4

Eliminating go to's while Preserving Program Structure

Lyle Ramshaw

July 15, 1985

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center and two other corporate research laboratories are committed to filling that need.

SRC was established in 1983. We are still making plans and building foundations for our long-term mission, which is to design, build, and use new digital systems five to ten years before they become commonplace. We aim to advance both the state of knowledge and the state of the art.

SRC will create and use real systems in order to investigate their properties. Interesting systems are too complex to be evaluated purely in the abstract. Our strategy is to build prototypes, use them as daily tools, and feed the experience back into the design of better tools and the development of more relevant theories. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

During the next several years SRC will explore applications of high-performance personal computing, distributed computing, communications, databases, programming environments, system-building tools, design automation, specification technology, and tightly coupled multiprocessors.

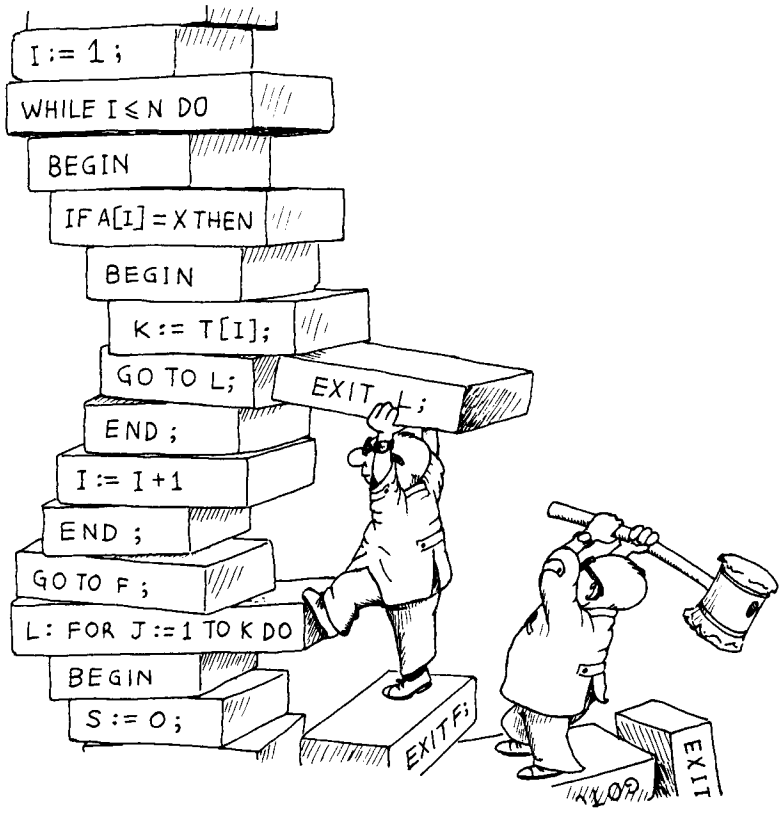
SRC will also do work of a more formal and mathematical flavor; some of us will be constructing theories, developing algorithms, and proving theorems as well as designing systems and writing programs. Some of our work will be in established fields of theoretical computer science, such as the analysis of algorithms. computational geometry, and logics of programming. We also expect to explore new ground motivated by problems that arise in our systems research.

DEC is committed to open research. The improved understanding that comes with widespread exposure is more valuable than any transient competitive advantage. SRC will freely report results in conferences and professional journals. We will actively seek users for our prototype systems among those with whom we have common research interests. We will encourage visits by university researchers and conduct collaborative research.

Robert W. Taylor, Director

Eliminating go to's while Preserving Program Structure

Lyle Ramshaw



Acknowledgements:

Much of this work was done while the author worked at 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 (November 1983).

Authors' abstract:

Suppose that we want to eliminate the local **go to** statements in a PASCAL program by replacing them with multilevel loop **exit** statements. There is a standard technique for doing so that succeeds if and only if the flow graph of the PASCAL program is reducible. This technique assumes that we don't allow ourselves either to introduce new variables or to replicate code, but that we do allow ourselves to reorder the atomic tests and actions within the text of the program and to rewrite the connecting control structures from scratch. In this paper, we shall investigate the extent to which **go to**'s can be replaced with **exit**'s while preserving as much as possible of the program's original structure. On the negative side, we shall find that there are programs whose flow graphs are reducible but whose **go to**'s cannot be eliminated without reordering their tests and actions. That is, programs with **go to**'s can have their atomic elements in some weird static order, an order that doesn't correspond in any structured way to the dynamic flow of control. We shall analyze this situation by augmenting our flow graphs with edges that encode the static order of the atomic elements and then showing that the augmented flow graphs of programs with **exit**'s are always reducible. On the positive side, given a program with **go to**'s whose augmented flow graph is reducible, we shall show that we can replace its **go to**'s with **exit**'s while preserving essentially all of its structure. In fact, we can simply delete the **go to** statements and the labels they jump to and insert various **exit** statements and labeled **repeat-endloop** pairs for them to jump out of, without changing the rest of the program text in any way.

Lyle Ramshaw

Capsule review:

The controversy that surrounds the use of the **go to** statement has spawned several studies of how **go to**'s can be replaced by more structured loop constructs. This paper reports one such study.

Whereas previous works tended to adopt notions of program equivalence that ignored the structures of programs, the preservation of program structure is of paramount importance in this work. In particular, the author is concerned with the class of programs for which **goto**'s can be eliminated by simply deleting them and replacing in their stead judiciously inserted **repeat** loops and multilevel **exit**'s. The author proves that this class contains precisely those programs whose flow graphs are reducible even when augmented with edges that encode the static flow of the program. In the proof, the author presents a method for eliminating **go to**'s from any program in this class.

This paper assumes only an elementary knowledge of some block-structured programming language. It is very readable in spite of its theoretical nature: the context is laid out clearly, the flow from initial lemmas to final results is structured well, and the paper is sprinkled liberally with intuitive explanations.

Sheng-Yang Chiu

Table of Contents

Policy	Source	Target	Condition
functional equivalence	go to's	one **while** loop	always
semantic equivalence	go to's	multilevel **exit**'s	always
path equivalence	go to's	multilevel **exit**'s	always
flow graph equivalence	go to's	multilevel **exit**'s	when flow graph is reducible
linear equivalence	go to's	multilevel **exit**'s	when augmented flow graph is reducible
structural equivalence	outward go to's	multilevel **exit**'s	when augmented flow graph is reducible
Elimination Rules only	outward go to's	multilevel **exit**'s	unless program has head-to-head crossings

Figure 1: Necessary and sufficient conditions for replacing the source construct by the target construct under the stated policy

1. Introduction

The **go to** statement was the center of much controversy ten or fifteen years ago [4, 12]. 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 Figure 1.

At the outset, let us agree to ignore the *non-local* **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. 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, such as *functional equivalence* [16]. Two programs are called functionally equivalent whenever they produce the same output for all inputs. In particular, introducing new variables in the target program is not ruled out. By introducing one new variable that acts as a program counter, we can replace all of the control structures of a 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.

The elimination of **go to**'s becomes more challenging if we adopt a stricter notion of equivalence. Two programs are called *path equivalent* [2, 16] or *strongly equivalent* [15] if, for all inputs, the sequences of tests and actions that the two programs perform are identical. Knuth and Floyd [11] 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 [21], supplement these three with

a "do forever" loop, for which we shall 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. (By the MODULA-2 rules, 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 can't 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**, a primitive that can terminate any enclosing **repeat** loop [15, 18]; ADA† [20] is one example of a language with a multilevel **exit** statement. 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 reduction doesn't work with single-level **exit**'s because of the tricky case mentioned in the previous parentheses.)

For completeness, we observe 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 all tests are free of side effects. It then loosens the rules by allowing the target program to perform redundant or useless evaluations of tests. From our point of view, semantic equivalence has the same properties as path equivalence: multilevel **exit**'s suffice but single-level **exit**'s do not [15].

There is a sense in which even the policy of path equivalence is rather permissive: 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 tests and actions of the two programs that respects control flow (a fine point: tests and actions in dead code don't count). 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 cycle 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, from the point of view of the back end of a compiler, essentially identical. 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. The structured programming movement has shown that this type of program structure is important. Indeed, the inventors of flow graph equivalence were well aware of the importance of program structure: their prime goal in eliminating **go to**'s was to improve the structure of their source programs [1]. Most of them had no interest, however, in trying to preserve the existing structure of the source. After all, since the source program had **go to**'s, how could it have any structure worth preserving? Our goal is

† ADA is a registered trademark of the U. S. Government (ADA Joint Program Office).

somewhat different: we are trying to understand the power of the **go to** statement. To be thorough, we must consider the interactions between **go to**'s and the other local control constructs. To what extent can **go to**'s be eliminated while preserving the original structure of the source program?

