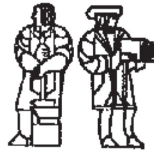
LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A Language Extension for Expressing Constraints on Data Access

Computation Structures Group Memo 146-1
Revised June 1977

Anita K. Jones
(Department of Computer Science, Carnegie-Mellon University)

Barbara H. Liskov

This research was supported by the Natioal Science Foundation under grants DCR74-21892 and DCR74-04187.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A LANGUAGE EXTENSION FOR EXPRESSING CONSTRAINTS ON DATA ACCESS

Anita K. Jones
Computer Science Department
Carnegie-Mellon University
Pittsburg, Pennsylvania

Barbara H. Liskov
Department of Electrical Engineering
and Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

ABSTRACT

Controlled sharing of information is needed and desirable for many applications and is supported in operating systems by access-control mechanisms. This paper shows how to extend programming languages to provide controlled sharing. The extension permits expression of access constraints on shared data. Access constraints can apply both to simple objects, and to objects that are components of larger objects, such as bank account records in a bank's data base. The constraints are stated declaratively, and can be enforced by static checking similar to type checking. The approach can be used to extend any strongly typed language, but is particularly suitable for extending languages that support the notion of abstract data types.

Keywords and Phrases: programming languages, access control, data types, abstract data types, type checking, capabilities

CR Categories: 4.20, 4.35

This research was supported by the National Science Foundation under grants DCR74-21892 and DCR74-04187. This paper reports a substantial extension of work previously published by the authors [7]; Section 2 of this paper describes the earlier work.

1. Introduction

One important aspect of a programming language is the way its scope rules control the sharing of data among individual program units (procedures, blocks, modules): two program units can share a datum only if both can name it. Ordinarily, however, languages provide access to data on an all-or-nothing basis. For example, if a module has access to a data structure, then every component of the structure may be read and modified. Yet experience in building large applications and applications involving sensitive data has indicated that finer control is desirable. The ability to limit a module to read-only access to a datum provides only a part of the needed control. For example, if some of the information in a data structure is sensitive, then the ability to limit a module to reading only a subset of the components is desired.

In this paper we describe a programming language extension that permits controlled sharing of data. We define a semantics and experimental syntax for this extension. Our approach borrows heavily from work in operating systems, where access-control mechanisms have long been one of the tools useful for realizing controlled sharing of data. In particular, our mechanism is modeled after the capability protection mechanisms provided by some operating systems [9, 16].

In capability-based systems, all data are assumed to be contained in *objects*, such as libraries, stacks, files or the central data base for an inventory control system or a banking system. For each object there exists a set of accesses, which are the only means for altering the object or extracting information from it. For some objects, accesses are the familiar read, write, and, possibly, execute access; for others, the accesses themselves are user-defined, tailored to the abstract notions appropriate to a particular application. Controlled sharing of objects is achieved by limiting a program unit to a subset of the accesses for an object. For example, a programmer defining a file system may choose to distinguish between write access and append access. In

contrast to a write access, an append access can be used to augment a file, but not to alter existing content. Thus, a program granted append access, but not write access, will be unable to overwrite existing data in a file.

There is a similarity between the idea of an object and associated accesses and the programming language concept of a data type. In strongly-typed languages, each type defines a set of primitive operations that are the only direct means for manipulating objects of that type. For example, the various arithmetic operations are primitive for integers, while primitive operations for arrays include operations to read and update array elements. The operations of a type correspond closely (though not identically, as we shall show) to accesses, and access control is a convenient way to control the use of the operations.

Our technique for access control is based on an augmented concept of a data type, and could be used to extend most (strongly-typed) programming languages. For example, if a language containing data structures (like PASCAL [15] or ALGOL 68 [14]) were extended, then control over reading and writing of each component of the structure would be possible. However, the class of languages including SIMULA 67 [2], CLU [10], and Alphard [17] is particularly well suited to our extension. Such languages permit the introduction of user-defined, abstract data types, consisting (just like the built-in types) of a set of primitive operations that provide the only direct means for creating and manipulating objects of the type. By introducing a new type, a user can tailor a set of accesses to an application. We will show how to extend this class of languages; part of our extension will involve defining rules governing new type definitions.

We will require that access-control restrictions be stated in a declarative fashion analogous to type declarations for variables. In languages requiring type declarations for variables, a static examination of the text of a program is sufficient to determine whether the program is type-correct. Similarly, our addition enables static analysis to determine whether a program is

access-correct: A program is *access-correct* if the accesses actually used in it do not exceed those stated in its declarations.

Thus, the access-control extension permits access-control restrictions to be stated explicitly in programs, and enforced by the compiler. The main benefit of the approach is enhanced software reliability: programs can be guaranteed to be well-behaved with respect to the constraints governing sharing of data. Access-control errors can be caught early, and a programmer can be confident that his program will not fail due to an access-control violation. In addition, the access-control restrictions in a program convey information about assumptions made by its programmer; this information can be relied on by someone reading the program to obtain a better understanding of its purpose.

An additional benefit of the extension is that a programmer will be able to express fully in the language how he intends to make use of the protection facilities of an operating system. At present, such access-control information is expressed separately from the program in some sort of job-control language; this separation increases the difficulty of writing programs for such systems. Also the language permits more precise specification of access requirements on a program unit by program unit basis, not on a user-job or job-step basis.

