MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Computation Structures Group Memo 150

Programming Methodology Group
Progress Report 1975-76

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant DCR74-21892.

July 1977

1. Introduction

The goal of the research of the Programming Methodology Group is the development of tools and techniques that ease the production of quality software, software that is reliable and relatively easy to understand, modify, and maintain. Our work is based on a programming methodology in which the recognition of abstractions is the key to problem decomposition. A program is constructed in many stages. At each stage, the problem to be solved is how to implement some abstraction (the initial problem is to implement the abstract behavior required of the entire program). This is done by performing the following four steps:

1. Problem Decomposition. The programmer envisions a number of subsidiary abstractions that are useful in the problem domain.

2. Specification. The behavior of each abstraction is specified precisely.

3. Implementation. Once the behavior of the subsidiary abstractions is understood and specified, they can be used in a program to implement the original abstraction.

4. Verification. The programmer verifies that the implementation is correct, assuming that the subsidiary abstractions are implemented correctly.

As soon as step (2) has been performed, new problems exist concerning how to implement the abstractions defined in step (2). The programmer can choose to work on one of these problems immediately, before steps (3) and (4) have been carried out for the current stage. The process terminates when all abstractions generated during design are realized either by programs or by the programming language in use.

To make effective use of this methodology, it is necessary to understand the nature of the abstractions that are useful in constructing programs; this includes what is being abstracted, and what form the abstraction takes. In studying this question, we identified three kinds of useful abstractions: procedural, control and especially data abstractions. While the procedural abstraction, which performs a computation on a set of input objects, and produces a set of output objects, has long been recognized as useful, control and data abstractions have been neglected in discussions of programming methodology.

A control abstraction defines a method of sequencing arbitrary actions. All languages provide built-in control abstractions; examples are the **if** statement and the **while** statement. In addition, however, it is helpful to allow user definitions of a simple kind of control abstraction, which is a generalization of the repetition methods (in particular, the **for** statement) available in many programming languages. Frequently the programmer desires to perform the same action for all the objects in a collection, such as all the characters in a string or all items in a set. The simple control abstraction permits the action to be described separately from the method of obtaining the objects in the collection.

A data abstraction is used to introduce a new type of data object that is deemed useful in the domain of the problem being solved. At the level of use, the programmer is concerned with the *behavior* of these data objects, what kinds of information can be stored in them and obtained from them. The programmer is *not* concerned with how the data objects are represented in storage, nor with the algorithms used to store and access information in them. In fact, a data abstraction is often introduced to delay such implementation decisions until a later stage of design.

The behavior of the data objects is expressed most naturally in terms of a set of operations that are meaningful for those objects. This set will include operations to create objects, to obtain information from them, and possibly to modify them. For example, push and pop are among the meaningful operations for stacks, while meaningful operations for integers include the usual arithmetic operations.

Thus, a data abstraction consists of a set of objects and a set of operations that characterize the behavior of the objects. To ensure that a data abstraction can be understood at an abstract level, we require that the set of operations *completely* determine the behavior of the data objects. This property can be achieved by making the operations the *only direct means* of creating and manipulating the objects.

The Programming Methodology Group is involved in two main areas of research that support the above methodology:

1. We are developing the programming language, CLU, which provides linguistic support for programming with abstractions. Data and control abstractions are not supported well by conventional languages.

2. We are developing techniques for specifying the meaning of abstractions, and for verifying the correctness of programs written in terms of abstractions.

In the following sections we discuss some of our accomplishments of the past year. In the next section, we describe how CLU supports the use of control abstractions. (A comprehensive treatment of the abstraction mechanisms in CLU can be found in [17].) In Section 3, we discuss how a language like CLU can be extended to incorporate an access control facility. Section 4 contains a discussion of optimization techniques for a CLU-like language. In Section 5, our work on specification of data abstractions is described.