Peterson, Kasami, and Tokura made a start on this problem in a paper in 1973 [18]. They introduced a notion of structure-preserving equivalence, which we shall adopt 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 their corresponding statement labels and then inserting various **exit** statements and appropriately labeled **repeat-endloop** pairs, without rearranging or altering any other statements. Note that this policy 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. There isn't any hope of eliminating **go to**'s in general unless we allow ourselves to create new **repeat** loops; the source program might not have any **repeat** loops, and **exit** statements aren't any use unless there are loops for them to terminate. Thus, structural equivalence is about the strictest policy that leaves much chance of widespread success.

How many **go to**'s can we eliminate under 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 an arm 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 and ADA, 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 all over the place. We shall give up on inward **go to**'s right away.

Peterson, Kasami, and Tokura also gave up on inward **go to**'s, of course; among outward **go to**'s, they focused on the forward ones [18]. 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, Kasami, and Tokura 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 shall 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 hardly enough of them to be worth mentioning. They allowed 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 shall begin our investigations by presenting an analogous Backward Elimination Rule that handles backward, outward **go to**'s. 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 and Backward Elimination Rules in the context of the languages GOTO and EXIT. Both of the Elimination Rules involve introducing new **repeat** loops; in certain cases, these requests for new loops can conflict with each other. Section 4 defines the concept of a *head-to-head crossing* and shows that the two Elimination Rules can handle all of the **go to**'s in a GOTO program if and only if that program is free of head-to-head crossings.

One way that head-to-head crossings can arise is when **go to** statements are used to write a program in which the static order of the tests and actions in the program text doesn't correspond to the dynamic order in which those atomic elements get executed. After reviewing the standard notion of a flow graph in Section 5, we shall define in Section 6 an augmented form of the flow graph in which there are edges encoding the static order of the atomic elements as well as the 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. This gives us a necessary condition for eliminating **go to**'s under structural equivalence: if the augmented flow graph of a GOTO program isn't reducible, then we can't possibly eliminate its **go to**'s without at least permuting its atomic elements into a different static order.

The reducibility of the augmented flow graph remains a necessary condition even if we adopt a policy that is intermediate between flow graph equivalence and structural equivalence. Call two programs *linearly equivalent* if their augmented flow graphs are the same; that is, their ordinary flow graphs are the same and their live atomic elements occur in the same static order. The result in Section 7 shows that programs (in GOTO or not in GOTO) with nonreducible augmented flow graphs cannot be translated into EXIT under linear equivalence.

Section 8 completes our analyses of structural and linear equivalence by proving that the reducibility of the augmented flow graph is also a sufficient condition for eliminating **go to**'s under either of the two policies. In Section 9, we show that the property of reducibility simplifies in the special case of augmented flow graphs to a rule prohibiting certain "conflicting" pairs of edges. Finally, Section 10 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.

2. The languages GOTO and EXIT

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 shall work with two idealized languages called GOTO and EXIT, which distill the essence of outward **go to**'s and multilevel **exit**'s.

To be more precise, GOTO and EXIT aren't programming languages, but rather languages in which to write uninterpreted program schemata. The atomic actions of a real programming language are explicit commands to do something, such as assignment statements, input/output statements, and procedure calls. But in GOTO and EXIT, the actions are just uninterpreted symbols drawn from the set $\{\text{action}_1, \text{action}_2, \ldots\}$. Similarly, the tests in GOTO and EXIT are just symbols from the set $\{\text{test}_1, \text{test}_2, \ldots\}$.

Both GOTO and EXIT have six types of statements: nulls, actions, conditionals, compounds, loops, and jumps. Null statements are represented by the empty string. Action statements consist of a single action symbol. Conditional statements, which we shall write

if ⟨test⟩ **then** ⟨statement⟩ **else** ⟨statement⟩ **fi**,

and compound statements, which we shall write

begin ⟨statement⟩; ⟨statement⟩; ... ⟨statement⟩ **end**,

are the standard things. The loop statement in GOTO and EXIT is a "do forever" loop, which we shall write

repeat ⟨statement⟩; ⟨statement⟩; ... ⟨statement⟩ **endloop**.

The only way out of a loop is to use a jump statement, and jump statements are the one area where the two languages differ.

Let us use the term ***block*** to denote either a compound statement or a loop. In GOTO, any top-level statement of any block may be labeled. If L labels some top-level statement of a block, the phrase "go to L" becomes a legal statement throughout that block, and its effect is to transfer control to the statement labeled L. Multiple labels on a single statement are allowed, as are multiple statements given the same label (as long as they are not top-level statements of the same block). When there are top-level statements labeled L in several blocks containing a "go to L", the label in the innermost enclosing block determines the destination of the jump.

Note that one can't jump into a block from outside it in GOTO, because the jump statement "go to L" is defined only inside that block of which the label "L:" labels a top-level statement. One can't jump into a conditional from outside it or jump from one arm of a conditional to the other because only top-level statements of blocks may be labeled. Thus, the language GOTO permits only outward go to's.

In EXIT, it is loops that may be labeled, rather than statements. We shall use a postfix ":L" for loop labels, in contrast to the prefix "L:" used for statement labels:

repeat ⟨statements possibly including "**exit** L"⟩ **endloop** :L.

Anywhere within a loop labeled L, the phrase "exit L" is a legal statement, and its effect is to terminate the loop immediately. As in GOTO, we shall allow multiple labels on a single loop and multiple loops given the same label. If there are several loops labeled L, each "exit L" statement terminates the innermost such loop that encloses it.

Statements in GOTO or EXIT are almost programs in their own right. The only difficulty is that they may contain ***dangling jumps***, go to's or exit's to labels that are not bound by the surrounding text. We shall dignify those statements that are free of dangling jumps by calling them ***programs***.

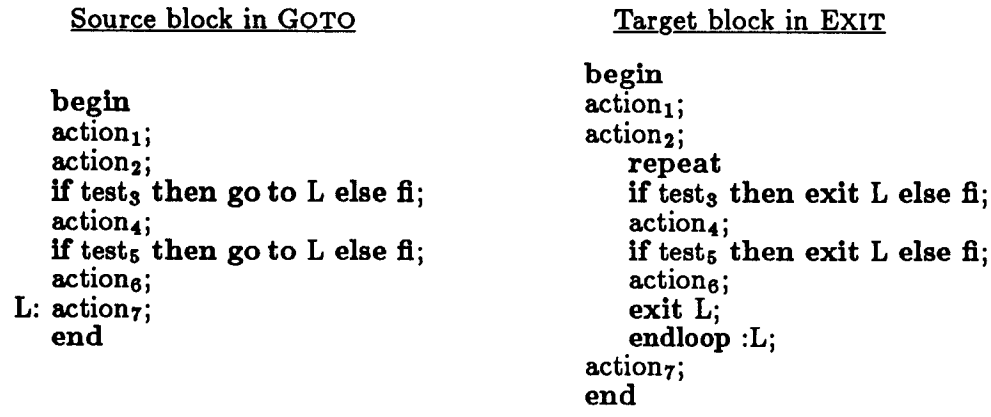


Figure 2: An example of the Forward Elimination Rule

3. The Forward and Backward Elimination Rules

Our goal is to translate GOTO programs into structurally equivalent EXIT programs. Suppose that only one of the top-level statements of a block in GOTO is labeled and that this label is only gone forward to. Following Peterson, Kasami, and Tokura [18], we can produce a structurally equivalent block in EXIT by the technique shown in Figure 2, which we shall call the *Forward Elimination Rule*. Note that no statement in the new, labeled loop will be executed more than once, despite the presence of the keyword **repeat**. We could make the Forward Elimination Rule a little simpler if EXIT allowed compound statements to be labeled and exited as well as loops. But it is traditional to restrict **exit** to exiting only loops, and we have no compelling reason to break with this tradition.

The end of the new loop in the EXIT program has to go in the same gap between statements as the label "L:" in the GOTO program. But we have a certain amount of freedom about where to put the beginning of the new loop. In Figure 2, for example, we could have started it either one or two statements earlier. In general, we can place the keyword **repeat** in any gap between top-level statements that precedes the first such statement containing a dangling "go to L".

