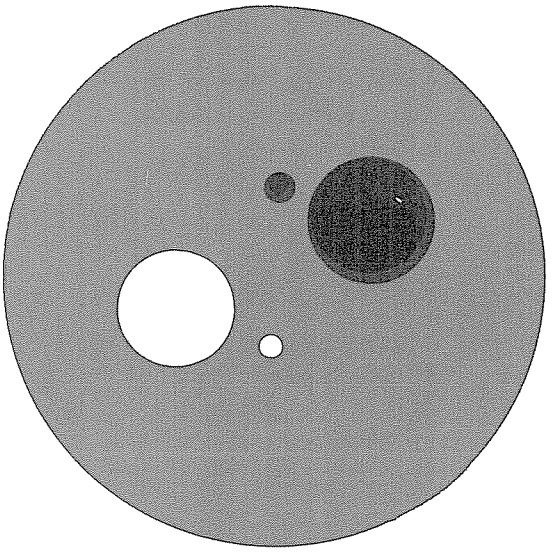
COMPUTER SCIENCES
DEPARTMENT

University of Wisconsin-
Madison



NAME PROTECTION IN BLOCK STRUCTURED PROGRAMMING LANGUAGES

by

Nai-Ting Hsu

&

Charles N. Fischer

Computer Sciences Technical Report #266

January 1976

NAME PROTECTION IN

BLOCK STRUCTURED PROGRAMMING LANGUAGES

by

Nai-Ting Hsu

&

Charles N. Fischer

Computer Sciences Department

University of Wisconsin

1210 W. Dayton Street

Madison, Wisconsin 53706

608-262-1079

ABSTRACT

Wulf and Shaw [1] have mentioned four deficiencies of Algol-like name scope rules (side effects, indiscriminant access, vulnerability, and no overlapping definition). In this paper a method is proposed to remedy these four deficiencies. This method can also be used to solve part of the confinement problem [8] in system protection at the programming language level. A method which provides for user defined access primitives is also proposed.

Introduction

Name access in block structured programming languages* is similar to resource access in computer systems in that both require protection. The necessity of resource protection in computer systems has been recognized as an essential role in implementing secure systems and much research has been done in this area. Compared with its dual problem, name protection in block structured programming languages has received little attention. Wulf and Shaw [1] have suggested that non-local variables should be abolished from all "higher level" programming languages and have illustrated several deficiencies of Algol-like name scope rules. These can be summarized as:

Side effects: Untold confusion can result when the consequences of executing a procedure cannot be determined at the site of the procedure call.

Indiscriminant Access: Restricting the use of a variable to a subset of the code at the block level where the variable is declared cannot be achieved.

Vulnerability: Within nested blocks, an intermediate block can prevent access from an inner block to a variable declared in an enclosing block.

*Since FORTRAN can be treated as a programming language with one block, it is also included in this discussion.

No overlapping definition: Variables should only be accessible to concerned program segments and inaccessible or invisible to others.

In this paper we intend to compare and to contrast these two protection probelms; to study existing protection structures in block structured programming languages; and then to propose a uniform solution to remedy these definiciencies. For convenience, "programming languages" will be used instead of "block structured programming languages" in the following discussion.

Comparison:

In most computer systems, a process can only access limited resources with specific access primitives. The collection of these limited resources and the corresponding access primitives forms the execution environment of the process. Ordinarily, the execution environments in a computer system form a tree-structured hierarchy [4]. The process in a parent node has the responsibility to specify the execution environments of its sons and to allocate resources to them. A process in a son cannot access those resources in its parent which are not allocated to it. Ideally, a son should also have the ability to selectively prevent its parent from accessing some resources which have been allocated exclusively to it. From an protection point of view this feature is desirable, since we sometimes wish to guarantee privacy to a son. From a reliability

standpoint, this can at times prohibit the propagation of faults from parent nodes. Unfortuantely, this feature is not provided in most current systems. In [4] this problem has been studied thoroughly and sufficient conditions to be satisfied by a hardware architecture have been proposed.

In contrast, most programming languages use Algol-like name scope rules. These rules are that names declared in a program block cannot be accessed outside the block and can be accessed arbitrarily in all inner blocks. This makes global names entirely unprotected. As indicated by Krutar [2], there are syntactic and semantic structures in current programming languages which can be used to achieve name protection. A few such structures may be found in [2][3][5][7][9]. Most of them use an all or nothing philosophy. That is either a name is known and can be accessed arbitrarily or it is unknown and no access of any kind is allowed. Such solutions are unsatisfactory. What we need is a solution similar to its dual problem (i.e., a way to explicitly control name accesses).

