ACM SIGPLAN notices, Volume 8,Issue 2, pp. 28-34



# GLOBAL VARIABLE CONSIDERED HARMFUL

W. Wulf, Mary Shaw
Carnegie-Mellon University
Pittsburgh, Pa.

In 1968 E. W. Dijkstra wrote a letter to the editor of the CACM [1] proposing that the **goto** statement be abolished from all "higher level" programming languages. Although this suggestion has not met with universal acceptance, we would like to nominate another well-known language construct as a candidate for abolition: the non-local variable.

We claim that the non-local variable is a major contributing factor in programs which are difficult to understand. For the moment we wish to keep the phrase "non-local variable" somewhat vague. Roughly, however, we mean any variable which is accessed, and particularly modified, over a relatively large span of program text. More specifically, we mean any variable referenced in a segment of program, S, such that not all uses of that variable are contained in S. We always intend conceptual locality, rather than textual locality; it may be the case that a single conceptual unit, (e.g., control of a loop) has elements that are separated by a number of lines of other text, but that the interspersed text can be treated as a single element and "collapsed" for purposes of understanding the surrounding unit. If the text is properly displayed (e.g. with proper indentation), the physical distance between such elements need not interfere with their (conceptual) locality.

Since the rationale behind this suggestion is similar to that for banishing the **goto**, we will begin by briefly paraphrasing the arguments against the **goto**. We must admit certain limitations on our intellectual powers - in particular, that we are better able to cope with static relations among objects than dynamically evolving ones. Since the text of a program is static but its execution is dynamic, anything which destroys or obscures the mapping between textual relations (largely concatenation) and execution relations (the "next" instruction) magnifies the difficulty of doing that which we are least equipped to do. While all forms of control transfer are indicted by this argument to some extent, control constructs such as **if-then-else**, **for**, **case**, etc., are reasonably acceptable because of the syntactic cues they provide. When a program is written using these explicit, well-structured control commands, it is reasonably easy to determine the antecedents of any given command, and hence to deduce the assumptions in effect at that point in the program. When **goto**'s are permitted, this task is difficult because the predecessors of a point in the program are hard to locate. Arguments for and against the **goto** may be found in [2-4]; the extensive bibliography in [2] should be helpful to anyone interested in further reading on the subject.

Note that this argument does not condemn every use of the **goto**; it is certainly possible to find programs which use the **goto** and are not difficult to understand - particularly if the programs are small, the targets of the **goto**'s are close to the jump points, and the control paths established by the **goto**'s are standard ones. The argument below concerning the non-local variable is similar in that it asserts that nonlocal variables may be misused - not that every use obscures program structure.

Consider what it means to "understand" a program. Suppose we are presented with a piece of program text together with the domain of its input parameters. When the program is applied to a particular set of input values (executed), a particular sequence of states (a set of variable values) results. Generally, different state sequences will result depending upon the choice of input parameters; we can thus speak of the class of all possible computations performed by a program as the set of all executions produced by the program under all valid combinations of input parameters. There is clearly a correspondence between the text of a program and the executions it produces, although the correspondence may not be obvious. It is a knowledge of this correspondence that is crucial to "understanding" a program.

Since the class of all possible computations of most programs is large, "understanding" directly in this sense is not within the range of our mental capabilities. We employ instead various devices to abstract from the specific actions of a program to the effects of those actions. Abstraction is often achieved by selectively ignoring information and modelling only the effects which we believe are relevant to succeeding stages of the computation. A program may have many effects other than the outputs of interest, but we do not regard an understanding of these effects as crucial to understanding the program.

We can formalize the notion of abstraction. Suppose S is a segment of program text and P and P' are logical propositions such that if P is true prior to the execution of S, then P' will be true after its execution. Then the transformation S:P→P' is a characterization of the effect of executing S, i.e. an abstraction of S.

It is clear that in the abstraction S:P→P', the proposition P must contain components describing the state of all variables used in S but not local to S. Further, P' must contain components describing the effect on all non-local variables modified in S. P and P' should not mention non-local variables which are not used is S; the inclusion, and particularly the checking, of such variables will increase the complexity of understanding S beyond all reasonable bounds. Thus the complexity of the abstraction is directly related to the number of non-local variables used, and we should expect the mental effort necesary to form an abstraction to increase with the number of non-local variables.