With just a little more work, we can also handle a block in GOTO with a single label that is only gone backward to, as shown in Figure 3. We shall call this technique the *Backward Elimination Rule*. We could get away with only one new loop in this rule instead of two if EXIT had a multilevel **next** or **continue** statement, a primitive that could force a new iteration of any enclosing loop to begin immediately. But, once again, we shall resist the temptation to clutter up EXIT with unnecessary features.

In the Backward Elimination Rule, it is the location of the beginning of the new loops that is precisely determined by the location of the label "L:". The terminating boilerplate, the phrase "**exit** L; **endloop** :LL; **endloop** :L;", can be placed in any gap between top-level statements that follows the last such statement containing a dangling "go to L".

In general, more than one of the top-level statements of a block in GOTO may be labeled, and each of these labels may be gone to from various places. If we try to eliminate all of these go to's

Source block in GOTO

```
   begin
   action1;
L: action2;
   if test3 then go to L else fi;
   action4;
   if test5 then go to L else fi;
   action6;
   action7;
   end
```

Target block in EXIT

```
begin
action1;
   repeat
      repeat
      action2;
      if test3 then exit LL else fi;
      action4;
      if test5 then exit LL else fi;
      exit L;
      endloop :LL;
   endloop :L;
action6;
action7;
end
```

Figure 3: An example of the Backward Elimination Rule

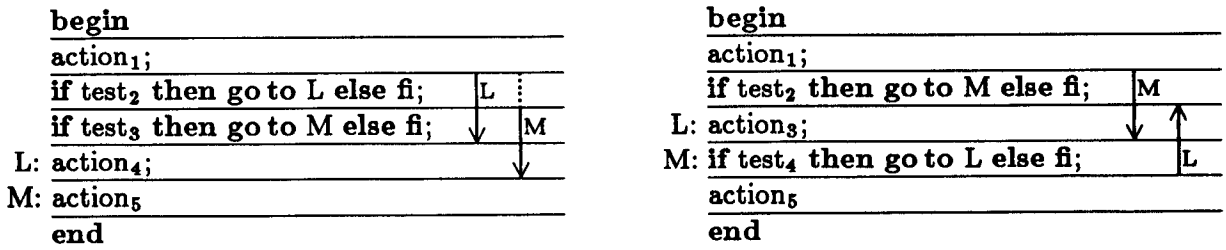


Figure 4: Conflicts between requests for new loops

by using the two Elimination Rules, the resulting requests to introduce new loops may conflict with one another. Two simple cases of conflict are shown in Figure 4.

In the left-hand example, 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 resolved quite easily by making the new loop introduced for the label M longer than it would otherwise have to be—in particular, just long enough so that it completely contains the new loop for the label L. We shall call this process *stretching* the M loop.

The right-hand example in Figure 4 is 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 Figure 4, are quite helpful in studying conflicts between requests for new loops; we shall call them *go to graphs*. Suppose that we are given a block B in the language GOTO, some of whose top-level statements 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 between top-level statements of B; they are drawn as horizontal lines in Figure 4. The edges, drawn as vertical arrows, are derived as follows. For each top-level label L that is gone forward

to, we add an arrow whose head is the statement gap containing the label "L:" and whose tail is the gap just before the first top-level statement of B, call it S, containing a dangling "go to L". Note that the dangling go to itself may appear anywhere in S, possibly inside one or more nested blocks; the arrow's tail is the statement gap in B just preceding S, whether or not such nested blocks exist. The arrows for backward go to's are similar; for each label L that is gone backward to, we add an arrow whose head is at "L:" and whose tail is just after the last top-level statement of B containing a dangling "go to L". Each arrow represents the extent of the shortest new loop that we could insert 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 shall draw a stretched arrow by a dotted extension of the arrow's shaft. The various possible stretched versions of an arrow correspond precisely to the various possible stretched loops that we could insert 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 top-level labels of the block B.

In general, a GOTO program will have many blocks, nested inside of each other, and each block will have its own go to graph. Note that we can apply the stretching technique and the Elimination Rules to each block separately; different blocks don't interfere with each other. Thus, if we can stretch the arrows in all of the go to graphs of a GOTO program so as to eliminate all conflicts, we can translate from GOTO to EXIT while preserving structural equivalence. 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 set of 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 5. (In this section, we shall draw our go to graphs rotated 90 degrees.) If two edges share an endpoint, we shall count that pair as either disjoint or nested; in order for a pair to cross, four distinct vertices must be involved. There are four types of crossing pairs, depending upon the directions of the two arrowheads. We shall call them *forward-forward*, *backward-backward*, *head-to-head*, and *tail-to-tail*.

Our goal is to stretch the edges of a go to graph so as to eliminate crossings. As we noted above, stretching moves can't eliminate head-to-head crossings. On the other hand, we shall find that stretching moves are powerful enough to eliminate all other crossings.

Proposition 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 1 is not trivial, consider the situation in Figure 6. 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 can't eliminate all crossings just by stretching arrows heedlessly.

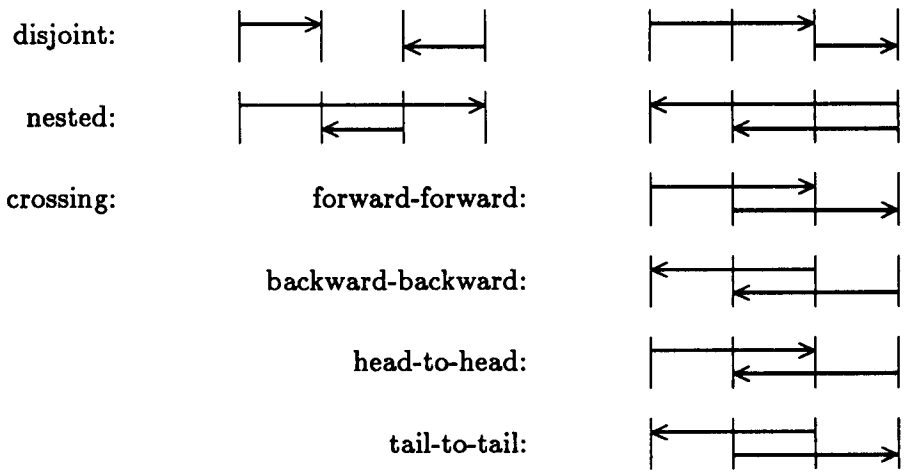


Figure 5: Classifying pairs of edges in a go to graph

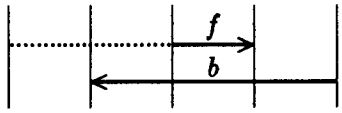


Figure 6: A head-to-head crossing caused by careless stretching

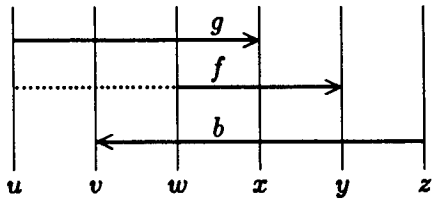


Figure 7: The case of a forward-forward crossing in the proof of Lemma 2

On the other hand, Proposition 1 isn't too subtle either. As long as we are careful not to introduce any head-to-head crossings, stretching arrows heedlessly does just fine. This observation is due to Susan Owicki; it forms the basis of her simpler proof of my Proposition 1. By the way, Susan Owicki also deserves credit for the good idea of drawing the vertices of a go to graph as parallel lines rather than as points.

Lemma 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-heads.*

If we can prove this lemma, Proposition 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 won't be able to keep applying Lemma 2 forever. When it no longer applies, the graph must be free of all crossings.

To prove Lemma 2, we shall 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 7. In such a

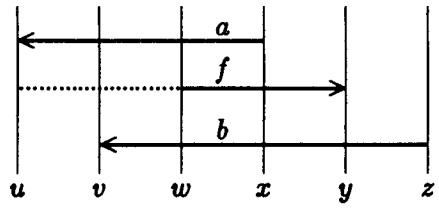


Figure 8: The case of a tail-to-tail crossing in the proof of Lemma 2

situation, we shall 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't 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 of arrows, call them a and f, 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. But stretching f makes this case, shown in Figure 8, 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 $u < v \leq w$. 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 can't introduce any head-to-heads, and that is enough to complete the proofs of Lemma 2 and of Proposition 1.

Putting Proposition 1 together with the Elimination Rules, we have the following corollary.

Corollary 3. *Let P be a program in GOTO. 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-heads 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.

