STATIC RESTRICTION OF THE GOTO STATEMENT*

T. Bishop, M. Bodenstein,

and R. Conway

TR77-325

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work was supported in part by the National Science Foundation under grant MCS77-08198.

<u>Static Restriction of the GOTO Statement*</u>

T. Bishop, M. Bodenstein, and R. Conway
Department of Computer Science
Cornell University

Although GOTOs in general have been indicted, it is actually the case that some uses of the GOTO are more destructive of good program structure than others. Particularly in languages designed before the age of enlightenment, a flat prohibition of the use of the GOTO can necessitate various circumlocutions that are not conspicuously superior to that which they replace. Recently we have defined a PL/I dialect which retained the GOTO (for reasons of compatibility) but which attempted to exclude its most damaging uses by means of two simple restrictions:

1. The target of a GOTO must come after the GOTO statement -- so all branches are forward.

2. A branch cannot enter a compound construction, i.e. a DO-loop, a DO-group (non-iterative) or SELECT-group.

For example:

```
A: DO;
    L4:;
    G3:GOTO L3;
    B: DO;
        G1: GOTO L1;
        G2: GOTO L2;
        L1:;
        L3:;
        END B;
    L2:;
    G4: GOTO L4;
    END A;
```

Statements G1 and G2 are allowable, but G3 is not (it violates restriction 2) and G4 is not (it violates restriction 1). In the course of implementing a compiler for this language, we realized that the second restriction has some interesting properties.

---

* This work was supported in part by the National Science Foundation under grant MCS77-08198.

The question arises in the <u>static</u> enforcement of the second restriction. The restriction can be enforced dynamically (checked at runtime -- PL/C has done so for years), but we consider it preferable to detect violation from syntactic analysis of the program text, without requiring its execution* It is possible to do so, but the algorithm is not entirely obvious, and it possesses some surprising characteristics.

Note first that there can be numerous GOTO statements referring to the same target label, and that these GOTOs can be at different nesting levels, so long as none are at a lower (outer) nesting level than the target. Then note that the static nesting level of GOTO and target alone are not sufficient to determine violation of restriction 2. For example:

```
C: DO;
    D: DO;
        G5: GOTO L5;
        END D;
    E: DO;
        L5:;
        END E;
    END C;
```

G5 violates restriction 2 since it attempts to enter construction E, although D and E are at the same static nesting level. A more complicated example is shown below:

```
R: DO;
    G6: GOTO L;
    S: DO;
        G7: GOTO L;
        T: DO;
            G8: GOTO L;
            END T;
        G9: GOTO L;
        END S;
    U: DO;
        G10: GOTO L;
        END U;
    L:;
    END R;
```

All of the GOTOs in this example have a common target and all are valid.

----

* Most ALGOL compilers enforce restriction 2 statically by shielding the loop body statement(s) in a "scope." Any references from outside the loop to a label within the loop go unresolved and are reported at the end of the procedure. We prefer to report illegal label placement at the point of label definition.

Each target label and the GOTO references to that label are obviously completely independent of every other label and its references, so each constitutes a separate checking problem and it is sufficient for us to examine the problem here in terms of a single label and its references.

Definition: A "reference-label set k" consists of target label k, and n>=1 GOTO references to label k.

The set may be considered to end with the definition of label k -- any later reference is clearly invalid due to restriction 1. A non-trivial set must begin with a reference to label k. So the interesting case for discussion is a set of one or more references to k, followed by the definition of k itself. All other checking is obvious and trivial.

Presumably, each GOTO should be checked to see that it is not inconsistent with its predecessors in the set, and the definition must be checked to see if its position (with respect to nesting level) is consistent with the preceding references.

There are several possible ways of accomplishing this checking, but all reasonable algorithms must capitalize on the implications of the following lemmas:

Definition: We use the term "block" to mean a compound construction in a program -- a compound statement, a loop or a SELECT group in PL/I terms.

Notation: Let Rik denote the ith reference to label k.

Let Dk denote the definition of label k.

Let Sik denote the set of blocks that are open at the point of Rik. Note that Rik is internal to every block in Sik.

Let Bk denote the innermost block containing Dk.

Lemma 1: Reference Rik and definition Dk satisfy restriction 2 if and only if Bk is a member of Sik.