In the next section, we define the semantics and syntax of the access-control mechanism, and describe how access control is achieved for simple, unstructured data objects. Section 3 extends the access-control mechanism to structured data objects such as records and arrays. In Section 4 we relate the mechanism to program construction and storage of long-lived objects. We conclude in Section 5 with a discussion of what we have accomplished.

2. Access Control for Simple Objects

To accommodate access control, we will augment the notion of a data type by adding one more component: In addition to objects and operations, a type also specifies a set of rights. A *right* is a name that represents one of the legal accesses to the objects of the type; often a right corresponds to the use of one of the type's operations. The basic idea behind rights is: to legally apply one of the type's operations, a user must hold appropriate rights to the objects passed to that operation as parameters.

An example is given in Figure 1 for the type *AssociativeMemory*. This type includes an operation *MakeMem* to create an empty *AssociativeMemory* object of a particular size, an operation *Insert* to add a name-value pair to an *AssociativeMemory*, an operation *Change* to alter the value associated with a given name, an operation *GetVal* to fetch the value associated with a given name, and an operation *Delete* to remove a name-value pair. In order for *Insert*, *Change*, *GetVal*, or *Delete* to be invoked, the invoker must present a right to apply the operation to the *AssociativeMemory* object passed in as a parameter; in this example, the name of the required right is the same as the name of the operation. The *MakeMem* operation returns all these rights for the *AssociativeMemory* object it creates. The *AssociativeMemory* operations also use objects of type *integer*; for simplicity we have chosen to omit information about required rights for *integer* objects.

In general, we can expect some rights to correspond to the use of a single operation, some to a group of operations (e.g., a single right "arithmetic" might control the use of all *integer* arithmetic operations), and some to a single parameter of an operation taking more than one object of the type. As an example of the latter case, consider the *file* type discussed in the introduction, and suppose a *merge* operation for files merged two files, f and g, by changing f to contain the result of the merge, and leaving g unaltered. It might be useful to require different rights for the

Figure 1. The AssociativeMemory type.

Type: AssociativeMemory
Rights: "Insert", "Change", "GetVal", "Delete"
Operations:

MakeMem
 input: integer; (desired AssociativeMemory size)
 output: AssociativeMemory; "Insert","Change","GetVal", "Delete" rights are given
Insert
 input: AssociativeMemory; "Insert" right required
 integer; (the name)
 integer; (the value)
 effect: (Insert modifies its AssociativeMemory parameter)
Change
 input: AssociativeMemory; "Change" right required
 integer; (the name)
 integer; (the new value
 effect: (Change modifies its AssociativeMemory parameter)
GetVal
 input: AssociativeMemory; "GetVal" right required
 integer; (the name)
 output: integer; (the value)
Delete
 input: AssociativeMemory; "Delete" right required
 integer; (the name)
 effect: (Delete modifies its AssociativeMemory parameter)

---

two file parameters, e.g., both the "merge" and "mergeto" rights are needed for *f*, but only the "merge" right for *g*.

2.1 Notation and Rules

Our approach to access control is based on a semantic model in which *objects* are shared among *variables*. Each object has a *type*, which determines the legal accesses to the object. A declaration must be given for each variable stating the type of object that variable may refer to, and the rights that are available for that object when it is used via the variable. These two pieces

of information are captured in the notion of a *qualified type*. A qualified type Q is written

$T\{r_1,...,r_n\}$

where T is the name of some type, and $\{r_1,...,r_n\}$ is a subset of the rights of T. We refer to the two parts of a qualified type as the base type and the rights; e.g., $base(Q) = T$ and $rights(Q) = \{r_1,...,r_n\}$. The following are some of the qualified types derived from the base type *AssociativeMemory*

AssociativeMemory {GetVal}
AssociativeMemory {Insert, Change}
AssociativeMemory {Insert, Change, GetVal, Delete}

The final example specifies all the *AssociativeMemory* rights; a special notation

T{all}

may be used instead of listing all the rights.

Qualified types are used in variable declarations and in formal parameter specifications in procedure headings. An example of a variable declaration is:

v: AssociativeMemory {Insert, Change}

The meaning of this declaration is: v is a variable that can be used to refer to *AssociativeMemory* objects, but only the "Insert" and "Change" rights may be exercised in conjunction with v.

We view a variable as a pair

{object-id, qualified type}

The object-id is a name that is interpreted by the underlying addressing mechanism to select a unique object. When a variable is created, its qualified type is defined once and for all and can never be altered. However, the object named by a variable (via the object-id) can change by application of the *binding* operation. Binding causes a variable to refer to an object by storing

that object's id in the variable. Sharing of objects takes place when two variables contain the same object-id. (Our variables are like typed pointer variables, and binding is pointer assignment.)

A variable contains a capability in the operating system sense [3, 4, 8]. The capability provides the basis for restricting the kinds of manipulation that can be performed on the object specified by the object-id. Intuitively, the restrictions on how an object can be used are expressed along the path to the object (the path through the variable). Thus, using one path rather than another to name an object may change the way the object can be manipulated. For example, suppose

a: AssociativeMemory {GetVal, Insert}
b: AssociativeMemory {GetVal}

both name the same object. Using *b* it is impossible to modify this object, since only the *GetVal* operation can be used; using *a*, the object may be modified by application of the *Insert* operation.