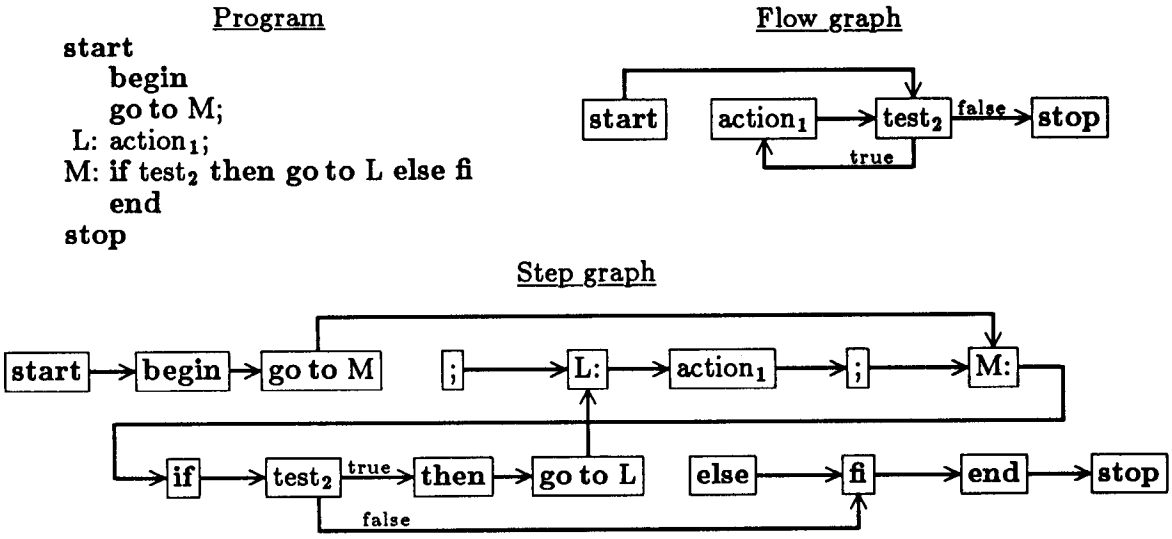


Figure 9: The GOTO program While, its flow graph, and its step graph

5. Step graphs, flow graphs, and reducibility

Before we can make further progress, we have to become precise about the notions of "flow graph" and "reducibility." As a first step in this direction, we shall define the semantics of a GOTO program by constructing the state diagram of an interpreter, which we shall call the program's *step graph*. When building the step graph, we shall pad out the text of the program by adding the special symbols **start** and **stop** at the beginning and end. Figure 9 shows a sample GOTO program along with its step graph.

The nodes of the step graph correspond to the tokens of the program text. We shall write a node by putting brackets around the corresponding token. Nodes come in four classes: [**start**] is the *initial node*; [**stop**] is the *final node*; each action or test is an *atomic node*; and the rest of the program text is broken up into *glue nodes* from the list

[**if**], [**then**], [**else**], [**fi**], [**begin**], [**end**], [**repeat**], [**endloop**], [;], [L:], and [**go to** L],

where L denotes any label.

The edges of the step graph are determined by a straightforward set of rules. Most nodes have a single outgoing edge leading to the node for the following token. This rule applies to the initial node [**start**], to all atomic action nodes, and to the glue nodes [**if**], [**then**], [**fi**], [**begin**], [**end**], [**repeat**], [;], and [L:]. Atomic test nodes have two outgoing edges. One of them is labeled "true" and goes to the matching node [**then**]; the other is labeled "false" and goes to whatever node immediately follows the matching [**else**]. The false branch has to start just after [**else**] rather than at [**else**] since the node [**else**] itself functions as the end of the true branch. This convention also implies that the edge leaving the node [**else**] jumps straight to the matching [**fi**]. Each node [**endloop**] has one outgoing edge to the matching [**repeat**]. Each node [**go to** L] has one outgoing edge to the matching [L:]. Finally, the node [**stop**] has no outgoing edges.

Note that a step graph may contain nodes that cannot be reached by following paths from [**start**]. We shall call the reachable nodes *live* and the unreachable ones *dead*. The step graph in Figure 9 has two dead nodes, both of which are glue nodes: the first [;] and the [**else**].

Compiling a GOTO program instead of interpreting it involves preprocessing the step graph in order to eliminate chains of glue nodes. We might as well delete the dead nodes at the same time. The resulting simpler graph is called the *flow graph* of the program. Figure 9 also shows the flow graph of its sample program. We shall call this program While, since it has the same flow graph as "**while** test_2 **do** action_1".

The nodes of the flow graph are just the live, non-glue nodes of the step graph. The edges of the flow graph correspond to paths in the step graph from one live non-glue 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 paths. There is one unpleasant possibility, however: we might get caught in a cycle of glue nodes, such as those in the step graphs of "**repeat endloop**" or "L: **go to** L". We shall call such cycles *subatomic*. If this ever happens, we shall add a second final node called [**spin**] to the flow graph, and we shall represent paths that enter subatomic cycles as edges to [**spin**].

The step graphs and flow graphs of EXIT programs are defined completely analogously; the edge leaving the glue node [**exit** L] goes to the matching loop label node [:L]. We can also define the step graphs and flow graphs of statements in GOTO or EXIT that include dangling jumps, that is, jumps to unbound labels, by treating the dangling jumps as final nodes instead of as glue nodes.

Step graphs are tied to a particular programming language, but we can define the notion of flow graph independent of any language. A *flow graph* is a directed graph (multiple edges and self-loops both allowed) with certain properties. Each node of a flow graph has a label drawn from the set of symbols {**start**, **stop**, **spin**, action_1, action_2, ..., test_1, test_2, ...}. There must be exactly one node [**start**], with no incoming edges and exactly one unlabeled outgoing edge. All nodes of the flow graph must be reachable from [**start**]. There must be at most one node [**stop**] and at most one node [**spin**]; if these nodes exist, they must have out-degree zero. Each action node must have exactly one unlabeled outgoing edge. And each test node must have exactly two outgoing edges, one labeled "true" and the other "false".

As a first exercise in the use of these concepts, we can prove that every flow graph is the flow graph of some GOTO program. Given a flow graph, label all of its nodes other than [**start**] with distinct identifiers. 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 L and its outgoing edge goes to the node labeled M, we write "L: action_i; **go to** M;". If the node [test_i] is labeled L and its successors are labeled M and N, we write "L: **if** test_i **then go to** M **else go to** N **fi**;". The node [**start**] is written "**go to** A;", where A is the label of its successor. If a node [**stop**] is present and labeled Z, it is written "Z:;", a null statement labeled Z. If a node [**spin**] is present and labeled Y, we write it "Y: **repeat endloop**;". We can then concatenate these fragments together in any order, as long as [**start**] goes first and [**stop**] goes last, and wrap them up into a compound statement with a **begin** and **end**. The result is a GOTO program with the specified flow graph.

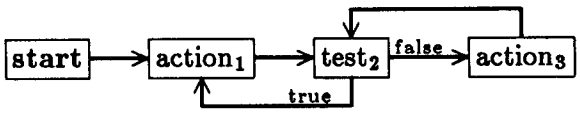


Figure 10: A reducible flow graph containing a two-entry cycle

Unlike programs in GOTO, programs in EXIT can't 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 that cycles in the graph can be first entered. Define a *simple cycle* in a flow graph to be a cycle that doesn't visit any node more than once. If v is a node on a simple cycle C, we shall 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 is not the same thing as an entry node. A node x in a set of nodes S is called an *entry to* S if x is the head of some edge whose tail lies outside of S. Every gateway to a cycle is also an entry to that cycle, but not every entry is a gateway. As Hecht and Ullman noted [6], reducible flow graphs, like the example in Figure 10, can have cycles with multiple entries: both of the nodes of the cycle consisting of [action_1] and [test_2] are entries. But only [action_1] is a gateway.

6. Augmented flow graphs and linear equivalence

Given a GOTO program with no head-to-head crossings, the Elimination Rules can translate it into EXIT under structural equivalence. In particular, this means that the flow graph of such a program must be reducible. The Elimination Rules are stumped, however, by head-to-head crossings. We can't fault the Elimination Rules for failing on programs whose head-to-heads are associated with a nonreducible flow graph; there isn't any hope of eliminating the **go to**'s from such programs while preserving even their flow graphs, much less their program structure. But what about the programs in the middle, the ones that have reducible flow graphs despite having head-to-head crossings? One example of such a program is While, the sample program in Section 5. A glance at the program text of While in Figure 9 shows that While does have a head-to-head in the **go to** graph of its only block, but the flow graph of While is certainly reducible. As a consequence of this reducibility, there are EXIT programs with the same flow graph as While; the simplest such is

repeat if test_2 **then** action_1 **else exit** L **fi endloop** :L.

