Alternative Scope Rules for Block-Structured Languages

R. T. HOUSE

School of Applied Science, Darling Downs Institute of Advanced Education, Post Office, Darling Heights, Toowoomba, Queensland, Australia 4350

A change to the scope rules made popular by Algol-60 is proposed which provides facilities similar to a Modula-2 module or an Ada package with no increased complexity over the original Algol scheme. Also,* own *variables become superfluous, as a facility very like them is an automatic byproduct of the modification. Implementation issues are discussed and a description is given of a run-time storage allocation system similar to an existing storage scheme for Algol-60.

Received December 1984

1. INTRODUCTION

The language Algol-60[1] first introduced the concept of scope based on block structuring which has since been widely recognised as an elegant basis for the visibility rules of a programming language. The essential idea is that identifiers are accessible in the block in which they are defined, or blocks within that block (unless 'shielded' by re-use of the identifier name in a more local block), but they are not accessible from outside the block in which they are declared. Most modern languages have some form of block structured scope rule, for example, Pascal,[2] Algol-68,[3] Ada[4] and many more. Nevertheless, many recent languages such as Ada and Modula-2[5,6] contain substantial language features violating this rule. Most of these changes centre on the inability of a pure block-structured language to set up self-contained facilities which access global objects without those objects becoming visible throughout the program, and thus being open to accidental or deliberate corruption.

Most attempts at solving this problem incorporate significant extra syntactic entities, adding substantially to the number of concepts fundamental to a language. Modula-2 has explicit import–export lists, while Ada has packages with a private and a public part. All these proposals suffer from considerable syntactic complexity, which seems out of keeping with the elegance of the original scope rules. Nevertheless they are still 'traditional' techniques with regard to their basic philosophy and implementation requirements. A much more radical alternative based on a dynamic view of types has been proposed by Harland.[7] He offers as 'first-class citizens' in a language, fully dynamic types which can be manipulated as data objects, and he shows how his proposal can provide program modularity facilities, as well as an equivalent of abstract data types. Although his scheme goes far beyond the scope of the scheme to be presented here (at the cost of a much more complex implementation) it deserves mention due to its similar philosophy of aiming for a clear and logical conceptual structure while possessing the greatest power possible.

The present proposal is much more modest than Harland's, capable of a much simpler and more efficient implementation by the traditional methods, and comparable in power with the 'package' style schemes mentioned above, but without the considerable syntactic and conceptual complexities of the extra language elements found in those schemes. Like the package schemes, it also relies on the fact that routines are not fully manipulable values. It retains static typing and proposes only a small change to the original Algol-60 block-structuring rules, but one which has surprising consequences, providing modularity facilities like those available with these more complicated mechanisms.

2. THE NEW SCOPE RULE

2.1. The basic concept

The Algol-60 scope rule can be stated as follows:
Accessibility from inner blocks: unlimited levels.
Accessibility from outer blocks: none.
The revised rule is:
Accessibility from inner blocks: unlimited levels.
Accessibility from outer blocks: zero or one level.

The rules regarding access from inner blocks are unchanged; also all variables declared in the current block are accessible as before. The difference is in allowing the programmer to specify at the start of each block whether its identifiers will be visible from the immediately surrounding block. We therefore envisage two possible block structures, the normal **begin..end** and the new block which we shall denote by **use..end**. Identifiers within a **begin..end** are only visible within that block; identifiers inside **use..end** are also accessible in the block within which the **use..end** occurs. This outside visibility never extends more than one level, however. Should a **use** block occur within another **use** block, the identifiers of the inner block are still only accessible inside the outer **use** block, and no more globally than that. Thus, for **use** to make an identifier visible outside, that identifier must be declared immediately within the **use** block, and not merely be visible within the **use** block. To improve documentation, we may allow identifiers to be qualified by the name of a label preceding a **use** block. So, if a **use** block labelled 'L:' declares a variable 'x', we may denote the variable by 'x' or by '$L.x$'. The latter alternative would often be preferable in the interests of readability.

Finally, we note that little purpose is served by making the main block of a procedure into a **use** block due to the fact that the entire text of a procedure constitutes a block within which the statement of the procedure resides. (The purpose of this extra block is to locate value parameters and restrict the scope of any label labelling the entire procedure statement part.) This extra block would prevent any **use** within it from having any effect outside the procedure declaration. This is probably just as well, as if it were otherwise access to the variables would be possible without the procedure necessarily having been called at all to set them up properly. This problem could probably be overcome but, as our purpose here is simplicity, it is fortuitous that, in Algol-60 at least, we need not worry about this difficulty.

