The place of own variables in programming language theory

B. Higman

Department of Computer Studies, University of Lancaster, Bailrigg, Lancaster

An exact definition of the scope of an own variable has not yet evolved. In the past this has given rise to troubles. This paper shows that an assumption that the scope of an own variable does not extend beyond the scope of the name of the owning procedure solves many of the problems and leads to a clearer understanding of their true nature. They are equivalent to additional parameters which have been 'frozen' to local generators.

(Received April 1975)

Own variables are variables created by the declaration of a (block or) procedure whose scopes are wider than the body of the said procedure, but which are protected from outside interference by their anonymity other than within it. This definition is unlikely to be disputed except that it leaves open the question as to what the scope of these variables actually is. A specific instance of the use of a published agorithm using own variables gave rise to troubles, elucidation of which led to the conclusions reported here. Contrary to popular ALGOL 60 philosophy, the continued existence of an own variable *need not* extend beyond the scope of the name of the owning procedure. The assumption that it *does not* solves many of the problems associated with these variables, and, it is suggested, leads to a clearer understanding of their true nature.

The specific instance referred to occurred in an attempt to solve a puzzle from a Sunday paper by the bull-at-a-gate method of writing

begin [1:*n*] **amode** *activities, boys, girls*; (A)
 for *all permutations of boys* **do**
  for *all permutations of girls* **do**
   if *conditions met* **then** *print all arrays* **fi od od end**

and expanding this in the manner approved by the structured programming approach. Neither ALGOL 68, in which this pretends to be, nor any other language of which the author is aware, implements 'all permutations' as an option in a for-statement.

Two methods of dealing with this suggested themselves. In the first, **while** versions of the for-statements are used

while *newperm of boys* **do** (B)
 while *newperm of girls* **do** *etc.*

In the second, by means of a procedure developed from

proc *permact* = (**int** *n*, **ref** [] **amode** *A*, **proc** *act*) **void**:
 (**for** *all permutations of A*[1] *to A*[*n*] **do** *act*)

the program is reduced to

permact(*n*, *boys*, *permact*(*n*, *girls*, *S*)) (C)

where *S* is the statement in the fourth line of A.

Permutation generating algorithms have been reviewed by Ord-Smith (1971) and Page (1971). All the algorithms quoted by the former accept the array to be permuted as a parameter, but refer also to a boolean and to an integer array, the former of which is treated as global and the latter declared own. In ALGOL 68 the first line of B could be written

while (*nextperm*(*boys*); *first*) **do**

(the outer brackets being strictly unnecessary, and *first* being the global boolean) but the second line demands a different boolean and a different own array if the method is to work. Thus the common format in which Ord-Smith presents the algorithms he reviews is unsuited to the present problem.

Linguistic considerations

It is unaesthetic, to say the least, and mildly uneconomic as well, for a program to contain two copies of the same procedure (under different names, of course). The prospect that this may be necessary actually arises in two distinct ways. In the first place, as things stand, if the arrays *boys* and *girls* are of different modes, then nothing can save us from compiling and storing two completely separate procedures identical in all respects save one, that *nextperma* (or *permacta*) refers to **amode** and *nextpermb* (or *permactb*) refers to **bmode**. This is because current languages do not permit a mode to be supplied as a parameter—as would be seen at its simplest in

proc *transpose* = (**mode amode**; **ref amode** *p*, *q*) **void**:
 (**amode** *r*; *r* := *p*; *p* := *q*; *q* := *r*)

—though this is nastier than it looks; at compile-of-call time, the first parameter must be known before the second and third can be checked. Nor do united modes in ALGOL 68 seem able to help. The matter is, however, already under consideration by Lindsey (1974) and here we can evade the issue by pointing out that in many cases (where **amode** involves several words of storage) little will be lost in efficiency by keeping the array intact and permuting the subscripts. That is, we actually permute an auxiliary array [1:*n*] **int** *aux*, and call the members of the main array by *A*[*aux*[*i*]]. (Incidentally, this also makes permutation of specified subsets no problem).