2. Iterators

The purpose of many loops is to perform some action on all of the objects in a collection. For such loops, it is often useful to separate the selection of the next object from the action performed on that object. CLU provides a control abstraction mechanism that permits a complete decomposition of the two activities. The **for** statement available in many programming languages provides a limited ability in this direction: it allows iteration over ranges of integers. The CLU **for** statement allows iteration over collections of any type of object. The selection of the next object in the collection is done by a user-defined *iterator*. The iterator produces the objects in the collection one at a time (the entire collection need not physically exist); the objects are then consumed by the **for** statement.

We illustrate the use of iterators by means of a simple example. Figure 1 shows an iterator called *string_chars*, which produces the characters in a string in the order in which they appear.

Figure 1. Use and definition of a simple iterator.

```
count_numeric = proc (s: string) returns (int);
        count: int := 0;
        for c: char in string_chars (s) do
                if char_is_numeric (c)
                        then count := count + 1;
                        end;
                end;
        return count;
        end count_numeric;

string_chars = (s: string) yields (char);
        index: int := 1;
        limit: int := string$size (s);
        while index <= limit do
                yield string$fetch (s, index);
                index := index + 1;
                end;
        end string_chars;
```

This iterator uses string operations *size (s)*, which tells how many characters are in the string *s*, and *fetch (s, n)*, which returns the *n*th character in the string *s* (provided the integer *n* is greater than zero and does not exceed the size of the string).

The general form of the CLU **for** statement is

```
for declarations in iterator-invocation
        do body end;
```

An example of the use of the **for** statement occurs in the *count_numeric* procedure (see Figure 1), which contains a loop that counts the number of numeric characters in a string. Note that the details of how the characters are obtained from the string are entirely contained in the definition of the iterator.

Iterators work as follows: A **for** statement initially invokes an iterator, passing it some arguments. Each time a **yield** statement is executed in the iterator, the objects yielded[1] are assigned to the variables declared in the **for** statement (following the reserved word **for**). Then the loop body is executed. Next the iterator is resumed at the statement following the **yield** statement, in the same environment as when the objects were yielded. When the iterator terminates, either by an explicit **return** statement (which must not return any objects) or by completing the execution of the body, then the invoking **for** statement terminates.

For example, suppose that *string_chars* is invoked by *count_numeric* with the string "a3". The first character yielded is 'a'. At this point within *string_chars*, *index* = 1 and *limit* = 2. Next the body of the **for** statement is performed. Since the character 'a' is not numeric, *count* remains at 0. Next *string_chars* is resumed at the statement after the **yield** statement, and when resumed,

1. One or more objects may be yielded, but the number and types of objects yielded each time by an iterator must agree with the number and types of variables in a **for** statement using the iterator.

index = 1 and *limit* = 2. Then *index* is assigned 2, and the character '3' is selected from the string and yielded. Since '3' is numeric, *count* becomes 1. Then *string_chars* is resumed, with *index* = 2 and *limit* = 2, and *index* is incremented, which causes the **while** loop to terminate, and the iterator to terminate. This terminates the **for** statement, with control resuming at the statement after the **for** statement, and *count* = 1.

While iterators are useful in general, they are especially valuable in conjunction with data abstractions that are collections of objects (such as sets and arrays). Iterators afford users of such abstractions access to all objects in the collection, while exposing a minimum of detail. Several iterators may be included in a data abstraction. Where the order of obtaining the objects is important, different iterators may provide different orders.

3. Access Control

One of the most important attributes of a programming language is the way the scope rules of the language define how data is to be shared among the individual program units (procedures, blocks, modules) out of which a program is constructed. Ordinarily, access to data is provided on an all-or-nothing basis: if a module has access to some data base, then every component of the data base is accessible, and every possible type of access (usually just reading and writing) may be performed. Experience in building large applications, or applications involving sensitive data, has indicated that sharing of data is enhanced if finer control than all-or-nothing access is provided. For example, manipulation of the information in a data base is much more controlled if not every program that reads the data base is also permitted to write it. In addition, if some of the information in a data base is sensitive, then control over which programs can read which information is also desired.