2.2. Examples

The following program in a pseudo-Algol code illustrates use of these rules to provide something similar to Algol-60 'own' variables, which are accessible to a group of procedures, yet unknown to the rest of the program:

```
begin
  use
    comment Package for stack facilities;
    use
      integer stkptr;
      real array stk[1:1000];
    end;
    procedure push(r); value r; real r;
      begin
        if stkptr < 1000 then
          begin
            stkptr: = stkptr + 1;
            stk[stkptr]: = r;
          end
      end;
    real procedure pop;
      begin
        if stkptr > 0 then
          begin
            pop: = stk[stkptr];
            stkptr: = stkptr-1;
          end else pop: = 0.0
      end;
    stkptr: = 0;
  end of stack definition;
  ... Main prog ...
end;
```

A number of things should be noted regarding the above program. First, the stack created is initialised to the empty state by the statement '*stkptr*: = 0' immediately the outer **use** block is entered, since it is an inline block, and as such is not procedured. This is possible because the code of the outer **use** block can access the variables declared in the inner **use** block. The code of the main block (not shown) can access both procedures *push* and *pop*, since they are themselves declared in the outer **use** block, but cannot access the variables implementing the stack, in spite of the fact that they do continue to exist. This continued, but invisible, existence of the variables makes it perfectly safe for the procedures to access the variables from any part of the program where the procedures themselves are visible. This is why the visibility rule and the variable lifetime rule differ. We see that the procedures are using the variables as if they were an extended form of 'own' variable, except that the lifetime of these variables only extends to the nearest surrounding **begin**, rather than to the life of the whole program. These variables are properly initialised.

The following example sets up the equivalent of a private type with the help of a **type** declaration similar to those in commonly used languages:

```
begin
  use
    use
      type  (array stk[1:1000];
             integer stkptr) innertype;
    end;
    type stack = innertype;
    procedure push(s,r);value r;
      stack s;real r;
      begin
        comment Knowing the structure of a stack, we
        can manipulate it here;
        if s. stkptr < 1000 then
          begin
            s.stkptr: = s. stkptr + 1;
            s.stk[s.stkptr]: = r;
          end;
      end;
    real procedure pop(s); stack s;
      begin
        if s.stkptr > 0 then
          begin
            pop: = s.stk[s.stkptr];
            s.stkptr: = s.stkptr-1;
          end else pop: = 0.0
      end;
  end;
  stack one, two;
  comment We can declare stacks here, but cannot access
  their internal structure, since the type innerstack is not
  available here. We must access via push and pop which
  are visible from this point;
  ...
end;
```

The **type** declaration in the inner **use** defines the structure of an **innertype** object. That type is available within the outer **use** to permit definition of type **stack**; however, to actually access the components of a **stack**, it is necessary to make use of the fact that it is equivalent to an **innertype** object. This is only possible within the outer **use** so, although procedures *push* and *pop* can access the inner structure, this cannot be done in the main program. There, objects of this type can be declared, but use is restricted to the procedures which have access to the inner structure. One might argue that, in order to declare an object, a compiler must know its inner structure, at least so far as knowing its size, but there is nothing inconsistent in the view taken here: any access requiring explicit mention of a shielded object (in this case, the fields of **innertype**) is prohibited more than one block out, whereas the object itself (the **innertype** type) continues to exist as long as objects in the surrounding block exist, and we can rely on its existence (just as we rely on continued existence of the stack in the first example) provided we do not actually mention it explicitly in the program text.

2.3. Detailed implications

Use blocks give rise to some new situations in which identifiers must be resolved to one of several competing declarations. These are as follows.

(*a*) The same identifier is declared in a block and in an enclosed **use** block.

(b) The same identifier is declared in an outer block and in an enclosed **use** block.

(*c*) The same identifier is declared in two **use** blocks enclosed by the current block.