But this duplication of code reappears in B if two copies of the permutation algorithm are required, no longer because of separate modes but because of the need for separate own arrays. The investigation began when the question was raised, whether B can be written in the following form

while *newperm*(*boys*) **do** (B′)
 while *newperm*(*girls*) **do** *etc.*

in order to avoid this duplication. That the immediate answer was 'no—because separate own arrays are required' led to a new look at the nature of own arrays (and other variables) and to an interpretation which, if accepted, leads to clear (and adaptable) answers to all the difficult questions such as the bounds of dynamic own arrays, and the single or multiple existence of own variables in recursive situations.

The matter was considered in the context of a number of others, one of which was the independent compilation of procedures. Global variables present a barrier to independent compilation; so also does any system which assumes that a procedure knows its own block level. Block level is conventionally available so that it can be used, in conjunction with '*display*', for accessing globals, whence a procedure without globals probably has no need to know its own block level, and if all procedures are without globals we can probably dispense with '*display*' as well. Globals can be eliminated from any procedure by making them into parameters, at the cost of quoting them explicitly at each call; this cost can be eliminated if means are available to freeze parameters. One is therefore led to consider that globals should be treated as additional parameters which are immediately frozen (i.e. at declaration time).

This concept permits the assumption that the machine code

representing a procedure need deal only with a structure situated on top of the stack and accessed through two pointers, CHAIN and STACK. (Both pointers are necessary since the distance between them depends on the current size of dynamic arrays and is unknown at compile time). The structure is made up as follows—since the time sequence is down the page, the 'top' of the stack is at the bottom of the printed layout:

(part of the stack in prior use)	Initially, STACK points immediately above this.
PARAMETERS	Evaluated from left to right in the main program, making free use of the space above STACK but leaving each one in position when its evaluation is complete and commencing the next from a new position of STACK immediately above it.
GLOBALS ADDRESS OF CODE	Copied unchanged from a list prepared at the time the routine was declared. (The address is retained for use by error diagnostics and, if the heap is in use, the garbage collector).
CHAIN: LINK INFORMATION	Placed on the stack by a system routine which chains CHAIN at the current position of STACK, stacks the return address, etc. and jumps into the code.
FIXED LOCALS	Either as the first action in the routine, or prior to the jump using a parameter from the routine, STACK is raised to allow a fixed space for each identifier declared locally. In the case of procedure identifiers this is two words, and in the case of arrays, space for a dope vector.
VARIABLE LOCALS	Array bounds are computed, the dope vectors filled in and STACK is raised. In the case of procedures, globals are evaluated on the stack, the address of the code added, and the 'procedure dope vector' filled in with a pointer and a length showing what has to be copied at call time.
STACK: (free space)	

Thus a procedure which never had any globals will be represented by two words in FIXED (the pointer into VARIABLE, and length = 1), and one word in VARIABLE (the pointer to the code). The conversion of globals into parameters and their subsequent freezing is something that can be organised by a compiler in the case of procedures written explicitly into a program, but in the case of independently compiled procedures we shall want the *language* to provide for the freezing of such parameters as are intended to be globals, and if it provides for this then it should provide for freezing of parameters as generously as possible. In terms of the stack layout protrayed above, this means a notation for pushing the line between GLOBALS and PARAMETERS up the page (down the stack); in language terms it means a new declaration syntax in which, say,

proc *newname* = *oldname* **freezing** (⟨*parameters from the right*⟩)

has the effect of adding a new pair in FIXED for the new name pointing to a new area in VARIABLE which is a copy of that belonging to the old name with its list extended. This feature is well known to users of POP2 (Burstall and Popplestone, 1968); in view of the extra facilities it provides, we regard it as preferable to **external** declarations which use the linking loader to achieve freezing by lexicographic matching. It is not in ALGOL 68, although there are proposals in Lindsey (1974).