The effect of binding is creation of a new access path for the object, and we must ensure that no new access rights are obtained from this new path. For example, suppose that x and y are variables, and that x is to be bound to the object currently bound to y. This binding should be allowed only if the qualified types of x and y both arise from the same base type, and if the rights obtainable by referring to the object via variable x do not exceed the rights obtainable by referring to the object via y.

We can formalize the binding rule as follows. First, we define what it means for one qualified type to be greater than or equal to another. If $Q1$ and $Q2$ are qualified types, then $Q1$ is *greater than or equal to Q2*, written

$$Q1 \geq Q2$$

if $base(Q1) = base(Q2)$ and $rights(Q1) \geq rights(Q2)$. Now the rule of binding can be defined:

$v \leftarrow e$

where v is a variable and e is an expression and

T_v = qualified type of variable v
T_e = qualified type of expression e

is legal provided that

$$T_e \geq T_v$$

Thus a binding is legal only if the new access path provides at most a subset of the rights obtainable via the original access path. Note that this rule ensures that a variable will always refer to an object whose type is the base type of the qualified type of the variable.

The form of an expression determines its qualified type. An expression is either a variable or a procedure invocation.[1] In the former case, T_e is the qualified type of this variable, and we have now defined the rule of binding. For example, consider variables a and b:

a: AssociativeMemory {GetVal, Insert}
b: AssociativeMemory {GetVal}

$b \leftarrow a$ is legal, but $a \leftarrow b$ is not. This is illustrated in Figure 2. In Figure 2a, an initial configuration is shown in which a refers to an *AssociativeMemory* object α, and b refers to an *AssociativeMemory* object β. Figure 2b shows the result of $b \leftarrow a$. Both b and a now refer to α. A new access path (from b to α) has been created as a result of this binding, but no new rights to α are obtained by it; in fact, the new access path via b has fewer rights to α than the old access path. Figure 2c illustrates what would be the result of $a \leftarrow b$. If this binding were allowed, the new access path from a to β would allow more rights than the old one; therefore the binding is not

1. We are treating all built-in operations as procedures, irrespective of how they are implemented.

Figure 2. Binding.

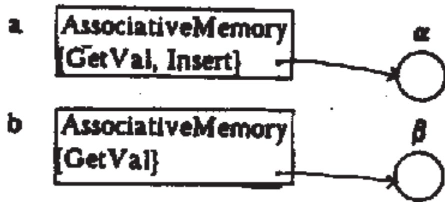


Figure 2a. The initial state.

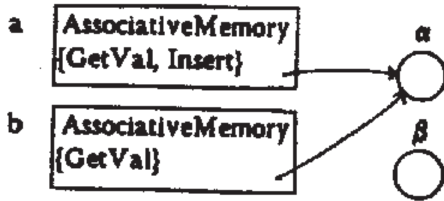


Figure 2b. Result of b ← a.

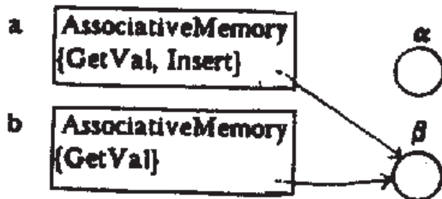


Figure 2c. Result of a ← b (disallowed).

---

permitted.

When an expression is a procedure invocation, the qualified type of the expression is determined by the procedure definition. A procedure definition has the form

procedure <procname> (<formals specification>) **returns** <result specification>
<body>
end <procname>

where <formals specification> specifies the name and qualified type for each formal parameter, and <result specification> specifies the qualified type returned by the procedure. The qualified

type of the invocation expression is the type specified in the <result specification>.

To understand whether a procedure invocation is legal, we must examine the semantics of parameter passing. Our notion of parameter passing is defined in terms of binding. Each formal parameter is a local variable of the procedure; this variable is created at invocation, and the actual parameter is bound to it. The procedure invocation is legal if the bindings of actual to formal parameters are legal.

For example, suppose a procedure P has type requirements

procedure P (x: T1{f,g}) **returns** T2{k}

and declarations

a: T1{f,g,h}
b: T2{k}

occur in the invoker of P. Then the statement $b \leftarrow P(a)$ is legal because the invocation $P(a)$ is legal ($x \leftarrow a$ is legal), and the qualified type of the invocation expression is $T2\{k\}$ and therefore the binding to b is legal. However, $b \leftarrow P(c)$, where $c: T1\{f,h\}$, is not legal because $x \leftarrow c$ is not legal.[2]

The question of whether a procedure definition is access-correct can be answered independently of any invocation of that procedure. A procedure definition is access-correct provided that all bindings within it are legal, and for every return statement:

return <expr>

the qualified type of <expr> is greater than or equal to the qualified type in the procedure <result specification>.

---

2. Our notation can easily be extended to permit the invoker to specify that a subset of rights is available, e.g., $P(a\{f,g\})$.

2.2 Remarks

Our semantics was chosen to model systems in which controlled sharing of objects is fundamental: the sharing of actual objects (rather than just copies of the values of objects) leads both to interesting behavior (e.g., many programs working with the same data base), and the need to exercise control over exactly how the objects should be shared. Sharing exists whenever two or more variables contain the same object-id; the binding operation causes sharing by storing an object-id in a variable. Note that sharing is significant only if operations exist to change the value inside the object. For example, sharing is significant for *AssociativeMemory* objects, since *AssociativeMemory* operations *Insert*, *Change* and *Delete* modify the values inside of an *AssociativeMemory* object. On the other hand, sharing of integers causes no problems if none of the integer operations modify the values in integer objects.