The resolution of these questions is, unfortunately, not as obvious as the resolution in normal block-structured languages of a local versus a global declaration. (Although the resolution is by no means *ad hoc*, as we shall see below.) The unmodified Algol-60 rules effectively mean that a more local declaration takes precedence over a more global one, and that the same identifier cannot be declared twice in the same block. Since the intention of the present proposal is to find a more powerful scheme than the original, but without added complexity, it is clear that all these problems should be resolved by appeal to a concise set of rules (just as the ambiguity in Algol was), otherwise the resolution will appear to be arbitrary, and one could then legitimately argue that the new scheme is conceptually more complex.

Fortunately a solution can be found by a shift in our understanding of the Algol rules above. We shall apply the Algol rules rewritten as follows. A declaration 'closer' (see below) to the application takes precedence over one further away, and two declarations of the same identifier are invalid if there is any point in the program text at which both declarations could possibly be applied and to which the two declarations have exactly equal closeness. The 'closeness' mentioned in this rule is closeness measured in terms of blocks exited/entered but not entered/exited in proceeding through the program text from the declaration to the point of application. (This, essentially a measure of difference in block level, is the same thing as the original rule when applied to ordinary blocks, since local variables are always closer than global ones, and declaration of one identifier twice in the same block is ambiguous within that block.)

Let us now address each of the three cases mentioned above. Case (*a*) is easily resolved, since in any given block its own variables are 'closer' than those of any other block; we may therefore say that the enclosed **use** block's variables should be hidden by those of the current block. Case (*b*) is harder since, if the outer block declaration is in the immediately enclosing block, one can legitimately ask whether one level out is closer than one level in, or vice versa. Since we are trying to obtain the simplest possible system, and since two (or more) levels out are definitely further away than one level in, clearly the one-level-out case should be grouped with the many-levels-out case rather than being a special case of its own. Another argument leading to the same conclusion is that included blocks actually form part of the current block, and so in that sense must be regarded as closer. One could also argue that a **use** block is half an ordinary block, since it takes two such blocks to erect an impervious barrier to access by more global blocks. Yet another reason for choosing this was is that it is more useful (and that, surely, is the unspoken reason behind most programming language designs).

The most serious problem occurs in case (*c*), where a block contains two **use** blocks, each declaring an identifier of the same name. In that outer block, both declared items would appear to have equal closeness. There are at least two ways to approach this problem. Application of the second part of our rewritten rules tells us that the declarations should be illegal. In a language like Algol-60 which has no separate compilation features, this would undoubtedly be the right choice, since the very purpose of having the variables in **use** blocks is negated if one of them (at least) cannot be used outside the block. Unfortunately, in a language with separate compilation facilities, the fact of life is that programmers will want to include modules which were not written by themselves. Therefore identifiers chosen by the writers of included **use** blocks may be, for quite legitimate reasons, beyond a programmer's control. These difficulties are sufficiently serious to warrant departure from the simple rules given above.

Languages which allow external compilations will of course provide a statement calling for inclusion of some external module. It is not uncommon for such systems to provide an identifier-renaming statement. An example is the ALIAS compiler directive in Hewlett Packard's Pascal/3000.[8] A common use for this sort of thing is when external modules might be programmed in an assembly language or some such system which permits extra characters such as dollar, apostrophe, etc. in identifiers. If such a thing already exists, it can be pressed into service to save the day here also, renaming any clashing identifiers so that no conflicts are presented to the compiler at all. Another possible solution is to require qualification by a label, as previously described, in any ambiguous case. Since this latter rule is only effective if a clash occurs, it does not complicate the basic rule in straightforward cases. If it should be objected that this is a complication simply in that the rule must exist, one could reply, first, that such complications to resolve ambiguities have a time-honoured history in programming languages (witness the Algol-60 rule requiring **begin..end** around an **if** after a **then**, and the Cobol rule requiring qualification of field names when ambiguity arises), and secondly, that the more complicated import–export and package schemes themselves require a similar complication to eliminate the same ambiguity (Ada, for example, having a system of qualification used to overcome ambiguities which can arise in a host of different ways).

One more possible objection to the simplicity of this scheme can be made, namely that having two types of block is more complex than having only one. To this we may reply that retaining the traditional block is little more than a piece of syntactic sugar, as the sequence '**begin..end**' is almost identical in effect to the sequence '**use use..end end**'. (The difference is earlier storage de-allocation in the former case; there is no operational difference. A good compiler could detect unnecessary storage retention in the latter case and arrange its earlier reclamation.) Viewed in this way, the discussion is seen to be about one set of scope rules versus another, and not about adding extra rules to an existing system.