The nature of own variables

One more feature is necessary to satisfy in full the desire for a solution to our original problem that is free of the loose ends of the sort that lead to programming errors. Consider the following:

```
proc perms = (ref [] int a1, a2; ref bool first) bool: ⟨body⟩;
[1:5] int boys, girls,
  bx, gx; bool by, gy;
proc newboys = perms freezing (boys, bx, by);
proc newgirls = perms freezing (girls, gx, gy);
for i to 5 do bx[i] := gx[i] := 0 od; by := gy := true;
while newboys do etc.
```

In this formulation all is clear for the implementation of B without duplication of code. What we have lost is the automatic and anonymous declaration of the own array as seen from the main program. What we have gained is the separate 'own' arrays for the two procedures we use in the while-statements without duplicating the rest of the code. We can secure our gains without incurring our losses if we can cause the freezing process to *create* an *anonymous* array for the second parameter. This is what ALGOL 68 does by its concept of a local generator —or would do if it entertained the symbol **freezing**—allowing us to omit the third and sixth lines above and replace the fourth and fifth by

```
proc newboys = perm freezing (boys, loc [1:5] int :=
  (0, 0, 0, 0, 0), loc bool := true);
proc newgirls = perm freezing (girls, loc [1:5] int :=
  (0, 0, 0, 0, 0), loc bool := true)
```

This has a simple realisation in our stack model—whereas 'boys' appears among the parameters-becoming-globals as a copy of the original dope vector of *boys*, the second parameter appears as a dope vector which is filled to point a short distance *ahead*, where new space is created for the elements as at any ordinary array declaration. The section of VARIABLE LOCALS which results from this declaration has three subsections

TWO DOPE VECTORS AND BOOL
ADDRESS OF CODE
SPACE FOR 'OWN' ELEMENTS

only the first two of which are included in the area pointed to by the two-word object in FIXED LOCALS, and therefore only the first two of which are copied to the top of the stack as the first stage of a call.

It has long seemed to the author (Higman, 1967) that the crudity of the interpretation of ALGOL 60 own variables as global (even cosmic?) was the source of the trouble they have given, and slightly absurd that own variables should outlive and outlast not merely the activity (call), but even the existence (scope) of the procedures that owned them. Hitherto he has advocated that own variables should be considered as declared concurrently with the procedures that own them, except that to avoid clashes of identifiers they are not recognised outside the procedures. Though less flexible than the current proposals—it does not solve the code-duplication problem—this already has the same clarifying effect on interpretation. There are no problems about changing bounds or multiple versions since these are redetermined only when control passes through the block-head in which the procedure is declared. The existence of multiple versions is under the control of the programmer; if the recursive call envelops the declaration then new versions are created, otherwise only one version is kept. But perhaps the most important reason for desiring some such interpretation is that it admits the well known technique

```
begin integer n; n := read;
  begin ⟨true program including arrays and own arrays [1:n]⟩ end
end
```

in which the bounds, though formally dynamic, are in fact merely a 'program constant'. Admittedly few implementations (if any) currently allow this, and the own arrays in Ord-Smith (1971) are declared [2:10] on the assumption that n will never exceed ten, but to standardise on an interpretation that excludes this technique, as advocated by de Morgan *et al.* (1974), would be deplorable.

Further remarks

It may be remarked that on the principle that one should try to write any program so that nothing is needlessly repeated (lest in changing it we fail to change all occurrences), we might wish to write something like

proc *P* = (**ref** [] **int** *A*) **proc bool**: (*perm* **freezing** (*A*, **loc** [1:**upb** *A*] **int** . . .
proc bool *newboys* = *P*(*boys*), *newgirls* = *P*(*girls*)

but this leads us into new troubles. One is how to perform the array initialisation; this is a wider problem than is covered by the questions discussed here; an immediate solution is to transfer the initialisation inside *perm* under the control of *first*—there being no problem about initialisation of simple variables. Another trouble is how to ensure an interpretation of *P* that does not lose the generated space when the call of *P* is completed, and this we shall not discuss.