Current programming languages are deficient in providing mechanisms for controlling the sharing of information among program units. For example, passing a data base "by value" ensures that the called procedure may not modify the data base. However, this mechanism does not provide control over what parts of a data base may be read; in addition, it is so expensive for large data bases that other parameter passing mechanisms (for example, call by reference) are used instead. Proposals for avoiding the overhead of call by value while retaining the benefit that the data base cannot be modified (for example, call by reference, but permitting only read access to the formal parameter) solve the efficiency problem, but still do not provide for selective reading of the data base. In addition, such proposals do not provide for the control of selective alteration of the data base.

B. Liskov and A. Jones[1] have investigated a programming language extension that provides for controlled sharing of data [12]. The approach taken 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 [24, 26].

To incorporate an access control mechanism in a programming language, we have chosen an approach that permits programmers to express access control restrictions in terms that are meaningful to their application domains. We assume that all data are contained in *objects* for which there exists a set of *accesses*. Objects are those entities, such as data bases, libraries, stacks or files, that are of interest to programmers. Accesses are limited to those that are meaningful manipulations of the objects; accesses are the only means for altering an object or extracting information from it. In some cases, meaningful accesses are the familiar read, write, and, possibly, execute access. In other cases, the accesses themselves are user-defined, tailored to the abstract notion the user intends to capture. For example, a file system may distinguish between write access and append access. In contrast to a write access, an append access is assumed to modify the file, but not to alter existing content. This permits a user to share a file with others, allowing them to augment the file by appending to it, but not allowing them the ability to rewrite any portion of it.

Thus, to discuss access control we require a language that permits the writing of programs in terms of data objects and the accesses that are meaningful for them. In particular, languages in which a datum is viewed as an aggregate of memory cells are not suitable, because of the difficulty of expressing access control on anything but a cell basis. One class of languages, including the languages SIMULA 67 [3, 4], CLU [17], and Alphard [28], provides a natural environment in which

1. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa.

to embed an access control facility. In these languages, a data type is viewed as a set of objects and a set of operations. The operations of a data type correspond very closely (though not identically, as we shall show) to our notion of access, and access control corresponds to the ability to control the use of the operations.

To accommodate access control, we will add one more component to a type: In addition to objects and operations, a type also specifies a set of rights. A *right* is a name that represents a meaningful manipulation of 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 2 for the type, *AssociativeMemory*. Operations for this type include an operation to create an empty *AssociativeMemory* object of a particular size (*makemem*), an operation to add a name-value pair to an *AssociativeMemory* (*insert*), an operation to change the value associated with a given name (*change*), an operation to fetch the value associated with a given name (*getval*), and an operation to remove a name-value pair (*delete*). 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 particular 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 all *integer* objects. In general, we can expect some rights to correspond to the use of a single operation, some to a group of operations and some to a single parameter of an operation taking more than one object of the type.

Embedding an access control facility in a programming language permits expression of access restrictions as an integral part of a program. In addition, the question of whether a program