The accessibility of variable names outside a block obviously raises the question of the lifetime of such variables. The system works correctly if variables declared in a **use** block exist as long as the variables in the surrounding block do. Note that this is a recursive rule. This may seem at odds with the strict limit on accessibility of such objects, but we shall see how this provides a very flexible and safe storage scheme. For sensible semantics, the outer block should be restricted to being a **begin..end** block as its data cannot exist longer than the execution of the program.

As a **use** block is a statement rather than a declaration, two issues arise concerning the sequence of execution of statements. First, a **use** block may be labelled, and therefore re-entered within a single execution of the surrounding block. Section 4 shows how a straightforward and efficient implementation can handle this. (There is one subtlety in this case, however. Entry to a block has always indicated redeclaration of entities within. If a block encloses a **use** block, then the items declared in that **use** block should be released no later than subsequent entry to the block which encloses it. This keeps the present proposal as close as possible to the usual system.) The second issue is that a **use** block might be skipped entirely in execution, but variables declared within it might be referenced further on. Two solutions are available. For an Algol-68-like language, the problem exists already for other entities, and the standard language mechanisms can be brought to bear on this case also. For an Algol-60-like language, space for simple variables can be allocated upon procedure (or program) entry, and arrays can be initialised to have empty bounds (i.e. [1:0]). This is the automatic effect of the implementation technique we shall present. It resembles in philosophy the automatic initialisation of **own** variables in the standard language, and so is by no means a forced or unnatural solution.

3. DISCUSSION

3.1 Comparison with own variables

We have seen an example which sets up a facility very much like an **own** variable. The precise differences bear elaborating. Since storage for **use** block variables is created in the same regime as that of the surrounding block, the point of allocation of storage may be transferred outwards if that blocks is itself a **use** block. This means that storage for nested **use** blocks is all grounded at the level of the most local normal block. As we have already seen, if that block is the outermost program block (as in the first example in Section 2.2.), we effectively have **own** variables, with the two advantages that we can include specific initialisations (rather than to the predetermined value zero as for **own** variables), and we can refer to them from any number of selected procedures, rather than from only one. If, on the other hand, we were to ground our **use** blocks in some other block, say the outermost block of a recursive procedure (if, say, our first example in 2.2 were the body of a certain procedure x), we would have a more interesting possibility. Within a single activation of that procedure, the **use** block variables are persistent. Suitably located internal procedures may access them just as if they were **own** variables. However, after recursive activation of the outer procedure, a new copy of the **use** variables will be created, and the inner procedures again treat them like **own** variables, except that they are not disturbing the values of the variables set up by the outer invocation of the recursive procedure. In that example, we see we would be supplied with a fresh, empty stack at each call of procedure x, be it a recursive call or not. This is a powerful improvement upon **own** variables, also including the ability to hide data.

There is one capability possessed by genuine **own** variables which is not available here, however. That is the possibility of declaring an **own** variable in a deeply nested procedure, and having it persist for the duration of the program. Which capability one would rather have is, of course, a matter of personal opinion, but the absence of **own** variables from recent languages would suggest that experience with these variables in Algol-60 has shown their limited advantages to be insufficient to justify the added complexity and the cost of their implementation; on the other hand, the possibilities in the present proposal for flexible data shielding on multiple levels within a structured and clean environment would seem much more likely to be extensively used.

3.2. 'Packages'

Our second example from Section 2.2 sketched out a 'package' scheme. The power inherent in this approach can be nicely enhanced in a host language possessing other powerful facilities. For example, there is a distinction made in Ada between private and limited private types. This distinction is within the capacity of these scope rules if embedded in a suitable language; a language like Algol-68, in which operators are declared, could require the programmer to declare all required operators in some manner. Then the assignment and comparison operators could be declared in suitable places, or not, as the programmer prefers. In fact, just the right combination of operators could be set up for the required application. The Ada **private/limited private** distinction, being based on an arbitrary set of operators, can be seen to be a more restricted concept.