The problem posed initially has given rise to several matters connected with the general question of the facilities a language should provide if it is to encourage good programming techniques. (Among these we include, on grounds of efficiency, avoidance of heap techniques as far as possible. Clearly, use of the heap would provide alternative solutions to some of the problems discussed here.) It seems possible that the full effects described here are not provided by any currently available language, although the author admits to ignorance of some that might provide them, and to inadequate acquaintance with either POP2 or ALGOL 68, the former of which seems to allow freezing of parameters but not local generators, and the latter to admit local generators, but not freezing in the manner required here.

ALGOL 68 seems to have got itself into this position through its almost hysterical determination to avoid ascribing any order to the parameters of a call. (Why else should the final report (1974) draw attention to the abolition of the 'gomma' of previous drafts in Section 0.3.11 as well as crow about the interpretation of collaterals in Section 0.2.4?) A simpler example than the permutation one will show what is involved in trying to freeze parameters anonymously in ALGOL 68, namely the provision of pseudo-random number generators. The procedure

proc *random* = (**ref int** *i*) **real**: (*i* := (*i* * *a* + *b*) **mod** *c*; *i*/*c*)

where *a*, *b* and *c* are suitably chosen integers, will provide all the code needed for the generation of pseudo-random numbers in the range 0-1. Independent sources are obtained from different parameters differently initialised. Thus *we* might write

proc *eureka*1 = *random* **freezing** (**loc int** := 0);
proc *eureka*2 = *random* **freezing** (**loc int** := 1)

But since it is not the same thing to write

proc *eureka*1 = **real**: (*random* (**loc int** := 0))

(because the parameter would be reinitialised on each call), we are forced in ALGOL 68 to program explicitly and separately, both the frozen structure and its use for calling, thus

mode closure = **struct** (**ref int** *param*, **proc** (**ref int**) **real** *routine*);
closure *S*1 = (**loc int** := 0, *random*), *S*2 = (**loc int** := 1, *random*);
proc *eureka*1 = **real**: ((*routine* **of** *S*1)(*param* **of** *S*1)),
*eureka*2 = **real**: ((*routine* **of** *S*2)(*param* **of** *S*2));

Nor do we achieve the security we aim at, for although this secures the immediate anonymity of the 'own' integers, they can still be accessed in the main program by the circumlocution '*param* **of** *S*'.

When it is considered that a stack is based on the philosophy that time sequence translates into space sequence and vice versa, it is not surprising that deliberate rejection of this philosophy exacts such a price.

Addendum on permutation algorithms

Apart from its linguistic interest, the investigation revealed a few points of interest in connection with permutation algorithms that seem to have escaped mention in previous published work. Structured programming calls for the elimination of **gotos** as far as possible, and the writing of programs in such a way that their correctness is as nearly self-evident as possible. Among the many trade-offs in programming there seems at times to be one between self-evident correctness and efficiency. It is not easy to put all the permutations of n objects in any order that ensures both an efficient transformation rule from one to the next and a simple proof that each permutation is generated once and once only. The following development of *permact* sacrifices efficiency for clarity.

proc *permact* = (**int** *n*, **ref** [] **amode** *A*, **proc** *act*) **void**:
(**for** *i* **to** *n* **do** *cyclic permutation of A*[1] *to A*[*n*];
if *n* = 2 **then** *act* **else** *permact* (*n* − 1, *A*, *act*) **fi od**)

(Proof: each cyclic permutation leaves a different element in the last place and is accompanied by all permutations of the other elements; the initial order is restored at the end of a call, ready for the next cyclic permutation in the case of recursive calls).

It was submitted to, and rejected by, the Editor of the Algorithms supplement, quite rightly at a time when interest was in efficiency and neither program proving nor structured programming had acquired the interest they command today. It does not contain any explicit own array, but this is misleading; examination of the use made by Ord-Smith's programs of this array shows that it contains information exactly equivalent to the nested set of values of i (one for each level of recursion) in this procedure at the moment when *act* is called.

