. Analysis of structured programs. *J. Comput. System Sci. 9*, 3 (Dec. 1974), 232–255.
16. LEDGARD, H. F., AND MARCOTTY, M. A genealogy of control structures. *Commun. ACM 18*, 11 (Nov. 1975), 629–639.
17. MITCHELL, J. G., MAYBURY, W., AND SWEET, R. Mesa language manual. Tech. rep. CSL-79-3. Xerox Palo Alto Research Center, Palo Alto, Calif., Apr. 1979.
18. PETERSON, W. W., KASAMI, T., AND TOKURA, N. On the capabilities of **while**, **repeat**, and **exit** statements. *Commun. ACM 16*, 8 (Aug. 1973), 503–512.
19. RAMSHAW, L. Problem 83-1, *J. Algorithms 4*, 1 (Mar. 1983), 85–86.
20. UNITED STATES DEPT. OF DEFENSE. Reference Manual for the Ada Programming Language. Springer-Verlag, New York, 1983.
21. WILLIAMS, M. H., AND CHEN, G. Restructuring Pascal programs containing **goto** statements. *Comput. J. 28*, 2 (May 1985), 134–137.
22. WIRTH, N. *Programming in Modula-2*, 2nd edition. Springer-Verlag, New York, 1983.

RECEIVED DECEMBER 1983; REVISED MARCH 1985, NOVEMBER 1987; ACCEPTED DECEMBER 1987