3.3. Separate compilation

Examination of existing compilers for languages such as Algol-60, Algol-68 and Pascal reveals a nearly universal tendency to provide some form of separate compilation facility, yet the syntax and semantics of these is often quite convoluted. To provide separately compiled 'packages' under this proposal, we simply assign a meaning to the as-yet forbidden use of a **use** block at the outermost level of a program, as follows: all outer block names become accessible to the linker (which most compiler-based systems have) for resolution of undefined references in other object files; such a **use** block becomes a 'package'. Provision of a suitable 'external' declaration will now permit the main program (or another package) to access the objects in this package. This would probably work best if the compiler wrote a special information file on the objects in the outer **use** block. The 'external' declaration could then, with a single request, incorporate all the public information about the package. The effect would be as if the entire **use** block of the package were included in the program text at the location of the external declaration. Remarks regarding the implementation of external compilation will be made in Section 4.

3.4 Other scope rules

One of the more interesting variants of block structure is found in the Euclid system.[9] There 'block structure' is retained in so far as scopes may be nested indefinitely, but the Algol system of local declarations taking precedence over global ones is discarded. There are two kinds of scopes, closed and open, but in both kinds it is illegal to redefine an identifier from another scope which would otherwise be visible in this scope. Thus questions regarding ambiguous use of names cannot occur. In a closed scope (the more interesting kind) identifiers from the enclosing scope are only accessible if explicitly mentioned in an import list. The purpose of this rule is to enforce explicit mention in the text of regions where access to an object is possible. Nevertheless, it must have proved to be quite a burden on the programmer, since a special dispensation is given to constants in that they may be declared 'pervasive', meaning that they are automatically imported into all enclosed scopes. The designers have been prepared to accept the loss of orthogonality in order to give each type of object what they believe to be the ideal set of properties.

The clarity provided by the Euclid rule just mentioned is obtained in the present proposal by physical placement of blocks. The Algol scheme has been widely recognised as providing clear indications of the hierarchical structure of a program as well as excellent protection for local variables against global access, but lacking equal protection for globals against illegitimate local access. This defect is remedied here, as the new system increases the power of block placement strategies. Thus *all* data protection is indicated by the physical structure of the program, rather than indicating some by block placement and some by use of import lists. In that only one strategy is required of the reader to understand the program, this scheme would appear to present a better human interface than any import/export or module scheme mixed with a variant of block structure. With notational differences, most modern languages (such as Euclid, Ada and Modula-2) fall into that latter category, although the designers of Euclid have given us a simpler system than the others, due to their changes to the block-structure system, described earlier.

Before leaving this subject, we must examine an argument in favour of import lists, namely that the present proposal by its very nature forces regimes of access to be hierarchically nested. Some arrangements will therefore prove to be unrepresentable in any reasonable way. For example, given variables *a*, *b* and *c*, and procedures *X*, *Y* and *Z*, there is no straightforward way to arrange for *X* to access only the set of variables $\{a, b\}$, *Y* only $\{a, c\}$, and *Z* only $\{b, c\}$. At least some of the statements accessing particular variables would have to be procedured out and relocated. (Another method which we cannot, in all fairness, consider is to allow all three procedures access to all three variables and redeclare the unwanted name!) The example given presents no difficulty, of course, to import-list schemes.

The interesting question, then, is whether the structured discipline of the present proposal would actually exclude useful cases or whether, on the other hand, the excluded arrangements would turn out to represent poorly designed solutions to problems or to be easily expressed in an alternative, more amenable form. It is hard to see how a convincing answer to this question can be provided without actual experience in using a language designed along the lines proposed. After all, without experience in any block-structured language, one might argue that a system such as FORTRAN's named *common* blocks is superior to the Algol 60 scheme, since *common* permits arbitrary selections of variables to be shared in any desired manner between procedures. Experience, however, and the subsequent proliferation of block-structured languages provides adequate testimony to the usefulness in actual practice of this idea. In fact, its restrictions are seen by many to be the actual source of its strength, enforcing as they do a certain discipline on the programming process. Whether similar considerations apply to this latest development of block structure only time may tell.

4. Implementation

Compared with the difficulties of implementing a full 'package' language such as Ada, effort required is remarkably modest. To implement correct run-time behaviour, encountering a **use** block should result in all inner variables being created along with those of the outer block. Thus, for any sequence of nested or sequential **use** blocks within a normal block, the total storage required is evaluated at compile-time, and the original stack allocation to that outer block is made that much larger at block entry. For a two-pass compiler this is very easy, and for a one-pass compiler it is no harder than arranging correct addressing for forward **goto**s; the address of the original allocation instruction is remembered, and a fix inserted into the object code when the end of the block is encountered and all **use** block space is known. The only difficulty for code generation occurs with dynamic arrays. Dynamic extension to the data stack can be performed, provided care is taken that a **use** block is not re-entered during a single execution of the surrounding **begin** block; even this is not really insurmountable: the DEC System-10 Algol compiler,[10] for example, permits redefinition of the size of **own** arrays (a task of comparable difficulty) by storing the array elements in a heap.