As Page points out, though in somewhat different terminology, the $n!$ permutations on n objects can be put into one-to-one correspondence with the $n!$ distinct $(n-1)$-digit numbers in which the rth digit is of radix $r+1$. If the rth digit is allowed to run from 1 to $r+1$ in certain contexts instead of from 0 to r, then these digits are precisely the contents of the auxiliary array p of Algorithm ACM 115A as set out by Ord-Smith. It does not seem to have been pointed out that the sequence generated by this algorithm presents these numbers in consecutive ascending numerical order if they are interpreted as being in Grey code. The main problem in Grey code is to know whether a digit is ascending or descending; generation of a Grey number from its predecessor is much simplified if this information is arbitrarily incorporated into each digit as a $\pm$ sign (one reason for avoiding the use of zero as a member of the 'character-set'), and indeed this technique permits a version of Algorithm 115A that dispenses with the array d, which must surely be to its advantage in both storage and speed. Algorithm 115A uses knowledge of the route taken through conditional statements in incrementing the auxiliary array to determine the choice of transposition for stepping from one permutation to the next, but Page shows how a permutation can be obtained directly from the Grey representation (the first object going into the $A[n-1]$th place, the second into the $A[n-2]$th *vacant* place, etc.), a technique easily adapted for random permutations in Monte Carlo work. An efficient body for *permact* along these lines is

proc *permacta* = (**int** *n*, **ref** [] **amode** *A*, **proc** *act*) **void**:
begin c int *n could be replaced by an internal* **int** *n* = **upb** *A* **c**
int *h*, *k*, *q*; **bool** *bool*; [2:*n*] **int** *p*; **amode** *t*;

```
for h from 2 to n do p[h] := 0 od;
while p[2] ≠ −2 do h := n; k := 0; bool := true;
 while bool do q := p[h] := p[h] + 1;
  if q = 0 then k := k + 1; h := h − 1
  elif q = h then p[h] := −q; h := h − 1;
   if q = 2 then bool := false; q := 1 fi
  else bool := false fi od;
 q := abs q + k; k := q + 1;
 t := A[k]; A[k] := A[q]; A[q] := t;
 act od end
```

ALGOL 68 notation has been used here to obtain the benefit of 'pure **while**' statements; the equivalent ALGOL 60 is easy to write by insertion of labels, extra statement brackets, and so on. Efficiency in ALGOL 68 might be slightly improved by use of **ref amode** *ak* = *A*[*k*], *aq* = *A*[*q*].

References

ORD-SMITH, R. J. (1971). Generation of Permutation Sequences, Part 2, *The Computer Journal*, Vol. 14, pp. 136-140.

PAGE, E. S. (1971). Systematic Generation of Ordered Sequences using Recurrence Relations, *The Computer Journal*, Vol. 14, pp. 150-154.

LINDSEY, C. H. (1974). Modals, *ALGOL Bulletin*, No. 37, pp. 26-29.

BURSTALL, R. M., and POPPLESTONE, R. J. (1968). POP2 Reference Manual, p. 16 in *POP2 Papers*, Oliver and Boyd.

HIGMAN, B. (1967). *A Comparative Study of Programming Languages*, Section 11.3, English edition, Macdonald.

DE MORGAN, R. M., HILL, I. D., and WICHMANN, B. A. (1974). A Commentary on the Revised ALGOL 60 Report, *ALGOL Bulletin* No. 38, pp. 5-39. (see also this *Journal*, pp. 276-288).

Revised Report on the Algorithmic Language ALGOL 68 (1974) issued by University of Alberta as Supplement to ALGOL Bulletin No. 36.

LINDSEY, C. H. (1974). Partial Parameterisation, *ALGOL Bulletin*, No. 37, pp. 24-26.

Book reviews

Introduction to Mathematical Control Theory by S. Barnett, 1975; 264 pages. (*Clarendon Press, Oxford University Press*, £5·75).