In conventional ALGOL-like languages, variables hold the values themselves rather than pointers to values, and assignment copies a new value into a variable. To support this view, we must define an assignment rule analogous to our binding rule. Although more elaborate schemes can be invented, a simple and probably sufficient rule is:

v := e

where T_v and T_e are the types of v and e, respectively, is legal provided

$$T_v \leq T_e$$

and furthermore

read $\in$ *rights* (T_e)
store $\in$ *rights* (T_v)

where the "read" right permits the value of a variable to be read, and the "store" right permits a

new value to be written in a variable. Almost all types would provide these rights.

The binding rule described above only permits a decrease in the rights available to an object. An object is originally obtained by invoking a creation operation of its type.[3] For both built-in and user-defined data types, the creation operations provide objects with full rights whenever they are invoked, as is illustrated by the *MakeMem* operation in the *AssociativeMemory* example shown in Figure 1. Thus the creator of an object obtains all rights to it. As the object is passed from one access-correct procedure to another, certain rights may be removed, but rights are never gained. This is true because binding is the only method provided for passing objects between procedures.

Sometimes, however, it is necessary for the called procedure to obtain *more* rights to the object than the caller had. When this occurs it is called *amplification* [5]. In our model, we permit amplification to occur at only one point: at entry to a procedure implementing a primitive operation of a type. Implementation of a new type would be done by means of a linguistic construct, such as the SIMULA class, CLU cluster, or Alphard form, that provides implementations for the primitive operations of the type in terms of a representation selected for the objects. Outside of the type definition, (direct) access to the representation is not possible, but inside each operation such access is needed and must be permitted. Thus, when it is invoked, a primitive operation obtains additional rights to each object of its type passed in as a parameter because it obtains rights associated with the object's representation type.[4]

Even within the operations of a type, access-control restrictions cannot be violated; amplification occurs on parameter objects of the type, but thereafter all bindings must obey the

---

3. Objects may also be obtained by asking the file system for them. This case is discussed in Section 4.

4. A more thorough treatment of amplification is given in [7].

binding rule. However, since extra rights are obtained at entry to a primitive operation, it is possible that the operation may make some of these rights available to its caller; indeed the operation may have been provided precisely for this purpose. Thus, amplification makes it possible for an access-correct procedure to obtain additional rights to an object.

Since it is the type definition that determines the behavior of the type, a user must study the type definition to determine exactly what sharing an object of that type means. Restricting amplification to type definitions ensures that only the type definition must be studied. A less stringent restriction on amplification, for example, to permit procedures outside the type definition to obtain additional rights, would mean that procedures outside the type definition must be considered to determine the behavior of objects. This is counter to the philosophy guiding the design of the class of languages we are considering. Our restriction on amplification matches the restriction in CLU and Alphard on access to the representations of user-defined objects.

3. Sharing Structured Objects

The access-control rules described in the previous section provide control over the sharing of objects that are passed directly from one procedure to another. However, they are inadequate to control sharing of objects passed indirectly -- through the medium of another object. For example, suppose a number of procedures share a data base of bank account records. Our rules can be used to control the sharing of the data base as a whole, but there is no way to also control sharing of the individual bank account records stored in the data base.

To discuss this problem further, we must introduce a notation that permits us to talk about both the structure[5] as a whole (the data base) and the elements or components of the structure (the

5. We will refer to types providing storage for component objects as "data structures" or "structures."

bank account records). The data type to be described is "data base of bank accounts," which is similar to data types already existing in programming languages such as "array of integers." The notation we will use is the following:

<data structure type name>[<element type names>]

Examples are

DataBase[BankAccount]
array[integer]

Both the data structure and the element types can be qualified. To specify qualified structured types we will use the notation

$$T[Q_1,...,Q_n]\{r_1,...,r_m\}$$

where T is the name of a structure type (for instance array, record or DataBase) for which rights $r_1,...,r_m$ are defined, and $Q_1,...,Q_n$ are the qualified types of the n kinds of elements in the structure. In the following discussion we will limit ourselves to structures containing a single kind of element; this simplifies the discussion without loss of generality.[6]

Suppose that we wish to write a program, *AccountSort*, to sort an array of bank accounts by account number. The program is not permitted to modify the bank accounts or even to determine the amount of money on deposit in the accounts. Assume the rights to bank accounts include *Deposit*, *Withdraw*, and *AccountNo*, and that all these rights permit the use of operations of the same name. Then the access control needs of procedure *AccountSort* to the array can be

---

6. Limiting what appears between the square brackets to just types is another simplification. It is easy to permit other compile-time-known quantities to appear between the brackets (for example, the selector names for components of records); an extension to quantities not known until execution time (for example, array bounds or limited ranges of element values) can also be made, but at the expense of run-time checking.

expressed:

procedure AccountSort (a: array[BankAccount{AccountNo}]{all})

Another legitimate array type, one that might be used by a caller of *AccountSort*, is

b: array[BankAccount{AccountNo, Withdraw}]{all}

We want the invocation *AccountSort(b)* to be legal. Intuitively, what we want is a binding rule that permits a structured object to be bound to a variable provided that the rights to the structure as a whole, and to the elements of the structure, do not increase. However, a straightforward extension of our rule is inadequate, as will be shown below.