Figure 2. 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)
```

---

obeys access control restrictions, and is thus *access-correct*, can be answered at compile time. This can lead to benefits similar to those derived from compile-time type checking: confidence that the program is access-correct, and enhanced efficiency over the dynamic mechanisms currently provided by operating systems.

3.1 Basic Model

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. Our notation for access control involves a declaration for each variable of 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 is written

T{r1,...,rn}

where *T* is the name of some type, and {*r1*,...,*rn*} is a non-empty subset of the rights of *T*. We refer to the two parts of a qualified type as the base type and the rights; if *Q* is a qualified type, then *base*(*Q*) is the base type and *rights*(*Q*) is the rights. For example, the following are some of the qualified types derived from 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 unique name that is interpreted by the underlying addressing mechanism to select an 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. Note that is is possible for sharing of objects to take place, because two variables may contain the same object id. In this case, the qualified type in the two variables may differ, but the binding rule (discussed in the next section) ensures that the base type is necessarily the same.

A variable contains a capability in the operating system sense [5, 14]. 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 object id in the variable). Thus, using one path rather than another to name an object changes 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.

Our notions of variable, object and binding are different from the related notions of value and assignment that underlie block-structured languages. This difference is illustrated in

Figure 3. Figure 3a shows the traditional view of variables and values, in which the value resides in the variable and a new value can be copied into a variable by means of assignment. Figure 3b illustrates our semantics: a variable is bound to an object, and a value is contained in an object. This value may be accessed or modified only by means of one of the operations of the object's type. Our rule of binding differs from assignment in that it causes *sharing* of the object involved, rather than the copying of the value in the object. Furthermore, this sharing is significant since, for some types of objects, operations exist to change the value inside of the object. For example, the *AssociativeMemory* operations *insert*, *change* and *delete* modify the value inside of an *AssociativeMemory* object.

Our notion of binding corresponds to assignment involving variables holding (typed) references to objects. Some programming languages are based on a semantic model like ours. The most widely known of these languages is LISP [18]; LISP lists are objects (with operations car, cdr, and cons) and LISP setq is similar to our binding. Our model is also used in SIMULA 67 and CLU.

We believe that our semantics models very well what is going on in systems where

---

Figure 3. Comparison of Semantic Models

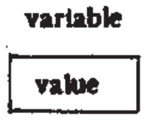

Figure 3a. Traditional view of variables and values.

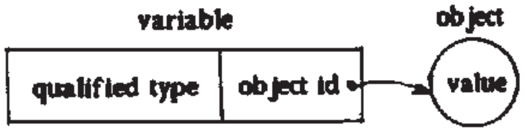

Figure 3b. Model used in this paper.

controlled sharing is of interest. Note that sharing of objects is a fundamental fact in these systems; 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 some control over exactly how the object should be shared. Protection schemes exist to provide this control.

3.2 Binding Rule

A single rule, governing the legality of binding of objects to variables, is sufficient to provide the required access control and is the basis for determining whether a program is access-correct (obeys the access control restrictions). Binding is the operation that causes a variable to refer to an object (by changing the object id). The effect of binding is creation of a new access path for the object. Therefore, to ensure that a program is access-correct, we must guarantee that no new access rights to the object are obtained from this new access 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 new 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 this 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.

An expression is either a variable, in which case its qualified type is the same as the qualified type of the variable, or it is a procedure invocation. In the former case, we have now defined the rule of binding (since T_e is the qualified type of this variable). For example, suppose

a: AssociativeMemory{getval, insert}
b: AssociativeMemory{getval}

Then $b \leftarrow a$ is legal, but $a \leftarrow b$ is not. This is illustrated in Figure 4. In Figure 4a, an initial configuration is shown in which *a* refers to an AssociativeMemory object α, and *b* refers to an AssociativeMemory object β. Figure 4b 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 4c 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, and therefore the binding must not be permitted.

In order to understand binding when the right-hand side is a procedure invocation, we

Figure 4. Binding.

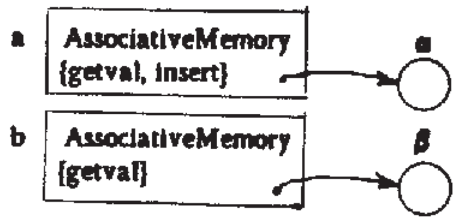

Figure 4a. The initial state.

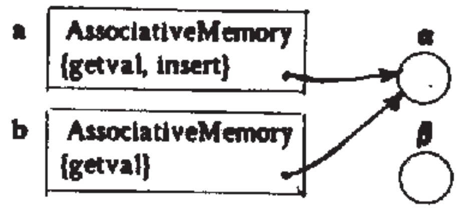

Figure 4b. Result of b ← a.

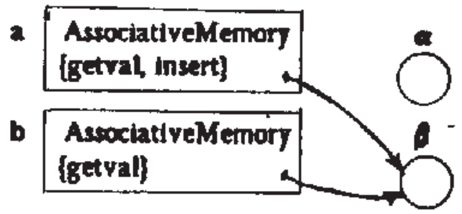

Figure 4c. Result of a ← b (disallowed).

must examine the semantics of parameter passing. Our notion of parameter passing is defined in terms of binding. 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. Each formal parameter is considered to be 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 only if the bindings of actual to formal parameters are legal. The qualified type of the invocation expression is then the type specified in the <result specification>.

For example, suppose a procedure P has type requirements