In order to consider this problem more thoroughly, let us discuss a basic protection scheme and the meaning of type in programming languages.

Protection and Type:

When an object is accessed, its name and the access primitive* to be applied to it should be specified either explicitly or implicitly (Figure 1). Protection of an

(name, access primitive) → Object

Figure 1

object can be achieved by either making the name of the object unknown or restricting the access primitives* which can be applied to it. Clearly making an object's name unknown is a degenerate case of restricting access primitives (i.e., all access to the object is prohibited). The ability to access an object with specified primitives is termed the access right.

In discussing name protection in programming languages, two kinds of names must be considered - variable names and procedure (or function) names. A variable name represents an information container** and the information (or data) contained in it. The type of a variable name specifies the structure of the named container and the information, the access primitives and the meaning of the information. The access primitives are those primitives which perform operations on information.

For example let A be declared to be an integer. This specifies a container which contains an integer datum. Its access primitives

---

* Access primitives specify permissible ways of using the object referenced.

* In those computers with supervisor/user mode, priviledged instructions cannot be executed in user mode. This is an example of restricting an access primitive.

** The relation between a container and a memory location depends on the memory allocation scheme used in a particular implementation.

are integer arithematic operations, testing operations, etc. All of them can be considered to access a datum either by nondestructively reading it and/or by overwriting it. This separates access primitives into 3 different classes. Each class can be represented by either the access primitive 'Read' (abbreviated 'R'), and/or the access primitive 'Write' (abbreviated 'W') respectively. All access primitives of variable names with types as specified in Pascal [6] can be classified in this way.

In those languages with abstract data types (e.g. clusters in CLU [5], monitors in concurrent Pascal [7]), an abstract data type is defined with the corresponding access primitives. For example, let S be defined as a stack, through cluster definition in CLU, with 'pop' and 'push' as the only two primitives. By definition, these two are the only legal access primitives of S . From the protection point of view a variable with an abstract data type will be treated the same as a variable with a Pascal-like type.

As well as such basic primitives, as +, -, etc., most languages also provide the facility to define new operators through procedure definition. The arguments of a procedure explicitly specify the data to be manipulated. Since the purpose of a procedure is to be executed, the access primitive associated with it is termed, 'Execute' (abbreviated 'E'). It should be clear that a procedure name can be treated as a name with a user defined type and an access primitive 'E'.

Name Protection in Current Programming Languages

In this section a number of the syntactic and semantic structures of name protection in existing programming languages are considered. As mentioned above, they are of the 'all or nothing' style.

1. The capabilities of modulization and of compiling different modules separatly make the local names in each module unknown outside the module. Each module can selectively make local names known to others through entry point declaration.

2. Argument transmital in procedure calls provides another kind of protection. Call by value is a form of 'R' only. Call by result is a form of 'W' only. Call by reference allows 'R' and 'W'. Call by name allows 'R', 'W', and also 'E' (really we are allowed to execute "thunks" [10] which may perform 'R' and 'W' indirectly). Procedure arguments and label arguments are other types of names which can be thought of as execute only.

3. Labeled common blocks in FORTRAN IV provide the ability to collect variables into groups (i.e., data areas). Each particular group of variables can be only shared among designated modules.

4. The concept of a monitor and a cluster provides the abilities to build structured data types with access primitives based upon existing data types and primitives. Access rights to

a structure and its substructures are decoupled. Substructures can be only accessed inside the structure and the structure can be only accessed through defined access primitives.

Syntactic and Semantic Structures for Name Protection:

As mentioned above, Algol-like name scope rules are inadequate. Something stronger is needed. One possibility is that the scope of a name be extended into an inner block only through explicit declaration. The declaration should specify not only the name but also the corresponding access primitives.

The name scope of a block represents all names which can be accessed in that block. This includes local names (which are declared in the block) and global names which are brought in from the inner most containing block through explicit declaration. In a block, local names can be referenced without restriction and global names can only be accessed with those primitives specified in the declaration. The access rights to a global name can only be a subset of the access rights of the name in the outer block (i.e., the access rights of a name cannot be increased by transmitting it to inner blocks).

The syntactic and semantic structures used for this purpose are explained informally in the following example (Figure2). Pascal-like variable declarations are used. Although only a few types are used in the example, all other types (including user defined types and access primitives) can be easily included. 'R' and 'W'

are used as access primitives for most types of variables. As mentioned above, 'R' and 'W' represent classes of access primitives. It is definitely possible that individual access primitives (such as +, -, etc.) might be explicitly transmitted (either individually or by classes).