Just as with simple data objects, a data structure such as array or DataBase may be characterized by a group of operations, including element *replacement* operations, which store new elements into the structure, and *retrieval* operations, which retrieve elements. For example, operations of interest for arrays are: *arraycreate*(*n*), which creates a new array with lower bound 1 and size *n*, *size*(*a*), which returns the size of array *a*, *update*(*a*, *i*, *x*), which binds object *x* into the ith element of array *a*, and *fetch*(*a*, *i*), which returns the object bound to the ith element of array *a*. Note that *update* is a replacement operation and *fetch* is a retrieval operation.

A data structure type name stands for a set of types, containing a different type for each possible combination of element types of the structure. Thus, "array" names the set of types containing among other elements

array[integer]
array[string]

The types in this set of types differ from one another only in the kinds of elements the arrays contain. Each type (in the set) is associated with a group of array operations that are specialized to

work for the particular element type by an appropriate selection of types for their input and output parameters. For example, the parameter and return types of the operations for the type *array[string]* are (ignoring qualifications on integer and string elements)

```
procedure arraycreate (size: integer) returns array[string]{all}
procedure size (a: array[string]{size}) returns integer
procedure fetch (a: array[string]{fetch}, i: integer) returns string
procedure update (a: array[string]{update}, i: integer, s: string)
```

Qualifications on the element type of a data structure type indicate the rights to elements that are potentially available throughout the lifetime of each object of the data structure type. A creation operation provides a new data structure with the stated set of rights available for each element. Element retrieval operations provide at most that many rights, while the user of replacement operations must provide at least that many rights to a new element. When the structure is passed to a procedure with fewer rights to elements, then a retrieval operation will provide only the smaller set of rights. However, it would be incorrect to invoke a replacement operation with the smaller set of rights because other programs that share the structure make use of the larger set of rights to elements.

For example, consider procedures *P* and *Q*:

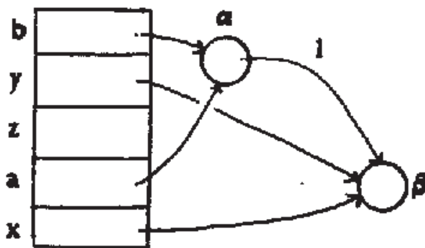
```
procedure P (a: array[ T{f} ]{all}, x: T{f});
        . . .
        update (a, 1, x);
end P;

procedure Q (b: array[ T{f, g} ]{all}, y: T{f, h});
        z: T{g};
        P(b, y);
        z ← fetch (b, 1);
        . . .
end Q;
```

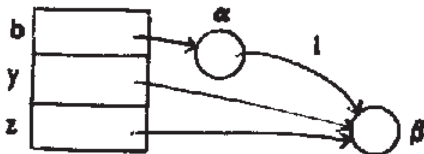
Both of these procedures appear to be access-correct, assuming that *Q*'s invocation of *P* is permitted. The invocation of *P* results in the situation shown in Figure 3a. Figure 3b shows the situation after *P* has returned and the binding resulting from $z \leftarrow fetch\ (b,\ l)$ is made. Note that access right *g* has been illegally obtained for object β.

In a capability-based system, this set of actions would eventually result in a run-time access-control error, most likely when *Q* attempted to use right *g* for object β. Note, however, that the source of the error is in *P*'s invocation of the array *update* operation. In the next section, we define an extended binding rule that permits certain invocations of *P* but guarantees that access-control errors, such as that described above, cannot arise.

Figure 3. Illustration of Simple Extension.



3a. The situation just before P returns.



3b. The situation after the binding z ← fetch (b, l).

3.1 Extended Rule of Binding

Our extended rule of binding permits a procedure to state precisely what limited rights it requires to all objects, including data structures and their component objects. Replacement operations are still allowed, but only if the bindings performed result in a structure that is consistent with the access-control assumptions made in all procedures that share the structure.

When a procedure is passed a data structure with restricted rights to the elements, information is lost about rights that the caller, and other procedures sharing the object, expect to have for the elements. All that is known is that the structure provides at least those rights to elements required by the called procedure. Consider the example of a procedure G, which accepts as a parameter an array of elements of type T with f and g rights, with full rights to use the array. A call to G will be legal only if it is passed an object of type

array[R]{all}

where $base(R) = T$ and $rights(R) \geq \{f, g\}$. Thus $R \geq T\{f, g\}$. We have just expressed exactly what is known inside G about the element type of the array. The notation we actually use is

```
procedure G (a: array[ ?R ]{all})
        where R ≥ T{f, g}
```

We will refer to types like R as *?types*; the "?" emphasizes that the rights associated with these types are not completely known.

A formal parameter declaration containing a "?" before the type identifier is called the *defining occurrence* of that ?type. Each ?type has a single defining occurrence. The ?type can be used in other declarations, but the "?" does not appear; for example, in

```
procedure H (a: array[ ?R ]{all}, b: array[ ?S ]{all}, c: S) returns R
        where R ≥ T{f, g}, S ≥ T{f, g}
```

the declaration of *b* is the defining occurrence of *S*, and this ?type is also used in the declaration of *c*. For simplicity, we limit defining occurrences of ?types to formal parameter specifications in procedure headings. Within a procedure body, ?types may be used to declare new variables in the usual way. For example, inside *H*,

```
c: array[S]
w: R
```

are legal declarations.