```
procedure P (x: T1{f1,f2}) returns T2{g1}
```

and declarations

```
a: T1{f1,f2,f4}
b: T2{g1}
```

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 object returned by P has qualified type $T2\{g1\}$ and therefore may be legally bound to b. However, $b \leftarrow P(c)$, where $c: T1\{f1,f3\}$, is not legal because the invocation $P(c)$ is not legal ($x \leftarrow c$ is not legal).

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

Procedure invocation is the mechanism whereby objects are created in the first place. There exist a number of primitive data types (for example *integer, boolean, array*). The creation operations of these types provide objects of the type whenever they are invoked, and these objects are returned with full rights. For the non-primitive, user-defined types the situation is analogous. This has already been illustrated in the *AssociativeMemory* example shown in Figure 1; whenever the *makemem* operation for *AssociativeMemory* is invoked, it returns a new *AssociativeMemory* object with full rights. 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 sharing objects between procedures.

3.3 Discussion

The access control mechanism described above is sufficient to control the sharing of many of the kinds of objects of interest in programming. For example, suppose we define a type *employee-record*, with operations (and rights) to *read-job-category, write-job-category, read-salary*, and *write-salary*, among others. Using the rules defined so far, we can define a procedure

procedure P (x: employee-record{read-job-category, write-salary})

which computes a new salary based on the employee's job category, but is unable to change the job category, or to read the old salary.

The above discussion is intended to introduce the reader to the access control facility. A complete description of this facility, which includes the following additional topics, is given in [12]:

1. The use of *amplification* [10] in the program module defining a new type.

2. An extension of the binding rule to control sharing of objects passed indirectly — through the medium of another object.

3. A comparison of the access control facility with the dynamic mechanism present in the Hydra system [26, 11].

4. Optimization

One objection raised to the adoption of structured programming methods is that they produce inefficient programs. While we believe that the major cost of software is its construction and verification, the cost of executing programs can not be ignored. Both costs can be reduced by the use of program optimization techniques. The rationale for program optimization is nicely stated by W. Wulf, *et. al.* [27, p. 131],

> The reason that compiler optimization is important is that programmer efficiency and execution efficiency need not be a choice we must make. Optimization is a technological device to let us have our cake and eat it, too -- to have *both* convenient and well-structured programming and efficient programs.

R. Atkinson has investigated an approach to optimization that is especially applicable to languages like CLU [1]. First a program is transformed by a technique known as inline substitution, which substitutes the bodies of procedures for certain invocations of those procedures. This transformation tends to increase the size of the transformed program, but tends to decrease the execution time by eliminating procedure call overhead, and by enabling more global optimizations. Then the data and control flow of the transformed program is obtained using symbolic interpretation. Finally, standard optimization techniques, such as constant propagation, are performed, making use of the data and control flow information and, in addition, information about properties of procedures and about the interaction among the operations of a data abstraction.

4.1 Inline Substitution

Inline substitution reduces execution time by eliminating the overhead involved in using the procedure call mechanism. The size change resulting from a substitution is simply the difference between the size of the expanded invocation and the size of that part of the call mechanism originally present in the code. Coupled with these "direct" effects on space and time are corresponding "indirect" effects. Placing a procedure body in a specific context can present new opportunities for optimization using other techniques. These optimizations will generally reduce execution time even further, but their effect on program size will depend on the technique.

