Problems of Storage Allocation

AKIHIRO NOZAKI*

It is well known that ALGOL 60 language provides a number of new programming techniques such as block nesting, dynamic declaration and recursive calling.

These techniques, however, comprise new difficulties for the compilers, especially for the storage allocating routine. In fact, no final solution has been proposed till now.

In this paper the characteristic features of the above three techniques are described. A new method is proposed for the allocation of the local and static variables.

1. *Terminology*

In this paper the variables are defined to include:

(1) simple variables;
(2) arrays;
(3) control words associated with arrays.

We call the variables *static* when their "sizes" are not changed in the computing stage. For instance, any simple variable is static. Non-static variables are called to be *dynamic*.

Strictly speaking, it is not at all easy to discriminate between static and dynamic. It may possibly be one practical resolution to take the all arrays as dynamic.

Own type variables are called to be *global*. Other variables are called to be *local*. Control words associated with local arrays are also local. Constant numbers can be regarded as static and global.

2. *Block Nesting*

Block nesting has evidently no influence on the global variables. On the other hand, it is the question how to pack the local variables with disjoint scopes into a limited memory area.

As for the local and static variables, the following method is well known (cf. [5], [6]).

Let l_* be a simple variable which belongs to the system program and indicates the address of the first usable storage cell in the memory area

This paper first appeared in Japanese in Joho Shori (the Journal of the Information Processing Society of Japan), Vol. 3, No. 6 (1962), pp. 312-318.

* College of General Education, University of Tokyo.

reserved for the local and static variables.

Every local and static variable A is allocated between l_* and l_*+n-1 where n denotes the size of A. After this allocation the value of l_* is increased by n.

On entering to a block B in the compiling stage, the compiler stores the value of l_* in a push-down storage. This value is restored on the exit from B so that the memory area allocated to the variables in B might be released.

No additional operations are executed in the computing stage.

For the local and dynamic variables, a separate area should be reserved. Allocation schemes will be discussed in the next section.

3. *Dynamic Allocation*

As for the local and dynamic variables, the allocotion can be executed in the analogous way using a simple variable L_* in the computing stage.

A serious problem, however, arises concerning to the restoring operation of L_*. In the computing time, there are abnormal exits by GO TO statements which may skip many END's instantaneously. Therefore the administration of GO TO statements has been proposed (cf. [4]), by which the destination of every GO TO statement is examined in the computing stage so as to restore the value of L_* properly.

Unfortunately, such the administration can not help consuming very much time. At this point, the following new method is preferable.

Predecessor method

Now let fix an arbitrary program Π and consider the blocks contained in it.

Lemma 1. For any blocks B and B':

$$B \frown B' \neq \phi \implies B \supset B' \quad \text{or} \quad B \subset B'.$$

Lemma 2. For every block B, a set of blocks

$$\mathfrak{A}(B)=\{D;\ D \supset B,\ D \neq B\}$$

forms a finite well-ordered set with respect to $\supset$.

$P(B) ::= \langle$the smallest block of $\mathfrak{A}(B)\rangle$

$P(B)$ is called the predecessor of B.

Note that $\mathfrak{A}(\Pi)=\phi$. ($P(\Pi)$ is undefined.)

Definition.

Now we can describe the principle of the Predecessor Method.

(1) To each block B, a simple variable L_B is attached.

L_B plays the same role as l_* in allocating the variables declared in B.

(2) On entering to a block B in the computing stage, the value of L_B is generally initialized as follows.

$$L_B:=L_{P(B)}.$$

If $B=\Pi$ then $L_B:=L_0$,

where L_0 is a certain constant.

If the block B happens to be an entire procedure, then $L_B:=L_{B'}$, where B' is the block which contains the procedure statement calling B at that moment.

The value of $L_{B'}$ will be delivered to B as one of the program parameters.

(3) Every local and dynamic variable V declared in B is allocated between L_B and L_B+n-1, where n denotes the total size of V. After this allocation, the value of L_B is increased by n.

No other operations are executed even on exits.

The validity of this method is easily ascertained by the following statements.

Lemma 3. Let B be a block which is now in execution. Then any variable V declared in a block D is accessible if and only if $D=B$ or $D\in\mathfrak{A}(B)$.

Theorem. All the accessible variables are allocated between L_0 and L_B-1, where B is the block now in execution.

The above area is completely occupied by only the accessible variables. These statements are easily proved using the concept of the scopes (cf. [1], 4. 3. 4).

Global and dynamic variables

The global and dynamic variables comprise a different kind of difficulties. For example, they require rearrangements of their elements (cf. [2]). Though such a rearrangement is very complicated, a precise description is shown in [3].

4. *Recursive Calling*

When a procedure P is called recursively, the variables in P should be declared doubly and have more than two storage areas. Therefore it is indispensable to discuss the *address reference scheme* as well as the storage allocation scheme.

The allocation scheme is also affected. As a matter of fact, recursive calling violates the concept of the scopes cooperating with the call-by-name parameters. Therefore the main theorem in §3 does not hold any more without relevant alternation.

Inevitably, discussion on these points will become delicate and com-

plicated. So we would like to note here only that the Predecessor Method is still available in principle. Further discussion will be found in a subsequent paper.

References

[1] NAUR, P. (editor), Report on the algorithmic language ALGOL 60. *Communications of the Association for Computing Machinery, 3*, 5 (1960), 299-314.

[2] SATTLEY, K., Allocation of Storage for Arrays in ALGOL 60. *Communications of the Association for Computing Machinery, 4*, 1 (1961), 60-65.

[3] INGERMAN, P. Z., Dynamic Allocation. *Ibid.*, 65-69.

[4] JENSEN, J., et al., A Storage Allocation Scheme for ALGOL 60. *Communications of the Association for Computing Machinery, 4*, 10 (1961), 441-445.

[5] INOUE, K., et al., ALGOL Syntax and Array Declarations (in Japanese). *Preprint for the National Congress of Information Prcessino Society of Japan*, 1961.

[6] NOZAKI, A., Automatic Programming, II (in Japanese). *Electrical Communication Laboratory Report*, No. 1802 (1962).