The association of actual type values with ?types is made at procedure invocation. Legality of a procedure invocation having ?types for some of its formal parameter types is checked as follows. Each ?type is matched with the type of the actual parameter passed in the position of the defining occurrence of the ?type. The match succeeds only if the type of the actual satisfies the constraints on the ?type stated in the **where** clause. If the match succeeds, it defines the unique qualified type associated with the ?type. Next, the declarations for the formals, and for the return type, are rewritten, replacing the ?types with the matched type values. Finally, the actuals are bound to the formals; if the bindings are legal (according to the rewritten declarations and the new binding rule discussed below), the invocation is legal.

For example, suppose the following declarations appear in the invoker of *H* (*H* is declared above):

```
x: array[ T{f, g, h} ]{all}
y: array[ T{f, g} ]{all}
u: T{f, g, h}
v: T{f, g}
```

The statement $u \leftarrow H(x, y, v)$ causes R to be associated with $T\{f, g, h\}$, and S to be associated with $T\{f, g\}$. Then the formal declarations are rewritten:

```
a: array[ T{f, g, h} ]{all}
b: array[ T{f, g} ]{all}
c: T{f, g}
```

and the return type is $T\{f, g, h\}$. This statement is legal since all the bindings of actuals to formals are legal, as is the binding of the returned object to u.

To define the extended binding rule, we must consider what assumptions can be made inside a procedure using ?types. Inside such a procedure, the qualified type associated with a ?type is not known, but a set of possible candidate for the qualified type is known. This set consists of all qualified types satisfying the constraints stated in the **where** clause. For example, inside of H ?types R and S are both known to be in the set

{T{f, g}, T{f, g, h}}

The binding rule is defined so that no extra rights can be obtained no matter which member of the set is associated with the ?type in the current invocation. Furthermore, no relationship can be assumed between two ?types with different names. In particular, even if two ?types are drawn from the same set, it is not possible to infer that one is less than or equal to the other. This can be illustrated by invocations of H: The invocation $H(x, y, v)$ causes $R \geq S$ while the invocation $H(y, x, u)$ causes $S \geq R$.

Now we are prepared to extend our rule of binding to cover the two additional cases introduced by data structures and ?types. We consider the binding

v ← e

where T_v and T_e are the types of v and e, respectively; again, we wish this binding to be legal if we are certain that $T_e \geq T_v$. In the case of data structures, the base type of qualified type $T[Q]\{r_1, ..., r_n\}$ is $T[Q]$ (where Q is the qualified element type), and we apply the binding rule of Section 2 directly: T_e and T_v must be identical up to the rights on the structure as a whole. For example, $y \leftarrow x$ is legal if

x: array[T{f, g}]{all}
y: array[T{f, g}]{fetch}

or if

x: array[S]{all}
y: array[S]{fetch}

where S is a ?type. Essentially this rule ensures that a reduction in rights to an element type can be accomplished only by introducing a new ?type.

If T_e and T_v are unstructured but involve ?types, a binding is legal only if it is legal no matter which member of the set is substituted for the ?type. For example, if R is a ?type known to be $\geq T\{f, g\}$ and y: R, then $x \leftarrow y$ is legal only if

x: R

or if the type of x is not a ?type and $T\{f, g\} \geq$ the type of x (e.g., x: $T\{f\}$). On the other hand, $y \leftarrow x$ is legal only if

x: R

or

x: T{all}

3.2 Remarks

The usefulness of the extended rule is demonstrated in Figure 4, which shows the implementation of the *AccountSort* procedure, discussed earlier, to sort an array of bank accounts by account number. A legal invocation of this procedure would be, for example, *AccountSort(b)*, where

b: array[BankAccount{all}]{all}

A sorting example is of interest because sharing of the object being sorted is necessary, and because it must be possible to read an element from the object being sorted, and later to write that element back into the object. Observe how the use of the ?type *R* enables this activity (the interchange of the ith and j^{th} elements of the array).

Figure 4. The AccountSort procedure.

```
procedure AccountSort (a: array[ ?R ]{all})
                where R ≥ BankAccount{AccountNo};

comment AccountSort sorts an array[BankAccount] by account number,
        using a bubble sort;

  index: integer{all} ← size (a);

  repeat
    bound: integer{all} ← index;
    index ← 1;
    for j: integer{all} ← 1 step 1 to bound - 1 do
      if AccountNo (fetch (a, j + 1)) < AccountNo (fetch (a, j))
          then begin
                temp: R ← fetch (a, j + 1);
                update (a, j + 1, fetch (a, j));
                update (a, j, temp);
                index ← j
                end
  until index = 1

end AccountSort;
```

The ?type notation requires only a slight extension to express information about general type parameters. So far, we have always related a ?type to some identified base type; for type parameters, what is needed is a notation that avoids naming the base type. A possible notation is

$T \geq$ **base**{f,g,h}

to indicate that any base type providing rights *f*, *g*, and *h* may be matched to the ?type *T*. For example, using this notation we could write a general sort procedure

procedure Sort (a: array[?R]{all})
    where R $\geq$ **base**{lessthan, equal}

which would sort an array of any element type providing rights *lessthan* and *equal*. Note that we are assuming that *lessthan* implies the presence of the "<" operation and that *equal* implies the presence of the "=" operation, and that the two operations together define a total ordering on the element type; similar assumptions must be made whenever general type parameters are permitted.