This EXIT program is flow graph equivalent to While, but not structurally equivalent. Can we eliminate the **go to**'s from While under structural equivalence, or not?

We can't. In the text of While, the atomic element "action_1" comes before the element "test_2". We shall prove below that the test comes before the action in every EXIT program with this flow graph. Since the static order of the atomic elements is one of the flavors of structure that structural

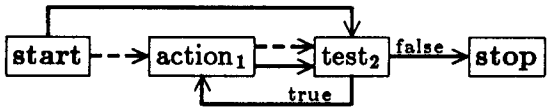


Figure 11: The augmented flow graph of While

equivalence preserves, the go to's in While cannot be eliminated under structural equivalence. The proof involves encoding this static order in a program's flow graph by adding extra edges.

Define an *augmented flow graph* to be a labeled, directed graph whose edges are partitioned into two classes: *dynamic* and *static*. If the static edges are deleted, what remains must be an ordinary flow graph. In addition, the static edges 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 of the final nodes. The order in which the augmenting path visits the atomic nodes encodes the static order of the corresponding live atomic elements in the program text. Note that, since dead atomic elements are deleted when building the flow graph, their static order is not encoded by the augmenting path. Figure 11 shows the augmented flow graph of the GOTO program While, with the two static edges drawn dashed. This graph is not reducible; both of the nodes [action$_1$] and [test$_2$] are gateways to the cycle.

Given a GOTO or EXIT program, we can construct the corresponding augmented flow graph. Furthermore, every augmented flow graph is the augmented flow graph of some GOTO program. To produce such a program, we need only assemble the program fragments that we constructed in Section 5 in the order specified by the augmenting path. What about EXIT programs? In Sections 7 and 8, we shall prove that the augmented flow graphs of EXIT programs are precisely the reducible augmented flow graphs. In particular, every EXIT program for a **while** loop must have the test before the action.

Call two programs *linearly equivalent* if their augmented flow graphs are the same. In addition to demanding that their ordinary flow graphs be the same, this policy demands that the live atomic elements in the two programs occur in the same static order. We can rephrase the result claimed in the last paragraph as follows: all go to's can be eliminated from a program under the policy of linear equivalence if and only if that program's augmented flow graph is reducible. Note that there is no need to restrict the source programs to have only outward go to's in this result; inward go to's or even weirder control structures could be allowed in the source program, since only the augmented flow graph of the source matters.

There are several fine points in our definition of flow graph that deserve comment. First, most people distinguish one of the atomic nodes of their flow graphs as the initial node, instead of adding a separate node [start]. Note that the nonreducibility of the augmented flow graph of While depends upon the existence of the separate [start]. Second, the careful reader might be curious about why we have chosen to leave the dead code out of our augmented flow graphs. We had to leave the dead code out of our ordinary flow graphs, since all of the nodes in any flow graph must be accessible from [start]. But we could have chosen to put the dead code back in when

```
start
   repeat
      repeat
         repeat
         exit M;
         action1;   } dead code
         exit L;    }
         endloop :M;
      action2;
      exit L;
      endloop :L;
   action3;
   endloop
stop
```

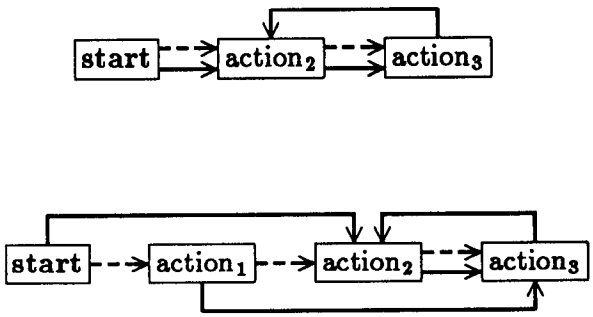


Figure 12: An EXIT program, its augmented flow graph, and the nonreducible graph that would result if dead code were included in augmented flow graphs

building the augmented flow graph, since the augmenting path itself would have been enough to guarantee that all nodes were accessible from [**start**]. The trouble is, if we had put the dead code back in, there would have been EXIT programs whose augmented flow graphs weren't reducible. One example of such a program is given in Figure 12.

7. Exploiting the structure of exit's

Proposition 4. *The augmented flow graph of every EXIT program is reducible.*

We shall begin by proving that the ordinary flow graphs of EXIT programs are reducible. This result has been known for some time [6], and the proof that we shall give is the standard one. We shall then extend the proof to show that adding the static edges doesn't foul up the reducibility.

Fix an EXIT program and consider some simple cycle C in its ordinary flow graph; we want to demonstrate that C has only one gateway node. Note that C can only include atomic nodes, since the initial node [**start**] has in-degree zero while all final nodes have out-degree zero. Each edge in the cycle C thus corresponds to a path from one atomic node to another through glue nodes in the step graph of the EXIT program; consider the cycle C' in the step graph formed by these paths. Note that the only control primitive in EXIT that moves control to the left in the program text is the backward jump from an [**endloop**] to the matching [**repeat**]. Since C' is a cycle, it must include at least one such backward jump.

Consider the set of backward jumps from [**endloop**]'s to their matching [**repeat**]'s that occur as we traverse the entire cycle C'. Any two of them must be either disjoint or nested; they can't cross because they arise from the block structure of the program. Thus, this set of backward jumps forms a forest. Furthermore, there can't be more than one tree in this forest, because there would be no way for control to move backward from one tree to another. We deduce that the set of backward jumps in C' forms a single tree, the root of which is a backward jump whose **repeat** loop, call it R, contains all of the nodes of C. We shall denote this root jump by $[\mathbf{endloop}_R] \rightarrow [\mathbf{repeat}_R]$.

Suppose that we begin at the glue node [**repeat**$_R$] of the step graph and follow edges; let y be the first non-glue node that we come to. Note that we can't enter a subatomic cycle, because the cycle C' includes the edge [**endloop**$_R$] $\rightarrow$ [**repeat**$_R$]. In fact, the node y must be an atomic node that belongs to C and lies in R. The only way into a **repeat** loop in EXIT from outside it is to sequence in through the **repeat**; thus, the node y is the only entry to R in the flow graph. This implies that y is also the only gateway to C: any path from [**start**] to any node of C must enter R, and can do so only by visiting y. Therefore, the ordinary flow graphs of EXIT programs are reducible.

We want to extend this proof to handle augmented flow graphs; how does the presence of the static edges change things? First, a simple cycle C in an augmented flow graph may traverse static edges, and they don't correspond to paths in the step graph. Note that all of the static edges go forward in the program text, however. Thus, all of the backward edges of C must be dynamic edges, and each of them will lift to a path in the step graph. Arguing as before on the edges in this set of paths, we deduce that there exists a **repeat** loop R in the program with two properties: R includes all of the nodes of C and C includes an atomic node y that is the first non-glue node reached by following edges in the step graph from [**repeat**$_R$].

Next, consider a path in the augmented flow graph from [**start**] to any node of C; in particular, consider how that path first enters the loop R. We already know that all of the dynamic edges that enter R from outside it have y as their head. What about static edges? Since the loop R is a contiguous chunk of program text, the nodes in the augmented flow graph that correspond to atomic elements in R occur as a contiguous subpath of the augmenting path. Hence, there is exactly one static edge that enters R from outside it, and its head is the node corresponding to the leftmost live atomic element in the text of R. We will be done if we can show that the node y defined above is also the leftmost live atomic element in R.

Lemma 5. *If a statement S in EXIT contains any live atomic elements, then the first atomic element to be executed when S is executed is also the leftmost live atomic element in S; that is, the dynamic and static successors of [**start**] in the augmented flow graph of S are the same.*

The proof is by structural induction. Let S be a statement in EXIT that contains at least one live atomic element, and let d denote the dynamic successor of [**start**] in the flow graph of S. Since S contains live atomic elements, the node d must be atomic rather than final. We want to show that the atomic node d is also the static successor of [**start**]. Of the six types of statements in EXIT, the statement S can't be a null statement or an **exit**, since S contains d. If S is an action statement, d must be that action; if S is a conditional, d must be the associated test. In either case, it is clear that d is also the leftmost live atomic element in S. The remaining two cases are compounds and loops. In these cases, the node d must lie in some top-level statement T of S. Since d is the dynamic successor of [**start**] in S, the top-level statements of S preceding T must each have the trivial flow graph [**start**] $\rightarrow$ [**stop**]. In particular, none of these statements can contain any live atomic elements. Also, the node d must be the dynamic successor of [**start**] in T. By

induction, we can conclude that d is the leftmost live atomic element in T, and hence also in S. This completes the proofs of Lemma 5 and Proposition 4.