In order to see that the above is really no more complex than a scheme designed purely for implementation of the normal variable scopes of Algol-60, we shall now modify an early and efficient scheme for Algol-60 storage administration to enable it to handle **use** blocks. The scheme to be modified is due to Gries *et al.*,[11] and is explained well in Gries.[12] To avoid lengthening this paper unduly, the reader is asked to refer to these references for full details, and only a brief outline of the unmodified scheme will be given here.

A fundamental insight of Gries' scheme is that blocks which are not procedures do not have to be allocated by a fully general mechanism, since the compiler knows more information about such blocks than for general procedure blocks (in particular, the point of activation, and the fact that they cannot be recursively activated without an intervening recursive procedure activation). Therefore Gries allocates space primarily on a procedure-by-procedure basis, only allocating arrays explicitly within blocks. Except for arrays, the space requirements of the entire procedure are evaluated statically, and upon procedure entry sufficient stack space is allocated for all blocks' fixed storage requirements. These requirements include, for each block within the procedure,

(1) a location *stacktop* which stores a pointer to the stack top while in the block,

(2) space for simple variables,

(3) locations for dope vectors of arrays, and

(4) locations for any temporary results needed in the block.

We shall call this the fixed space. Upon entry to a particular block, setting *stacktop* to the *stacktop* of the surrounding block is the only required action. (If this is the outermost block of the procedure, *stacktop* is set to indicate the end of the space allocated as just described.) Then, for each array in the current block, an array allocation routine is called which allocates space for the array on the end of the stack and updates *stacktop*. Since a separate *stacktop* exists for each block, absolutely nothing need to done for a block exit, whether normal or caused by a **goto**. (A **goto** out of the entire procedure will, of course, require execution of the usual procedure termination code.)

Gries points out that for blocks which are not nested, the necessary fixed spaces may be overlaid, thus reducing the required total storage. We now come to ask what changes are needed to implement the **use** blocks suggested in this paper. Clearly, our first change must be that space for **use** blocks cannot be overlaid with any other block's storage if that other block can be entered while the **use** block's data is still required, be that other block a **use** block or not. The solution to this problem is to alter the compiler's algorithm for working out the locations within the stack frame for allocations of storage space, and is not inherently difficult.

A trickier problem than the foregoing is ensuring that space allocated during a **use** block is not a de-allocated upon exit. Since block exit involves no instructions, we cannot proceed by trying to prevent an explicit de-allocation, but rather by taking some extra action to ensure that outer blocks' *stacktop*s are correctly set. The only way they can be incorrectly set, given the allocation mechanism described, is if arrays are declared. The solution is to ensure that, upon array declaration, all *stacktop* variables out to and including that for the most local non-**use** block are also updated. The number of these and their locations within the procedure data frame are known at compile time, so performing this is simple. Furthermore, since, usually, relatively few **use** blocks will be nested within each other, this will be quite fast. It also adds nothing to the difficulty of allocating normal Algol blocks. We have noted the fact that a **use** block at the outermost level of a procedure is meaningless, and this procedure-oriented allocation scheme fits in well will that observation, since we can always be sure that there will in fact be a most local non-**use** block.

Regarding interference with other parts of the implementation, we can be sure than none of the Algol complications such as thunks, recursion, procedures as parameters, etc. can have any effect on, or be affected by, these changes. This is thanks to the insulation provided by the usual procedure-calling mechanism, which this modification does not alter in any way. This can also be appreciated by noting that the first modification could be implemented anyway for normal blocks. (Gries in fact points out that his method of sharing storage is not necessary.) As for the second modification, it only means that array space used by a **use** block is not reclaimed at its exit, but it certainly does not cause any failure in the mechanism.

Should the above scheme be used, correct operation of programs including **use** blocks will result, unless one wishes to allow **use** blocks to be re-entered within a single activation of the procedure. Under the above scheme, re-entry would correctly allocate space for arrays with the sizes specified at the latest entry to the block, but the space which was previously occupied will not be reclaimed, as the array allocation mechanism simply allocates on top of the existing stack, which has not been reset.