General type parameters are useful for implementing (user-defined) data structures. The replacement operations for data structures store element objects for later retrieval; on retrieval the rights available when the objects were stored are needed by the user. The ?types can be used to provide the user with rights to element objects without also providing those rights to the operations of the structure type. Therefore, a programmer can use a data structure as a repository for sensitive objects, without having to give the procedures implementing the operations of that structure access to the component objects.

Access-correct programs written in terms of ?types cannot lead to illegal acquisition of rights. This can be argued as follows. Binding that involves ?types is permitted only if it would be legal (according to the basic binding rule of Section 2) for any qualified type in the set determined by the constraints on the ?type. Therefore it must be legal whenever the actual type is

a member of that set.

4. Writing Long-Lived Programs

To simplify the presentation of our access-control mechanism, we focussed on the specification and the access-correctness of individual programs units. However, real programs of any size are composed of multiple program units. In the languages under discussion, two kinds of program units are of interest: procedures and type definitions. In this section we discuss the access-control aspects of the construction and use of long-lived and large programs. The issues involved include storing program units for use in more than one program, combining (separately compiled) program units to construct a program, and the storage and reacquisition of user-defined objects across multiple invocations of a program. Thus we are led to consider the interface between access control as provided in our language extension and in operating systems.

4.1 Sharing Long-Lived Objects

In systems where users share objects, there is usually some storage facility, which we will call a *library*, to provide for long-term storage and later reacquisition of data objects, including both programs and program units, as well as the data objects pertinent to various applications. The library must:

1. permit users to associate a symbolic name with an object.
2. retain the "name, object" pairs for retrieval.
3. only permit retrieval, deletion, or replacement of stored objects by authorized users or their programs.

We will discuss only the third feature of the library, for it is here that the language and operating

system access-control mechanisms interface. To be compatible with the access-control facility in the language, it is necessary for the library to retain information about the rights available to a stored object, as well as any constraints on which users can obtain access to it. In the following discussion, we will assume the existence of a capability-based operating system, since this is perhaps the most hospitable environment in which to build such a library.

When a request is made to the library to obtain access to a long-lived object, the library must perform authentication of the requestor before granting access. Here the library must rely on the operating system. We will sketch only a single alternative design for achieving this authentication. We assume that with each object the library maintains an authorization list [13]. Each entry in the list gives the name of a user and a capability for the object. The capability specifies the base type of the object, and the rights that the named user has to the object. (Recall that the qualified type of a variable and the operating system capability are similar.)

When some program (on behalf of a user) requests access to an object, it specifies the symbolic name of the object. If the symbolic name is known, the library invokes the operating system requesting identification of the user on whose behalf the request is made. The library then returns the capability associated with that user (if any) in the authorization list, effectively granting the user's program the ability to access the object. (For simplicity, we have described authentication in terms of users. More general designs, if supported by the operating system, would allow programs or (user, program) pairs to have identities for access-control purposes.)

Linguistic extensions are needed to program the library and the programs that interface with it. Since users can define new types, these programs must deal with object types that were not defined when the programs were written. Enumeration of the types of all objects that might be presented as parameters to an invocation of a library operation, for example, is impossible. Therefore, the language must then include syntax and semantics to permit parameter objects of *any*

type. This can be accomplished by providing a type *any*. A variable of type *any* can be bound to an object of any type; such a binding, for example $v \leftarrow e$, causes information about the qualified type T_e of the right hand side to be retained in the type *any* variable. Later, information about this qualified type can be recovered by coercion: an expression of type *any* can be *coerced* to some stated qualified type provided this qualified type is less than or equal to the original qualified type. The combination of type *any* and coercion permits programs to save objects in the library, and then later to retrieve and manipulate them.

4.2 Program Construction

To construct a program out of multiple program units, we require a policy for controlling when a program unit is made available for use. One possible policy is to assume, as is done in languages like PASCAL or ALGOL 60, that an entire program text, including the texts of all program units, is compiled at one time. Then the scope rules of the language determine where a type or procedure is known and usable.

A more promising alternative for long-lived programs is a policy permitting program units to be separately compiled and stored in a program library. If this approach is followed, then the issue of program unit availability becomes a subproblem of the general problem discussed above of retrieving objects from the library: A user wishing to bind two program units together must have the authorization to retrieve "execute" capabilities for both units from the library.

If program units are to be compiled separately, then a method is required for ensuring that the interface requirements of a program unit are satisfied by every invocation of that unit. The access-control mechanism requires only a slight extension of a method that provides type checking for independently compiled program units (see [10] for a method that does this checking at compile time). In particular, the method must guarantee that every use of a program unit

provides actual parameters of a qualified type greater than or equal to the qualified type required for the formal parameters.

Another issue in program construction involves the question of what the availability of a type definition means. Recall that a type definition defines a set of operations; the question we are addressing is whether availability of the type definition implies availability of all the operations or a subset of the operations. We have made the simple assumption that if a type definition can be used at all, then any operations defined in the type definition can be used. (Whether invocation of such an operations fails due to insufficient or incorrect rights for actual parameter objects is a different question.) Note that alternative policies are feasible and may be desirable. For example, a language supporting separate compilation of program units could require that the use of each operation be separately authorized. This is similar to the approach used in the Hydra operating system [6], and in the programming language Gypsy [1].