Corollary 6. *If a GOTO program is free of head-to-head crossings, then its augmented flow graph is reducible.*

Proof: In the absence of head-to-head crossings, we can use the Elimination Rules to produce a target EXIT program that is structurally equivalent to the source GOTO program and, consequently, has the same augmented flow graph. This common augmented flow graph must be reducible by Proposition 4.

8. Dealing with the go to pathologies

We have made progress, but we aren't done yet. Proposition 4 explains the failure of the Elimination Rules on all GOTO programs whose augmented flow graphs aren't reducible. But there are GOTO programs with reducible augmented flow graphs that, nevertheless, have head-to-head crossings in their go to graphs. Can such programs be translated into EXIT under structural equivalence, or not? In this section, we shall find that they can.

How can a program have a head-to-head crossing when its augmented flow graph is reducible? The most likely way in practice is probably jumps to jumps. Suppose that the destination of the jump statement "go to L" is itself an unconditional jump, of the form "L: go to M;". We wouldn't change the augmented flow graph any if we replaced the statement "go to L" with "go to M". But this change could easily eliminate a head-to-head. There are at least three other, more devious ways that head-to-heads can arise. First, one of the go to's involved in the head-to-head might be dead. Second, one of the go to's might jump to a label L that labels a subatomic cycle, as in "L: repeat endloop;". In this case, we might as well dive off the deep end right away, replacing the statement "go to L" with "repeat endloop". Third, the two labels involved in a head-to-head crossing might be redundant; that is, they might refer to the same atomic element even though they label different statements, as in the example "L:; M: action$_1$;". In this case, we could eliminate the head-to-head by choosing one of the labels and making both of the go to's jump there.

All of these pathologies can be handled without destroying structural equivalence; unfortunately, the proof is a bit tedious. We shall describe four preprocessing phases that can be applied to a GOTO program to fight the pathologies. Then, we shall prove that these four phases get rid of all of the head-to-head crossings in any GOTO program whose augmented flow graph is reducible. Once the 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 won't be bothered by scoping problems because one identifier is used more than once as a label. To keep things as simple as possible, we begin Phase 1 by inventing a brand-new label for every top-level statement of every block. 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 labelable statement has exactly one label and all of the labels are distinct.

At this point, it is helpful to classify go to statements in yet another way. Consider the statement "go to L", and follow the path λ in the step graph that starts [go to L] $\rightarrow$ [L:] $\rightarrow \cdots$. The path λ is uniquely determined as long as it travels through glue nodes, since glue nodes have out-degree one. If λ enters a subatomic cycle, we shall call the statement "go to L" a *looping* go to. Otherwise, we shall end the path λ at the first non-glue node l that it visits, and we shall call the node l the *arrival node*. If l = [stop], the go to is *halting*; otherwise, l must be atomic, and we shall call the go to *atomic* as well.

Phase 2 deals with the go to's that loop or halt. The looping ones are easy: we just replace them with the statement "repeat endloop". The halting go to's are a little more trouble. We are trying to simplify things by making go to's jump more directly to their destinations. But the language GOTO doesn't allow the padding symbol stop to be labeled. Let GOTO$^+$ denote the extended language in which stop may be labeled. Phase 2 deals with halting go to's by producing a program in GOTO$^+$; in particular, Phase 2 labels stop with yet another distinct label, say Z, and then replaces all halting go to's with "go to Z". When all four preprocessing phases are done, the Forward Elimination Rule will be able to replace each "go to Z" with "exit Z" by wrapping up the whole program into a loop labeled Z. Thus, moving from GOTO to GOTO$^+$ in Phase 2 won't cause us any trouble later on.

Phase 3 has the trickier job of simplifying the atomic go to's. It would be nice if we could arrange that every atomic go to jumped in one step directly to its arrival node. Unfortunately, a go to can arrive at an atomic node that is hidden inside a nested block, as in the example

begin ... **go to** L ... L: **begin** action$_1$; ... **end** ... **end**.

The only way to jump directly to [action$_1$] would be to use an inward go to, which GOTO forbids. We shall deal with this problem by lowering our expectations. Call the atomic go to statement "go to L" *concise* if its arrival node lies somewhere inside the statement labeled L. A concise go to may not jump directly to its arrival node, but it can't wander around too much. By convention, we shall say that a halting go to is concise only if it jumps directly to [stop] and that no looping go to is concise. Phase 2 eliminated all looping go to's and, in our new terminology, made the halting go to's concise; Phase 3 will make the atomic go to's concise. To do so, we shall define a reduction process. Given an atomic go to that isn't concise, we can replace it by another go to that arrives at the same atomic node and passes through fewer glue nodes along the way. Phase 3 repeats this reduction process as often as necessary to make all of the atomic go to's concise.

How do we reduce a go to? Suppose that the jump "go to L" arrives at the atomic node l. Let S_L be the statement labeled "L:", and let B_L be the block of which S_L is a top-level statement. If l lies in S_L, the go to is already concise. If not, let t denote the first node on the path λ from [go to L] to l that does not lie in S_L. There are two possibilities: control either trickles off the end of S_L or it jumps out of the middle of S_L with a go to.

First, suppose that control leaves S_L with a go to. In this case, the node t will be a label node, call it [K:], that lies outside of S_L, while the preceding node on the path λ will be a [go to K] within S_L. This implies that K must label a top-level statement of the block B_L or of some block containing B_L. Since Phase 1 eliminated all name conflicts between labels, the scope of K must include all of B_L; in particular, the identifier K won't be reused as a label in some block nested inside of B_L. Since the "go to L" that we are reducing certainly lies in B_L, we can achieve a reduction by replacing "go to L" with "go to K".

On the other hand, control might leave S_L by sequencing out past its last token. There are six possible tokens that can immediately follow a statement in GOTO: [;], [**end**], [**endloop**], [**else**], [**fi**], and [**stop**]. The node t must be of one of the first three types, since S_L is a top-level statement of B_L. If t = [;], let M be the label of the statement following the semicolon. Because of Phase 1 again, the scope of M must include all of B_L; hence, we can replace "go to L" with "go to M", thereby reducing the jump. Similarly, if t = [**endloop**], we can achieve a reduction by jumping to the first statement of that loop instead of the last. In the remaining case, we have t = [**end**], meaning that S_L is the final statement of a compound. What node could follow t on the path λ? Since t itself ends a statement, it must be followed by one of the six types of nodes listed above. Both [**end**] and [**fi**] nodes themselves end statements, while the edge leaving an [**else**] goes directly to a [**fi**], which ends a statement. Proceed along the path λ from t to the first node following t that isn't [**end**], [**fi**], or [**else**]. This node can't be [**stop**], since λ arrives at the atomic node l. Thus, it must be either [;] or [**endloop**]. We can achieve a reduction by jumping either to the statement following the semicolon or to the first statement of the loop. We conclude that every non-concise 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 replace all of the dead go to statements in the program with null statements. We waited until now 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.

These four phases suffice to banish all pathologies. In particular, whenever the augmented flow graph of a GOTO program is reducible, the four phases will transform that program into a GOTO$^+$ program that is free of head-to-head crossings, as the following proof demonstrates.

Proposition 7. *If all of the go to's in a GOTO$^+$ program P are live and concise, and if the augmented flow graph of P is reducible, none of the go to graphs of P can contain any head-to-head crossings.*

To prove this, suppose that all of the go to's in P are live and concise, but that the go to graph of some block of P has a head-to-head crossing. We shall prove that the augmented flow graph of P has a 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 L and M be the labels of two top-level statements S_L and S_M of B that are involved in a head-to-head crossing of

the form

$$\ldots \textbf{ go to M } \ldots \text{ ; L: } \ldots \text{ ; M: } \ldots \textbf{ go to L } \ldots ,$$

and let λ and μ denote the paths to the corresponding arrival nodes l and m. Neither l nor m can be [**stop**], since all concise halting go.to's jump directly to [**stop**]; thus, both l and m are atomic nodes. By conciseness, the atomic element l lies inside of S_{L} and m lies inside of S_{M}.