In order to permit this, one might contemplate recording the *stacktop* prior to array allocation and reset it to its old value upon re-entry to the same **use** block. This would work except for the possibility of two or more non-nested **use** blocks within a block. The first **use** block might be entered, allocating stack space for arrays, then the second, also allocating space, and then the first might be re-entered. Resetting the stack would result in inadvertent de-allocation of the array space allocated to the arrays in the second **use** block (since it lies on top of the space allocated for the first one). Allocating the space in a heap will obviously get around all these problems, but we want to be sure that the scheme is not thereby made much more inefficient. Luckily we need not introduce any inefficiency in processing normal blocks, as their arrays may still quite happily be allocated on the stack, whether or not **use** blocks are present.

If we are to allocate space in a heap, provision must be made to deal with the problem of de-allocation. One of the advantages of the example implementation method given above is that block exit requires no instructions. To keep faith with Gries' original idea, we shall continue to insist that no operations be performed at block exit. This is the hardest situation, and success will provide the most convincing demonstration of the viability of the rules proposed. It is certainly essential that the full complexity of a generalised heap-management system be avoided, and heap management is much simpler if an explicit de-allocation instruction can be relied upon. This can be arranged as follows. Each **use** block array is allocated a pointer in the fixed space in the stack frame, as for ordinary array variables, but it points to regions allocated in a heap rather than on the stack top. If this pointer is non-zero, it signifies that it is currently in use. The actions to be taken up encountering each syntactic entity during execution are as follows.

(*a*) Procedure (and main program) entry: zero the pointers for all **use** arrays within the procedure.

(*b*) Block entry: check the pointers for all arrays in **use** blocks within the block at any level of nesting. If any are non-zero, call the de-allocation routine and zero the pointer.

(*c*) **Use** array declaration: If the pointer is zero, merely allocate the required space; if it is non-zero, the old space must be de-allocated and fresh space assigned (since the array bounds may have been changed) – if required, elements which existed in both the old and new arrays could be copied over, as is done by DEC System-10 Algol when **own** arrays are redefined.

This means that the array space might remain allocated beyond the point where it could logically be reclaimed if an outer normal block is exited, and also the procedure exit mechanism must de-allocate any such arrays before exit. The latter objection is minor, as the locations of all these pointers within the stack frame are known statically, and procedure exit code cannot be avoided anyway; the former objection can only be overcome by tolerating operations at block exit time (not that there is anything wrong with that, but we have deliberately posed ourselves the harder problem).

Another problem connected with a mixture of normal and **use** blocks concerns allocation of fixed space for normal non-nested blocks which, we have observed, may be done in an overlapping manner. It is obviously desirable to permit any reasonable overlap of **use** block space with space for other blocks to avoid wasted space in those cases where large numbers of local variables exist. The following procedure illustrates such a case:

```
procedure x;
  begin
  h: begin real a;
```

```
 i:    use real b;
          array e[...];
       end;
          ...
     end;
     ...

 j: begin real c;
 k:    use real d;
          array f[...];
       end
          ...
     end
     ...
     goto h;
     ...
  end;
```

In this case, space for blocks *h* and *i* is never needed at the same time as space for *j* and *k*. However, if blocks *i* and *k* are overlaid, as they declare arrays, the pointer to the allocated heap space in one block will be re-used for storage within the other, thus resulting in incorrect actions should the first block be re-entered. This can be circumvented by either de-allocating at block exit or not overlapping storage for any **use** blocks at all. In the above example, blocks *h* and *j* could use the same space, but *i* and *k* would be stored in different locations. If this latter alternative is adopted, storage allocation is slightly complicated for the compiler, but at run-time no feature of standard Algol has its efficiency compromised, while array allocation in **use** blocks introduces a modest overhead.