5. Discussion

In the preceding sections, we have proposed the semantics and syntax of a programming language extension to permit controlled sharing of data objects. In Section 2 we introduced notation and rules sufficient for controlling the sharing of simple, unstructured objects. In Section 3, we extended our notation and rules to permit controlled sharing of both data structures and the elements contained within them; we then showed how a slight further extension gave us the ability to have general type parameters. In Section 4, we discussed the interaction of the access-control facility with the construction and use of long-lived programs. In the course of this discussion, we identified the need for a non-enumerated, discriminated union mechanism (type *any* and the coercion mechanism), if programs such as the library are to be programmed within the extension.

The goal of statically performing the maximum amount of access-control checking

influenced the definition of our binding rule. If more run-time checking were deemed desirable, the rule could be altered so that currently disallowed bindings would be permitted. We believe, however, that the effect of program execution should be clear when only the static program text is considered, so that relaxing the restrictions of the extended binding rule is inappropriate. Furthermore, with the addition of type *any*, we believe the extension is powerful enough to be useful in practical program construction.

Several areas associated with the extensions are worth further investigation. One topic is to extend our mechanism to express additional access constraints; for example, constraints based on the contents of the objects may be of interest. Another topic is to study the implementation of our language extension to determine the set of programs and the set of properties on which access-correctness depends. For example, it might be possible to structure a compiler in such a way that access-correctness depends on only a subset of the modules. This work is similar to ongoing work in operating system security [11, 12], and can be expected to enhance understanding of program specification and verification. Furthermore, only after such work has been done can the relationship between our notion of access-correctness, and the acquisition and use of accesses in programs, be fully understood.

In the introduction, we argued that the access-control extension was worthwhile because it would enhance program understandability and verifiability. Certainly the extension permits a kind of program property, access-correctness, to be defined and statically checked. To argue that the extension is truly an aid to the production of correct software, however, it is necessary to show that the constraints that can be expressed are of interest in a large class of programs, and are important enough to compensate for the linguistic complexity arising from the incorporation of the mechanism. The extension is clearly beneficial for programs that are written to run on operating systems providing access-control mechanisms. We believe that the extension is beneficial in a more

general environment as well, and that the presence of the extension will lead to a style of programming in which careful attention to access-control improves program structure. Only by practical experience in using the extension can this claim be evaluated.

ACKNOWLEDGEMENTS

We would like to express our appreciation to the referees and to our colleagues, particularly those working on CLU and Alphard, whose criticism and observations have helped us prepare this paper.

REFERENCES

[1] Ambler, A. L., et al. GYPSY: A language for specification and implementation of verifiable programs. *Proc. of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12*, 3 (March 1977), 1-10.

[2] Dahl, O. J., and C. A. R. Hoare. Hierarchical program structures. *Structured Programming* (Dahl, Dijkstra, and Hoare, Eds.), Academic Press 1972.

[3] Dennis, J. B., and E. C. Van Horn. Programming for multiprogrammed computations. *Comm. of the ACM 9*, 3 (1966), 143-155.

[4] Fabry, R. Capability based addressing. *Comm. of the ACM 17*, 7 (July 1974) 403-412.

[5] Jones, A. K. *Protection in Programmed Systems*. Ph.D Thesis, Carnegie-Mellon University, Department of Computer Science, 1973.

[6] Jones, A. K., and W. A. Wulf. Toward the design of a secure system. *Software Practice and Experience 5* (1975), 321-336.

[7] Jones, A. K., and B. H. Liskov. A language extension for controlling access to shared data. *IEEE Trans. on Software Engineering SE-2*, 4 (December 1976), 277-285.

[8] Lampson, B. W. Protection. *Proc. of the Fifth Annual Princeton Conference on Information Sciences and Systems*, Princeton University, 1971, 437-443.

[9] Lampson, B. W., and H. E. Sturgis. Reflections on an operating system design. *Comm. of the ACM 19*, 5 (May 1976), 251-190.

[10] Liskov, B. H., A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. To appear in *Comm. of the ACM*.

[11] Neumann, P. G., R. S. Fabry, K. N. Levitt, L. Robinson and J. H. Wensley. On the design of a provably secure operating system. *International Workshop on Protection in Operating Systems*, IRIA, Paris, August 1974, 161-175.

[12] Popek, G. J., and D. A. Farber. A model for verification of security in operating systems. To appear in *Comm. of the ACM*.

[13] Saltzer, J., and M. Schroeder. Protection of information in computer systems. *Proc. of the IEEE 63*, 9 (September 1975), 1278-1308.

[14] Tanenbaum, A. S. A tutorial on ALGOL 68. *Computing Surveys 8*, 2 (June 1976), 155-190.

[15] Wirth, N. The programming language PASCAL. *Acta Informatica 1*, 1971, 335-363.

[16] Wulf, W. A., E. Cohen, W. Corwin, A. K. Jones, R. Levin, C. Pierson and R. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Comm. of the ACM 17*, 6 (1974), 337-345.

[17] Wulf, W. A., R. L. London and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Trans. on Software Engineering SE-2*, 4 (1976), 253-265.