Since the node [**go to** L] is live, we can choose a path α in the step graph of P from [**start**] to [**go to** L]; we might as well choose α to be simple. The path α must enter the block B at some point, and the only way to do that is to sequence through the first node of B, which will be either **a begin** or **a repeat**—we shall write it [$\mathbf{bgn/rpt}_B$]. Since α is simple, it can visit [$\mathbf{bgn/rpt}_B$] only once; thus, we can decompose α into the two paths β and γ where

$$\alpha = [\textbf{start}] \xrightarrow[\beta]{*} [\mathbf{bgn/rpt}_B] \xrightarrow[\gamma]{*} [\textbf{go to L}],$$

every node of β except the last lies outside of B, and every node of γ lies within B.

The path γ might include **go to** edges of the form [**go to** K] $\rightarrow$ [K:]; we claim, however, that γ must visit some atomic node after traversing any **go to** edge. Suppose, on the contrary, that γ has the form

$$\gamma = [\mathbf{bgn/rpt}_B] \xrightarrow{*} [\textbf{go to K}] \rightarrow [\text{K:}] \xrightarrow[\delta]{*} [\textbf{go to L}]$$

where δ is free of atomic nodes. From the node [**go to** L], the path λ takes us through glue nodes to the atomic node l; hence, the jump "**go to** K" will arrive at l. By conciseness, the label K must label some statement S_{K} containing l. The statements S_{K} and S_{L} can't be disjoint, since they both contain l. Could we have $S_{\mathrm{K}} \subseteq S_{\mathrm{L}}$? If so, the node [L:] 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 [L:], then reenter S_{K} again to get to l. But a path of glue nodes can't enter the same statement twice without being caught in a subatomic cycle, which would preclude arriving at l. Therefore, we must have $S_{\mathrm{K}} \supset S_{\mathrm{L}}$. But this can't happen either; since S_{L} is a top-level statement of B, the relation $S_{\mathrm{K}} \supset S_{\mathrm{L}}$ would imply [K:] $\notin B$, contradicting the fact that γ lies entirely in B.

Our next claim is that the path γ must visit at least one atomic node and that the last atomic node it visits must lie after [M:] within B. To see this, note that γ manages to get from [$\mathbf{bgn/rpt}_B$] to [**go to** L] while remaining within B. In particular, it starts to the left of [M:] and ends to the right of [M:]. It might go back and forth across [M:] several times; but consider the last time that γ crosses from the left of [M:] to its right. It is enough to show that γ must visit some atomic node after this last cross. To achieve this cross, γ must either visit [M:] or else jump forward across [M:] by traversing a **go to** edge; the only other forward edges in the step graph of a GOTO program are the ones that skip forward over branches of conditionals, and such edges can't carry γ past a top-level label like [M:]. If γ visits [M:] itself on the last cross from its left to its right, it must then follow the path μ to the atomic node m before continuing on to [**go to** L]. Thus, in this case, γ does visit an atomic node after the last cross. Suppose, on the other hand, that γ achieves the last

cross by jumping over [M:] with a forward **go to**. By the result in the last paragraph, γ must visit some atomic node after traversing this **go to** edge, so our claim holds once again.

Let n denote the last atomic node that γ visits; we have just shown that n lies after [M:], which we shall write $[\mathrm{M:}] < n$. Let C denote the simple cycle in the augmented flow graph of P that consists of the static edges from l to n followed by the dynamic edge from n back to l. The augmenting path shows that l is one gateway to the cycle C. We intend to show that the node m lies on C and is also a gateway to C.

Since the cycle C consists exactly of the nodes from l through n inclusive, the problem of proving that m lies on C reduces to the problem of showing that $l \leq m \leq n$. We know, in fact, that $l < m$, so the first inequality holds. We also know that $[\mathrm{M:}] < n$, which means that n lies either in the statement S_{M} or in some succeeding statement. If n doesn't lie in S_{M}, we must have $m < n$. Suppose, on the other hand, that both m and n lie in S_{M}. The statement S_{M} might contain dangling **go to**'s, but it can't contain any head-to-head crossings, by the minimality of B. Thus, we can use the Elimination Rules to replace all of the non-dangling **go to**'s of S_{M} with **exit**'s. Apply Lemma 5 to the resulting statement in EXIT (the resulting statement isn't really in EXIT because it has dangling **go to**'s instead of dangling **exit**'s, but that doesn't affect the proof of Lemma 5). We conclude that the node m, which is the dynamic successor of [**start**] in S_{M}, must also be the leftmost live atomic element in S_{M}. Since the node n is live and, by our current assumption, lies in S_{M}, we must have $m \leq n$. Thus, in either case, the node m must lie on the cycle C.

It remains to show that m is a second gateway to C. For this, we need to use the fact that [**go to** M] is live. As before, let ϵ be a simple path in the step graph of P from [**start**] to [**go to** M], and partition ϵ into ς and η where it enters the block B; we have

$$[\mathbf{start}] \xrightarrow[\varsigma]{*} [\mathbf{bgn/rpt}_B] \xrightarrow[\eta]{*} [\mathbf{go\ to}\ M] \xrightarrow[\mu]{*} m.$$

The image of this path in the augmented flow graph of P is almost enough to show that the node m is indeed a second gateway to C. The only problem is that η might visit atomic nodes of C. We shall consider two cases. If η doesn't visit any atomic nodes at all, it certainly doesn't visit any nodes of C, and we are done. So suppose η visits at least one atomic node, and let k be the last atomic node that η visits. By the same argument that worked for γ, we can see that η must visit some atomic node after traversing any **go to** edge. Hence, after visiting its last atomic node k, the path η can't traverse any **go to** edges. We claim that $k < [\mathrm{L:}]$. Otherwise, the path η would have to jump backward across [L:] to get from k to [**go to** M]. The only backward jumps in GOTO other than **go to**'s lead from an [**endloop**] to the matching [**repeat**]. Looping back around a loop nested inside of B couldn't carry us back over the top-level label [L:]. If B itself were a loop, looping back around B would carry us backward over [L:], but doing so would demand that η visit $[\mathbf{repeat}_B]$ twice, contradicting the simplicity of ϵ. Thus, we must have $k < [\mathrm{L:}]$. This means that the augmenting path from [**start**] to k avoids all nodes of C. Hence, we can show that m is a second gateway to C in this case by following the augmenting path from [**start**] to k and then the dynamic edge from k to m. At long last, the proof of Proposition 7 is complete.

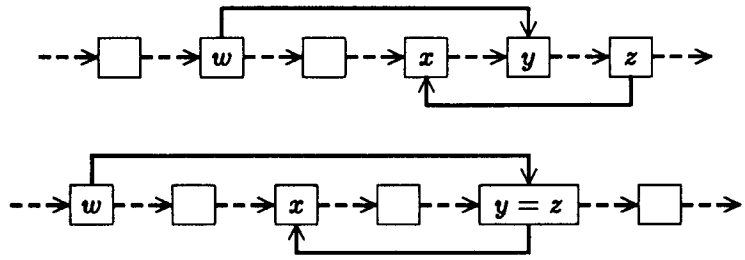


Figure 13: Two examples of conflicting pairs of edges in an augmented flow graph

Corollary 8. *A GOTO program can be translated into EXIT under the policy of structural equivalence if and only if its augmented flow graph is reducible.*

Corollary 9. *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 Section 6; then apply Corollary 8 to that GOTO program.

9. Testing the reducibility of augmented flow graphs

Our theoretical investigations are almost complete, but there is one more point worth making. We have identified two new regularity conditions for programs in GOTO: the reducibility of the augmented flow graph (RAFG) and the absence of head-to-head crossings in the go to graphs (AHHC). These two conditions differ a little in strength: AHHC implies RAFG by Corollary 6, but RAFG allows the go to pathologies while AHHC does not. These two conditions also differ in another way: AHHC is a fairly simple, essentially first order condition, while RAFG seems more subtle and second order. In fact, this latter difference is an illusion. The notion of reduciblity in general is subtle, but the linear order inherent in an augmented flow graph allows us to rephrase reducibility of those graphs as a first order condition, in fact, as a prohibition against certain "conflicting" pairs of edges.

We shall say that two edges $w \to y$ 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 edges are shown in Figure 13. 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 wouldn't count as a head-to-head.