Note that the variables whose states are described by the logical propositions P and P' associated with any given segment of text S will not in general be the same variables described by the propostions associated with adjacent segments of text. In order to check the correctness, or even the plausibility, of a program, it is necessary to ensure that the proposition P is valid - that the variables in P actually have the values claimed for them. In a straight-line program this would present no problem: it would be sufficient to scan backward through the program until a proposition involving each of the variables in question was found. However, straight-line programs are rare (except in languages such as APL where much of the control is implicit), so it is necessary to check backward along all possible control paths until an assertion about each variable of interest is found in every path. The complexity of this task increases exponentially with the distance of the last previous use of each variable along each control path. Just as well-structured control operators help to identify which assumptions are in effect at any point in a program, appropriate scope rules (naming operators) will aid in locating the relevant values of variables.

We might consider minimizing the number of non-local variables being considered at any instant by choosing a smaller segment of text - thereby hoping to minimize the complexity of the abstraction. This scheme is of limited use, since after a point the combinatorial complexity of keeping track of the remaining non-local variables along various control paths leading to S may again increase the complexity of P.

It is not possible to avoid these issues by making global statements about global variables. Statements of intent certainly help the reader to understand a program, but they are not sufficient. In programming languages that do not permit simultaneous assignments, any global statement describing a relationship between two variables will be momentarily false each time the variables are altered. Consider, for example:

```
integer i,j;
comment j is 2*i;
    .
    .
i := <some value>;
comment the assertion is false at this point;
j := 2*i;
comment the assertion is true again;
    .
    .
```

Since programming errors may arise in the course of maintaining the asserted correspondence, the global statements alone are not sufficient.

To illustrate the problems created by the non-local variable, we suggest the following exercise. We give below an un-commented Algol procedure which implements a simple numerical technique. The procedure is small and purports to be "well-structured". We suggest that you study it to determine precisely how it works. As you do so, ask yourself what it is about the procedure that makes the understanding task more difficult than the size of the program seems to warrant.

```
real procedure  q(f,a,b,i,e1,e2);
    value a,b,i,e1,e2;
    real procedure f;
    real a,b,i,e1,e2;
        begin
        real cf,nf,s,i1,x;
        cf ← f(a); s ← (f(a-i) - cf)/i; i1 ← i; Q ← 0;
        for x ← a step i until b-i do
            begin
            nf ← f(x+i);
          if abs (cf + s*i - nf)/nf > e1
              then begin x ← x-3*i/2; i1 ← i/2 end
              else begin
                  if abs (cf + s*i - nf)/nf < e2 then i1 ← i*2;
                  Q ← Q + i*(cf + nf)/2;
                  s ← (nf - cf)/i;
                  cf ← nf
                  end;
          i ← i1
          end;
      q ← Q
      end;
```

Examples and Related Issues Concerning Block Structure

The most familiar mechanism through which the non-local variable is implemented is Algol-like block structure. Although this technique for defining scopes is superior to none at all, in this section we would like to point out some limitations and problems of both the non-local variable and Algol-like scope rules.

Not all the problems posed by these examples are direct consequences of the locality of names. We believe that the entire issue of programmer control over the naming environment is important, and we would like to see available a variety of techniques for controlling the scope of names. Algol-like block structure is one such technique (but not the only one) and certain problems arise as a consequence of this particular scope rule. We therefore give examples directed at deficiencies of block structure as well as at the perils of non-locality.

**Side effects:** The problems associated with a procedure's making changes in the values of non-local variables are well known. See, for example, [5]. Untold confusion can result when the consequences of executing a procedure cannot be determined at the site of the procedure call.

**Indiscriminant Access:** Block-structured scope rules do not permit a programmer to restrict the use of a variable to a subset of the code at the block level where the variable is declared. Suppose that we want to implement a stack, sharing a vector STACK and a pointer STKPTR among several procedures which operate on the stack. The natural implementation is:

```
begin
    real array STACK [1:100]; integer STKPTR;
    procedure initialize; STKPTR ← 0;
    procedure push (x); real x;
        begin STKPTR ← STKPTR + 1;
            STACK[STKPTR] ← x end;
    real procedure pop;
        begin pop ← STACK[STKPTR];
            STKPTR ← STKPTR - 1 end;
    boolean procedure empty; empty ← STKPTR = 0;


    <calls on initialize, push, pop, empty, etc.>


end
```

In this implementation the variables STACK and STKPTR are accessible to the rest of the program. They cannot be protected by adding block structure, for that would also protect the procedure names.