This introduction to control theory has a clearly recognisable flavour distinguishing it from the many books in this area which have appeared over the last ten years. Most of the latter have been devised for relatively lengthy courses, permitting a degree of detailed discussion unsuitable for the shorter introductory courses typically given in the UK. Secondly the author's own research interests, particularly in linear system theory, although strictly disciplined as is proper in an introductory text, are apparent, mainly indirectly through the nature of discussions and proofs, but occasionally directly as in the section on 'controllability and polynomials'. Thirdly the origins of the book lie partly in a course given to undergraduate mathematics students.

Although designed to serve mathematicians, the book is concrete in nature, making minimal demands on mathematical sophistication. Chapters 2 and 3 review relevant theory of matrix algebra and linear vector differential equations. Chapter 4, on linear control systems, is the most important (and interesting) in the text. In less than 60 pages it discusses complete controllability (and the associated minimum energy control for steering a system from one state to another), algebraic equivalence, observability and observers, pole assignment, and realisation theory, the presentation being clear and consice. There are, inevitably, omitted topics. There is no discussion, for example, of the recent contributions of Wonham, Morse et al, on the algebra of invariant subspaces. Since this is itself a major topic, its omission is not surprising. However engineers may be disappointed at the lack of any mention of those topics, such as inverse systems and decoupling, which motivated this work.

The next chapter defines stability, discusses stability of linear systems (Routh, Hurwitz, Nyquist), gives an elementary exposition of Lyapunov theory and concludes with the Popov and circle criteria. Typically, the simpler theorems only are proven: thus the proofs of Barbushin's theorem and of the Popov and circle criteria are omitted.

The treatment of optimal control, in the last chapter, is largely formal, as the introduction of Lagrange multiplier functions is not justified. The standard necessary condition of optimality (Pontryagin's minimum principle) is illustrated by means of a few simple examples, and then used to determine the optimal (feedback) controller for a linear system with quadratic costs. Brockett's approach to this problem, using a sufficient condition of optimality is preferable, being rigorous, simpler, and well within the spirit of a book devoted largely to linear systems.

Despite these minor criticisms the book, unlike many others, clearly fulfils its author's aims of presenting a 'concise, readable account of some basic mathematical aspects of control' and as such will be widely welcomed.

D. Q. MAYNE (London)

Algol 60 and FORTRAN IV, by R. A. Vowels, 1975; 173 pages. (*John Wiley*, £5·00).

The author of this text believes that, by presenting two languages simultaneously, the reader will gain a greater understanding of programming languages generally. One wonders if this approach might not prove rather confusing to the novice, though the point is made that the book is not intended to be a complete treatment, The idea is that it should be used as a jumping-off point for deeper study of FORTRAN or ALGOL, if required, or even of some other language. Consequently, there are omissions: EQUIVALENCE, in FORTRAN, is not described at all, and the treatment of, for instance, the ALGOL concept of call by name is quite superficial.

A number of cases occur of the use of jargon and technical terms without prior definition—'keyword', 'memory location' and 'debug' were noticed, and would not be self-explanatory to the beginning programmer. One might, perhaps, class as jargon the crossing of the letter 'oh' in the program examples, and elsewhere. This habit, which appears to be becoming more popular, is very ANNØYING to this reviewer, who is NØT a key-punch ØPERATØR. Books should be readable, and the use of coding sheet conventions by their authors does not aid their readability.

The exercises at the end of many sections are interesting, but no solutions are provided. There are appendices on the comparative hardware representations and input/output conventions of System 4, 1900 series, Cyber 70 and System/360 (and 370) computer systems. The book has a useful bibliography.

The book is quite well produced, though a few typographical errors and other evidence of poor proofreading were noticed and the plastic protective laminate began to peel from the paper cover under the stress of brief-case treatment. It is not a 'do it yourself' text—tutor guidance is necessary, if only with the exercises—and it does seem rather expensive to recommend when there are books as good or better, available more cheaply, even though they may not attempt quite such a broad coverage as this one.

A. S. RADFORD (Leicester)