Proposition 10. *An augmented flow graph is reducible if and only if it contains no conflicting pairs of edges.*

If there is a conflicting pair of edges, it is easy to see that the augmented flow graph is not reducible: the simple cycle composed of the augmenting path edges from x to z followed by the edge $z \to x$ 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

edges 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 might as well ignore final nodes. The augmenting path imposes a linear order on the non-final nodes of the flow graph. Let u be the smallest node of C in this order, and let v be the largest non-final 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 edge $t \rightarrow h$ 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: take the assumed edge 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 a 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 doesn't visit any node less than u, and consider the first edge $p \rightarrow q$ on that path whose head q satisfies $q < h$; such an edge must exist since $u < h$. We have $t < u \leq q < h \leq p$; this implies that the edge $t \rightarrow h$ forms a conflicting pair with the edge $p \rightarrow q$, which completes the proof.

10. A case study of arrow stretching

The results in this paper are mostly of theoretical rather than practical 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.

During the latter half of 1982, while I was working at the Computer Science Laboratory of the Xerox Palo Alto Research Center, I decided that I would like to import the spiffy new PASCAL version of Donald E. Knuth's document compiler TEX [13] 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 happens, Edward M. McCreight 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 "Structured Programming with go to Statements" [12] 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 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

```
S ← the empty stack;
for each vertex v in increasing order do
    for each forward arrow f with f.head = v in decreasing order of f.tail do
        while f.tail < top(S) do
            w ← pop(S);
            for each backward arrow 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 fi;
        f.newtail ← top(S);
    push v onto S;
while S is not empty do
    w ← pop(S);
    for each backward arrow b with b.head = w in any order do
        b.newtail ← the last vertex;
```

Figure 14: An algorithm that stretches the arrows of a go to graph

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 [14]. 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 statements were in a funny order 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 remove this head-to-head.

In the course of this effort, several issues of a practical nature arose that are worth mentioning. First, when I was writing the documentation for the enhanced translator, I wanted to describe the class of PASCAL programs that it could handle, but I didn't want to define go to graphs and head-to-head crossings. It isn't hard to show that all of the go to graphs of a GOTO program are free of head-to-heads if and only if the pattern

... go to M ... L: ... ; ... M: ... go to L ...

never occurs in the program text. The reason for including the semicolon in the middle of the pattern is to guarantee that L and M don't both label the same statement; since PASCAL doesn't allow more than one label on a statement, the semicolon is superfluous for PASCAL.

Second, we proved in Section 4 that, in the absence of head-to-heads, the arrows of a go to graph could always be stretched so as to eliminate all crossings; but we didn't give an efficient algorithm. Figure 14 shows the algorithm that I implemented. Each edge is represented as a record with the three fields *head*, *tail*, and *newtail*, the last of which stores the tail of the edge after it has been stretched. The variable S denotes an auxiliary stack of vertices. 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 corresponding MESA target as a tree of program fragments. The additional time needed to stretch arrows and apply the Elimination Rules in that context was 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.

Third, what about the non-local 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 non-local 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, non-local go to's have another problem as well: it is hard to predict where they are going to come from. Think about a non-local go to from the point of view of the destination label L, which labels a top-level statement of some block B. A non-local "go to L" doesn't occur textually within B at all. Instead, there is a call on some procedure P within B, and the "go to L" appears in the body of P or in the body of some procedure that P calls. We shall say that a non-local go to is *forward* if the call on P occurs before the label "L:" in B, else *backward*. By proper use of the MESA exception machinery, we can prepare for either forward or backward non-local go to's to L. In each case, we must introduce a new block that is enabled to handle the exceptional condition. One end of this new block must be at the label "L:", while the other end must be far enough away so that no non-local go to's 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 are likely to generate head-to-head crossings.

I don't know of any good solution to this problem. In McCreight's translator, I chose to assume that no backward non-local go to's would ever happen, but that forward non-local go to's might come from anywhere. This worked fine for TEX, since TEX has only one non-local go to statement in it: a forward non-local 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 non-local go to's.

Acknowledgments

I would like to thank Greg Nelson for numerous 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 (January 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." *Comm. ACM* **9**, 5 (May 1966), 366–371.
4. Dijkstra, E. W. "**Go to** statement considered harmful." *Comm. ACM* **11**, 3 (March 1968), 147–148.
5. Harel, D. "On Folk Theorems." *Comm. 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*, third edition, revised for the ISO PASCAL Standard by Mickel, A. B. and Miner, J. F. 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, NJ, 1978.
11. Knuth, D. E. and Floyd, R. W. "Notes on avoiding **go to** statements." *Inf. Proc. Lett.* **1**, 1 (February 1971) 23–31.
12. Knuth, D. E. "Structured Programming with **go to** Statements." *Comput. Surveys* **6**, 4 (December 1974), 261–301.
13. Knuth, D. E. *The TEXbook.* Addison-Wesley, Reading, MA, 1983.
14. Knuth, D. E. *TEX: The Program.* Addison-Wesley, Reading, MA, 1985.
15. Kosaraju, S. R. "Analysis of Structured Programs." *J. Comput. System Sci.* **9**, 3 (December 1974) 232–255.
16. Ledgard, H. F. and Marcotty, M. "A Genealogy of Control Structures." *Comm. ACM* **18**, 11 (November 1975), 629–639.
17. Mitchell, J. G., Maybury, W., and Sweet, R. *MESA Language Manual*, technical report CSL-79-3, Xerox Palo Alto Research Center (April 1979).
18. Peterson, W. W., Kasami, T., and Tokura, N. "On the Capabilities of **while**, **repeat**, and **exit** Statements." *Comm. ACM* **16**, 8 (August 1973), 503–512.
19. Ramshaw, L. Problem 83–1, *J. of Algorithms* **4**, 1 (March 1983), 85–86.
20. United States Dept. of Defense. *Reference Manual for the ADA Programming Language.* Springer-Verlag, New York, 1983.
21. Wirth, N. *Programming in MODULA-2*, second edition. Springer-Verlag, New York, 1983.

Index

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Report #1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Report #2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Report #3, January 11, 1985.

"Eliminating go to's while Preserving Program Structure."
Lyle Ramshaw.
Report #4, July 15, 1985.

Selected Publications by SRC Members

"Implementing Remote Procedure Calls."
Andrew Birrell and Bruce Nelson.
ACM Transactions on Computer Systems, February 1984.

"Coin Flipping in Many Pockets."
Andrei Broder and Danny Dolev.
Proceedings of the 25th Symposium on Foundations of Computer Science, 1984.

"Pessimal Algorithms and Simplexity Analysis."
Andrei Broder and Jorge Stolfi.
SIGACT News 16(3), Fall 1984.

"The Alpine File System."
Mark Brown, Karen Kolling, and Ed Taft.
Xerox PARC Technical Report CSL–84–4, 1984.

"Fractional Cascading: A Data Structuring Technique with Geometric Applications."
Bernard Chazelle and Leo Guibas.
ICALP Twelfth International Colloquium, 1985.

"Visibility and Intersection Problems in Plane Geometry."
Bernard Chazelle and Leo Guibas.
ACM Conference on Computational Geometry, 1985.

"Bulldog: A Compiler for VLIW Architectures."
John Ellis.
Yale University, Department of Computer Science Research Report 364, 1985.

"Tools: An Environment for Time-Shared Computing and Programming."
John Ellis, Nat Mishkin, Mary-Claire van Leunen, and Steve Wood.
Software: Practice & Experience, October 1983.

"A Kinetic Framework for Computational Geometry."
Leo Guibas, Lyle Ramshaw, and Jorge Stolfi.
IEEE 24th Annual Symposium on Foundations of Computer Science, 1983.

"An Introduction to the Larch Shared Language."
John Guttag and Jim Horning.
IFIP 9th World Computer Congress, 1983.

"Combining Algebraic and Predicative Specifications in Larch."
Jim Horning.
Formal Methods and Software Development II. Springer-Verlag Lecture Notes in Computer Science 186, 1985.

"The Future of Thinking for Non-Thinkers."
Jim Horning.
Computer, July 1984.

"Hints for Computer System Design."
Butler Lampson.
IEEE Software, January 1984.

"Experience with Grapevine: The Growth of a Distributed System."
Mike Schroeder, Andrew Birrell, and Roger Needham.
ACM Transactions on Computer Systems, February 1984.

"Finding the Convex Hull Facet by Facet."
Garret Swart.
Journal of Algorithms 6(1), March 1985.

Eliminating go to's while Preserving Program Structure

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301