A slightly more subtle scheme from the point of view of the compiler is to remember exactly which locations in each **use** block's fixed space are critical and, although taking advantage of permitted overlap of ordinary variables, always skip over the critical storage when allocating space for other blocks. This gives the greatest compactness of the stack while adding nothing to the execution overheads. This is not as difficult as it sounds because the compiler can collect all the required information at compile-time, in spite of Algol's various dynamic features. This final subtlety must not be applied in such a way that external **use** blocks ('packages') have their fixed space allocated non-contiguously. With this precaution there will be no problem provided an information file is available to the compiler as mentioned in Section 3.3; the compiler, knowing the required stack usage, allocates space at the procedure level as described. The actual code for the external section is brought in from other object files by the linker. It is not, of course, actually merged in-line; a procedure call is inserted. The base address for the **use** block's fixed space must be passed across whenever execution passes into code in the external section.

For correct compile-time error detection, any variables or type definitions **use** d in a surrounding block should be implemented by a pointer into the (retained) symbol-table of the inner block. When a reference in a program results in more than one such pointer being followed, a syntax error results. For correct operation of the 'private' type shown in the previous section, we need only postulate that the size of a structure is directly associated with any type name referring to it, whereas detailed description is only accessible by following the trail back to the original declaration.

We see that both ease of compilation and run-time efficiency are good in an implementation of this proposal. Perhaps one reason for this is that the proposal forces all of the non-traditional data accesses to obey a single highly disciplined rule.

5. CONCLUSION

The rules presented here are essentially no more complex than the original block-structure scheme defined in Algol-60 when one allows for that language's **own** variable feature. In fact, one could argue that they are in fact simpler than the Algol-60 rules, because they are based on a single concept, namely block structure, rather than two concepts (block structure and **own** variables). Nevertheless, we have designed the equivalent of **own** variables with two extra advantages: they can be accessible from more than one procedure, and they can be 'grounded' on any level, not just the outer block, depending on how far out the surrounding nest of **use** blocks extends.

Import/export and package schemes are often suggested as a replacement for the unlimited accessibility of data in outer blocks. Under Algol scoping rules, the danger of an inner block damaging global data will always be present. This danger is even greater in languages like Pascal where block equals procedure, because in such languages all procedure data must be declared at the head. The rules proposed here, however, provide excellent security on a par with that of import/export or package schemes, but much simpler. The author's belief is that block structuring is elegant and simple and can be used safely without complicated additional language features.

Although Algol-60 has been mentioned frequently in the preceding discussion, the ideas presented here could be of relevance to any block-structured language which has not already been complicated by inclusion of more complex data visibility rules. Nevertheless, out of existing languages, these proposals seem to fit most naturally in Algol-60 or Algol-68. The point is, of course, that these proposed rules, even though just as simple as the original Algol-60 scheme, provide a basis for much more flexible and secure treatment of data, and demonstrate that it is not always necessary to include large extra 'features' in a language to obtain improved functionality.

Acknowledgement

I wish to thank Mr Mike McFarlane for his kind assistance in reviewing the first draft of this paper.

REFERENCES

1. J. W. Bachus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden and M. Woodger, Revised report on the algorithmic language Algol 60. *Communications of the ACM* **6** (1), 1–17 (1963).
2. ISO – International Organisation for Standardisation Programming Languages – *PASCAL, ISO/DIS 7185* (12 Aug. 1982).
3. A. van Wijngaarden, B. J. Mailoux, J. E. L. Peck and C. H. A. Koster, *Report on the Algorithmic Language ALGOL 68*. Mathematisch Centrum, Amsterdam.
4. United States Department of Defense, *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD 1815A (January 1983).
5. N. Wirth, *Modula-2*. Eidgenössiche Technische Hochschule Zürich, Institut für Informatik, Bericht 27 (December 1978).
6. N. Wirth. *Modula-2*. Eidgenössiche Technische Hochschule Zürich, Institut für Informatik, Bericht 36 (March 1980).
7. D. M. Harland, User-defined types in a polymorphic language. *The Computer Journal* **27** (1) (1984).
8. *Pascal/3000 Reference Manual*. Hewlett Packard Company, pp. 8.7–8.9.
9. B. W. Lampson, J. J. Horning, R. C. London, J. G. Mitchell and G. J. Popek. Report on the Programming Language Euclid. *SIGPLAN Notices* **12** (2) (1977).
10. *DECsystem*-10 *Algol Manual*. Digital Equipment Corporation.
11. D. Gries, M. Paul and H. R. Wiehle. Some techniques used in the ALCOR ILLINOIS 7090. *Communications of the ACM* **8** (8), 496–500 (1965).
12. D. Gries, *Compiler Construction for Digital Computers*, pp. 193–205. Wiley, New York (1971).