Vulnerability: When the scope rules permit many nested levels of locality, any program segment which did not actually declare its variables has lost all control over the assumptions under which it executes; the programmer can interpose (perhaps inadvertently) new definitions of the non-local variables Consider:

```
A: begin integer x;
      .
      .
        B: begin
              <use of x>
           end;
      .
      .
   end
```

where B is possibly deeply nested within A and uses of x are "rare". It is entirely plausible that a programmer might discover at some block level that he needs a new temporary variable and create:

```
A: begin  integer x;
      .
      .
      C: begin real x;
           .
           .
             B: begin
                   <use of x>
                end;
           .
           .
           end
      .
      .
   end
```

thereby destroying the binding of the inner x. If a programmer wishes to introduce at some level a variable which will be used at some (possibly deeply nested) inner level, he must be aware of the names used at all intervening levels.

The problems of indiscriminant access and vulnerability are complementary: the former reflects the fact that the declaror has no control over who uses his variables; the latter reflects the fact that the program itself has no control over which variables it operates on. Both problems force upon the programmer the need for a detailed global knowledge of the program which is not consistent with his human limitations.

No overlapping definitions: In some calculations, each program segment depends only on the output values of its immediate predecessor. These variables may be large arrays of various shapes. In such cases, variables should be accessable to only the two program segments concerne and inaccessible or invisible to others. Unfortunately, the block structure discipline provides no way to describe such overlapping definitions, and global strategies make the variables too vulnerable.

Other anomalies of block structure produce situations where the interpretation of the use of a variable is hard to understand. One such difficulty is the "hole-in-the-scope" problem described by Knuth [6].

### What Then?

Having condemned the non-local variable in general and Algol-like scope rules in particular, it behooves us to examine the alternatives. In the case of the **goto** the primitive notion of "transfer of control" is not condemed as such. The argument is rather that the textual representations (syntax) which indicate that such transfers are to occur must enforce textual locality of the transfers. A similar assertion applies to the issues we are raising about scopes of names. We do not condemn the primitive notion of controlled access to a name. Rather we argue that the textual representation of such control should be explicit and localized.

As with the **goto** proposal, the present proposal should not be construed as a trivial quibble over syntactic facilities. Rather, we are attempting to extend the analysis of what constitutes well-structured programming. Issues of whether specific syntactic features should or should not be in a language in order to promote good program structure are of lesser importance. In the case of Algol we have exhibited both features which permit the construction of ill-structured programs and the lack of features which would allow well-structured access patterns to be expressed. Although this appears to call for a change in the scoping syntax of languages, our concern is with the semantics of languages and programs written in those languages -- not with syntax.

Unlike the **goto** situation, the scope problem does not have 1) a single offending construct whose elimination will correct the problem and 2) a collection of familiar constructs which are (almost) adequate to capture the essential uses of the underlying primitive concept. We do not have a specific proposal for a set of constructs to replace or augment conventional scope rules. We can, however, enumerate some of the attributes that a suitable alternative would have:

- The default should not be to extend the scope of a name to inner blocks.
- The right to access a name should be by mutual agreement between creator and accessor.
- Access rights to a structure and to its sub-structures should be decoupled.
- It should be possible to distinguish different types of access.
- Declaration of definition, name access, and allocation should be decoupled.

We would like to submit the question of what constitutes appropriate alternative rules to the language community.

We also believe the following questions should be addressed:

- Are there criteria, other than those listed above, which should be used to judge a scoping mechanism?
- In what sense is a set of scope rules "complete"?
- What classes of algorithms are precluded by Algol-like (or other) scope rules?

- The discussion above is framed soley in terms of scope. To what degree does "extent" interact with these considerations? In particular, what about
  - implicit vs. explicit allocation
  - retention vs. deletion

- Many of the objections raised above have been addressed in the context of protection within operating systems. Is the analogy to languages stronger or weaker than apparent?

References

[1] Dijkstra, E.W., "Goto statement considered harmful," letter to the editor, CACM, 11, 3, March 1968.

[2] Leavenworth, B., "Programming With[out] the Goto," Proceedings ACM National Conference, August 1972.

[3] Hopkins, M., "A Case for the Goto," Proceedings ACM National Conference, August 1972.

[4] Wulf, W., "A Case Against the Goto," Proceedings ACM National Conference, August 1972.

[5] Knuth, Donald E. "The Remaining Trouble Spots in Algol 60." CACM 10,10 Oct. 1967, pp. 611-618.

[6] Knuth, Donald E. "Algol 60 Confidential." CACM 4,6 June 1961, pp. 268-272.