Proof:

(=>): If Bk is not a member of Sik, then Rik is not internal to Bk, hence Rik is an "entering reference" -- improper under restriction 2.

(<=): If Bk is a member of Sik, then Rik is internal to Bk and the reference is valid under restriction 2.

For a complete reference-label set to be valid with respect to restriction 2, it follows from Lemma 1 that Bk must be a member of Sik, for all i=1,2,...,n. That is, the definition must be valid with respect to all the preceding references. What is initially surprising is to discover that S1k dominates all of these sets:

Lemma 2: References Rik, i=2,3,...,n, and definition Dk satisfy restriction 2 if and only if Bk is a member of S1k.

Proof:

(=>): follows directly from Lemma 1.

(<=): (by contradiction) Assume Rik and Dk don't satisfy restriction 2 for some i,i=2,3,...,n. By Lemma 1, Bk is not a member of Sik. Since Dk occurs after Rik (by the assumption that restriction 1 is satisfied), the only way this can occur is if Bk is not yet open at the point of Rik. (It could not already be closed.) But this implies Bk is likewise not yet open at the point of R1k, since R1k precedes Rik. Hence Bk is not a member of S1k. But we assumed Bk is a member S1k. Therefore our assumption of an Rik that doesn't satisfy restriction 2 must be invalid.

In effect, Lemma 1 says that a reference to a label is legal if and only if it is internal to the block that contains the label definition, while Lemma 2 states that if a reference and a label are internal to some block, then all points between the two are internal to that block.

The surprising consequence of Lemma 2 is that references after the first require no checking whatsoever. It is not possible for subsequent references to be inconsistent with the first reference, nor is it possible for them to influence the validity of the subsequent definition.

Given these lemmas, all that is required is some method of recording the nesting structure that exists at the time of the first reference, and then verifying that the subsequent definition is positioned in one of the blocks that was open at the time of the first reference. There are various ways to do this. For example:

```
for each procedure do

    if stmt is "GOTO Ki" and this is first reference to Ki
       then (* record nesting level and block number of    *)
            (* first reference                              *)
               first_reference_nesting(i) =
                  nest_level of current_block
               first_reference_block(i) = current_block

    if stmt is "DO"
       then (* make the new block a son of the current      *)
            (* block                                        *)
               father_block = current_block
               current_block = next_available_node
               next_available_node = next_available_node + 1
               father of current_block = father_block
               nest_level of current_block =
                   (nest_level of father_block) + 1

    if stmt is "END"
       then (* make the father of the current block the     *)
            (* new current block                            *)
               current_block = father of current_block
               father_block = father of current_block

    if stmt is "Ki:"
       then (* trace up the tree from the node where the    *)
            (* first reference occurred to the level of     *)
            (* the current node                             *)
               difference = first_nesting_level(i) -
                               nest_level of current_block
               ancestor_block = first_reference_block(i)
               for j = 1 to difference do
                  ancestor_block = father of ancestor_block

            (* if the current block is an ancestor of the *)
            (* first reference block, the label is valid  *)
               if ancestor_block ≠ current_block
                  then error('illegal label position')
                  else (* label is okay *)

    end
```

The above algorithm actually maintains a tree representing the nesting structure of the blocks. Another algorithm could use an idea from Knuth. Assume that the maximum number of first level blocks within a single block (maximum number of "sons" for a given "father") is n. Then the father of node z is $\lfloor z/n \rfloor$ and its sons are nz, nz+1, nz+2, ... ,nz+n-1. When the label definition is encountered, verify that the node where the label definition occurred (Zl) is smaller than the node where the first reference occurred (Zr). Then, if Zl = $\lfloor$Zr/n**difference$\rfloor$ (where difference is computed as in the above algorithm), the label definition is valid. Again, it's simply a matter of recording the nesting structure of the various blocks and verifying that the block where the label definition occurred is an "ancestor" of the block containing the first reference.

References

1. Conway, R., "PL/CS - A Highly-Disciplined Subset of PL/C", *SIGPLAN Notices*, December 1976.

2. Knuth, Donald E., *The Art of Computer Programming*, Volume III, Addison-Wesley, pp. 144-145, 1973.

3. Djikstra, E.W., "GOTO Statement Considered Harmful", *Communications of the ACM*, March 1968, pp 147-148.