In this example, the external declaration is used to bring variable names into a block from the name space of the innermost contianing block. The external declaration at line 14 specifies that variables a and b which are declared in block B1 (the innermost containing block), can be also referenced in block B2. As well as variable names, the corresponding type and access rights are also specified explicitly in the external declaration. In this declaration (line 14) variables a and b, with type integer, are write-only.

The local declaration is used to specify explicitly those names in current block which cannot be transmitted into an inner block. Line 19 specifies that variables h and a are 'local' in block B2. If line 21 were changed to

21' external b, h : integer (R);

a compiler would be expected to signal a semantic error. Another error would also be found in Line 21'. Since in B2 b can be only accessed with 'W' (line 14), it would not be correct to include the access right 'R' with b in B3.

In some applications, we would wish to have variables which

```
1  B1 : begin
2         var  a,b : integer;
3         var  c   : record   d : real;
4                             e : integer
5                    end
6         var  add : integer procedure with
7                    parameter  p1 : integer (R);
8                               p2 : integer (R,W);
9                    external   b  : integer (R);
10                   {procedure body which can only reference
                      variables p1, p2, b, and local variables}
11                   end       {of procedure add}
12        share c;
13     B2 : begin
14           external a,b  : integer (W);
15           external  c   : record   d : real (W);
16                                    e : integer ( )
17                         end
18           var       h  : integer;
19           local     h, a;
20           B3 : begin
21                  external b : integer (W);
22                  {statements here can only reference variable b}
23                  end {of B3}
24                {statements here can only reference variables a,
                    b, c.d, and h}
25                end {of B2}
26     B4 : begin
27           external  add : integer procedure with
28                           parameter p1 : integer (R);
29                                     p2 : integer (R,W);
30                           external  b  : integer (R)
31                           end
32           external  c   : record    d  : real (B);
33                                     e  : integer (R,W)
34                           end
35           external  b   : integer (R);
36           share     b   ;
37           B5 : begin
38                  external   b   : integer (R);
39                  {statements here can only reference variable b}
40                  end {of B5}
41                {statements here can only reference variable c,
                    and procedue    add}
42                end {of B4}
43        {statements here can only reference variable a,b,
           and procedure     add}
44        end
```

Figure 2

can be shared among blocks with same lexical level and which can only referenced within them. The share declaration is used for this purpose. Line 12 declares variable c (with record type) to be a shared variable. This means that c can be brought into the name spaces of the inner blocks (e.g., B2, B4) through external declarations (e.g. lines 15 & 16, lines 32 & 33), but it is not considered to be in the name space of B1. Lines 15 & 16 and lines 32 & 33 also show the partitioning of access rights to different fields of a record ('( )' in line 16 means empty access rights). Line 36 shows another usage of the share declaration. Variable b is brought into block B4 through external declaration (line 35). The share declaration in line 36 makes it inavailable in block B4, but it can be brought into block B5 through an external declaration (line 38).

Since a procedure can be treated as a type, with special structure, its declaration can be treated the same as other variables*. Line 6 declares an integer procedure named add. Lines 7 & 8 specify the parameters with corresponding types and access rights. Line 27 brings the procedure add from the name space of B1 into the name space of B4. Since the only access right to a procedure is 'Execute', it need not be explicitly stated.

---

*Actually procedures are more like named constants. For simplicity, we treat them as variable names. Naturally named constants can be thought of as variables with access rights limited at the point of declaration.

Scope Rules for System Defined Names:

As well as variable and procedure names, there are other kinds of names (most of them are predefined), used in a programming language. These include operators (e.g., +, -, etc.), I/O routines and I/O file names (I/O unit numbers in FORTRAN), system routines, type names, etc. From a conceptual point of view, all of these can be thought to be defined by the system and/or the user in a pseudo block which contains the user's program module. Their names are extended through the whole program. In practice, some of these names need more restrictions. For types predefined by the user, we suggest that a name scope be extended into an inner block only if a global name with this type is brought in through external declaration.

The confinement problem [8] in system protection is due to the inability to control one program module's I/O from another program module* (i.e., a special case of side-effect problems). Part of this problem can be solved, if the latter can specify explicitly those files which the former can write on and the system provides facilities to enforce this.

For example, assume procedure Account to be a service routine used for accounting purposes. Since accounting data are considered private, leakage of data through shared use of this service routine should be prevented. This can be achieved by the following method.

*We assume that a new copy of a program is used for each execution and that the only way to transfer data from one run to another run is through files.

1. In translation, the language translator will collect all output file names* used by Account.

2. In any program, from which Account will be called, these output file names must be specified explicitly through external declarations. For example, we might have

```
external Account: procedure with
                  parameters <parameter list>;
                  external    file names: file (W);
```

3. At linkage time, the system should check the information as specified in 1 and 2. If there are names which do not match, the system will report the error and inhibit the execution.

User Defined Access Primitives

The concept of a monitor and a cluster provides the ability to build structured data types with access primitives. A data structure defined in a monitor or a cluster is a local data structure which can be only referenced indirectly through defined access primitives. In some situations, we may need the ability to access directly part (or all) of a data structure defined in a monitor or a cluster in a particular name space. Let us call this new structure a semi-monitor or a semi-cluster. This concept and a way to use it can be illustrated by the following example (Figure 3).

*These also include all output file names used by any subroutine called by Account. Input file names are not required to be specified (if they cannot be changed during the run), since the program can only read data from these files. Also in some situations, they are private to the service routine and it is not deisrable for the user to even know of their existence.

```
1  B1:begin
2       var a,b : integer;
3       var c : semi-cluster with inc-by-one
4               external  a : integer (R,W);
5               inc-by-one : operation
6                            a := a + 1 ;
7                            end {of operation}
8               end {of semi-cluster}
9    B2:begin
10         external  c : semi-cluster (inc-by-one) ;
11         {statements with a name space contain c only}
12         end
13      {statements with a name space containing a,b, and c}
14      end {of B1}
```

Figure 3

In this example, line 3 defines a semi-cluster* c with the access primitive inc-by-one.** Variable a is declared to be an external variable in c (line 4). This means that a still can be accessed in Block 1. Line 10 brings c with access primitive inc-by-one into block B2 (i.e., in B2 we can only add 1 to a - nothing else).

One application of this concept is that a calling program can have more control on the usage of parameters in a called program by explicitly specifying the allowed access primitives without loosing itself any privilege to the parameters.

Conclusion:

From the above discussion, it is clear that the four deficiencies mentioned earlier (side effects, indiscriminant access, vulnerability, and no overlapping definition) of Algol-like name scope rules can be remedied. Explicitly specifying the name scope in a programming language is a natural way to achieve this purpose. Part of this motivation is due to the study of protection properties in computer systems. We expect such interaction between programming languages and computer systems to help us understand both more clearly and to lead to the creation of more secure and understandable systems.

---

*Definitly a semi-monitor can be also defined in this way.

**"inc-by-one" can be thought as a restricted 'Write' access primitive.

Naturally, our proposals are by no means the final solution to the problems of Algol-like name scoping. Rather, they are but a single step toward the goal of more secure, systematic and understandable programming languages.

References:

1. Wulf, W., and Shaw, Mary, Global Variable Considered Harmful, SIGPLAN Notices, Vol. 8, No. 2, February 1973, pp. 28-34.

2. Krutar, R. A., Restricted Global Variables in Algol 60, SIGPLAN Notices, Vol. 8, No. 12, December 1973, pp. 15-17.

3. George, James E., and Gary R. Sager, Variables - Bindings and Protection, SIGPLAN Notices, Vol. 8, No. 2, December 1973, pp. 18-29.

4. Hsu, Nai-Ting, Protection Properties and Hardware Architectures for Recursive Virtual Machines, Ph.D. dissertation, University of Wisconsin at Madison, in preparation.

5. Liskov, B., and Zilles, S., Programming with Abstract Data Types, SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp. 50-59.

6. Wirth, N., The Programming Language Pascal, Acta Informatica, Vol. 1, No. 1, 1971, pp. 35-63.

7. P. Brinch Hansen, The programming language Concurrent Pascal, IEEE Transactions on Software Engineering 1, 2, June 1975.

8. Lampson, Bulter W., A note on the Confinement Problem, CACM Vol. 16, No. 10, October 1973, pp. 613-614.

9. Chow, Tsun S., and Blackwell, Paul, Scope Rules--Deficiencies and Remedies, Proceedings of the 4th Texas Conference on Computing Systems, University of Texas, Austin, Texas, November 17-18, 1975, pp. 1A-5.

10. Gries, David, Compiler Construction for Digital Computers, John Wiley & Sons, Inc., 1971.