When procedure bodies are small, as they are in CLU programs, many optimization techniques are ineffective, simply because they require the presence of a substantial context. Thus, performing inline substitution before using other techniques may be the key to successful optimization of structured programs.

R. Scheifler has studied inline substitution as an independent optimization technique [23]. This study involved the analysis of the following problem: Given a program and constraints on the final program size, find a sequence of substitutions that minimizes the expected execution time, considering only "direct" effects.

A key phrase in this problem statement is "expected execution time". Some method is needed to determine the number of times an invocation is expected to execute. We believe a good method is to run the program using data selected by the programmer, and to count the number of times each invocation executes. These statistics can then be used as the initial expected numbers. They are "initial" numbers for two reasons:

1) Inline substitution can create new invocations, each of which must be assigned an expected number.

2) When the body of a procedure P is substituted for an invocation, P is no longer called as often, implying that new expected numbers must be assigned to invocations contained in P.

To completely determine how expected numbers change, the control flow history must be retained in the statistics, necessitating many counters for each invocation. However, a single counter will suffice if a simplifying assumption is made about control flow: For any procedure body and any invocation contained therein, the expected number of executions of the invocation per execution of the body is constant. From this assumption a set of equations has been developed for calculating new expected numbers. The equations work when substituting for recursive as well as non-recursive invocations.

Using these equations, an algorithm to perform inline substituion can be formulated. However, as a practical matter, the problem of finding a set of substitutions that minimizes execution time is intractable. R. Scheifler has shown this problem to be NP-hard, meaning there is no known algorithm that will always solve the problem in polynomial time, and the existence of such an algorithm would imply polynomial-time algorithms for many classic hard problems [23].

An approximate solution to the problem has been developed, and is implemented for the current CLU system. The algorithm is built on a very simple heuristic: substitute for invocations that execute often but call small procedures. More precisely, at each step choose the invocation that will yield the greatest time savings per unit space increase. Continue until the maximum program size is reached. Lastly, while there is an invocation that is the sole remaining invocation of a non-recursive procedure, substitute for the invocation. This allows the procedure itself to be discarded, and so does not increase the program size.

Preliminary results using this algorithm indicate that, in programs with a low degree of recursion, over 90 percent of all procedure calls can be eliminated with little increase (-1 to 25 percent) in the size of compiled code, and with moderate savings (10 to 30 percent) in execution time.

4.2 Program Analysis

Following inline substitution, two kinds of program analysis are carried out. First, the program is analyzed to obtain information about its control flow and data flow. Then the flow information is analyzed to identify potential optimizations.

R. Atkinson has investigated a non-standard method for obtaining control and data flow information [1]. He has adapted the technique of *symbolic interpretation* [13], in which a program is executed using symbolic objects rather than actual objects. Symbolic interpretation can be used to obtain both data and control flow information.

As an example of obtaining data flow information, suppose we have the procedure:

```
square = proc (x: int) returns (int);
   return x * x;
   end square;
```

The symbolic interpretation would start by associating a symbolic object (*1) with the variable *x*. Then the integer multiply operation would be interpreted to obtain another symbolic object (*2 = int$mul(*1, *1)). The object returned by the procedure is *2. The symbolic interpretation removes our dependence on variables, so that we are only concerned with the symbolic objects.

After performing symbolic interpretation on the program, the optimizer searches for transformations that will make the program less costly to execute. One such transformation is the replacement of redundant expressions by variables that hold previously calculated objects. The method used is to search the set of symbolic objects created by the symbolic interpretation for equivalent symbolic objects; then the control flow information provided by the symbolic interpretation is used to discover whether the calculation of one of the objects precedes the other. For example,

```
u := a[i]
 ...
v := a[i]
```

where *a* is an *array[t]*, for some type *t*, and *i* is an integer, can be transformed into

```
u := a[i]
 ...
v := u
```

provided that in the intervening code there are no assignments to variables *u*, *a* and *i*, and there are no side-effects that affect the equivalence of the objects in variables *u* and *v*. If *u* and *v* are found to contain equivalent symbolic objects, this guarantees that none of *u*, *a* and *i* have been assigned to in the intervening code. To determine whether a side effect has occurred, the optimizer requires information about the properties of the data and procedural abstractions used in the program being optimized. For example, the only side effect that could invalidate the substitution shown above is to update the ith element of the array object referred to by *a*. Thus, the information that use of the array update operations can affect the later use of the array fetch operation *a[i]* constitutes a property of arrays that is of interest to the optimizer.[1]

1. In CLU, *a[i]* is not considered to be a variable, but rather syntactic sugar for an invocation of an array operation. If *a[i]* appears on the right hand side of the assignment symbol, it stands for a call on the array fetch operation; if it appears on the left hand side, it stands for a call on the array store operation. The reader is referred to [17] for an explanation of CLU semantics.

4.3 Determining Properties of Abstractions

Some properties of data and procedural abstractions that we have found useful for optimization follow:

> (1) *mutability*: An object is mutable if the information in it can change over time, and immutable if all of its information is constant over time. A data abstraction is immutable if all of its objects are; otherwise the data abstraction is mutable. Integers and strings are immutable in CLU, while arrays and records are mutable.
>
> (2) *isolated representation*: A data abstraction has an isolated representation if the objects of that data abstraction can only be modified through operations of the abstraction.
>
> (3) *obscuring*: Procedure P obscures procedure Q if the execution of P modifies an object and Q uses the modified component.
>
> (4) *side-effect free*: A procedure P is side-effect free if executing P does not modify any objects existing prior to its execution. All procedures that implement mathematical functions are side-effect free, as well as many procedures that examine mutable objects.

The optimizer design we have proposed can use properties about abstractions. We assume these properties are computed prior to optimization and are stored in a data base. In general, however, it is costly (and sometimes impossible) to determine such properties. Therefore, R. Atkinson [1] has developed techniques that provide conservative approximations to the desired properties. Where the properties cannot be determined, worst-case assumptions are made (for example, if a data type cannot be shown to have immutable objects, the optimizer must assume that the objects are mutable).

In making these approximations, we depend on the notion of reachability for CLU objects. The only objects reachable are those in some basis set (such as the parameters passed to a procedure), or those objects that are reachable from other reachable objects. We call the set of objects that are reachable from some object X the *reachablity closure* of X.

Unfortunately, the reachability closures for mutable objects are dynamic, and cannot generally be determined prior to execution. We can approximate reachability closures, however, by noting that CLU data types partition the set of all CLU objects in such a way that objects in different partitions can never be reached from one another. Furthermore, a static structure *does* exist for CLU data types (once implementations have been selected for these types). We therefore define a *type closure* of an abstract type T to be the set containing T and all types in the type closure of the representation type of T (the type chosen to represent objects of type T, and referred to within a cluster implementing T as the **rep** -- see [17] for more information). The type closure of a basic type B (such as *integer*, *boolean*, *string*, *array*[...], and *record*[...]) is the union of the type closures of the type parameters to B and the set containing only B. As an example, the type closure of *array[integer]* is {*array[integer]*, *integer*}. As a second example, suppose that *array[integer]* is the representation type of the abstract type *stack[integer]*. Then the type closure of *stack[integer]* is {*stack[integer]*, *array[integer]*, *integer*}.

Given an object X of type T, then the type of every object in the reachability closure of X is in the type closure of T. For example, from any object of type *array[integer]* only objects of type *integer* or *array[integer]* can be reached, while from a *stack[integer]* object, only objects of type *stack[integer]*, *integer* or *array[integer]* can be reached.

The use of type closures may be illustrated by returning to our earlier example. Suppose the actual code segment was

```
u := a[i]
p(x, y)
v := a[i]
```

where *x*: *S* and *y*: *R*. If the union of the type closures of *S* and *R* does not include *array[t]*, then we can be certain that *a* is not modified in *p*, since *a* cannot be reached from either *x* or *y*.

Other closures can be constructed in much the same way as type closures. Two closures defined on procedures are the *mutability closure* and the *access closure*. The mutability closure of procedure P is the set of all types with mutable objects that can be changed during an execution of P. The access closure of procedure Q is the set of all types examined during an execution of Q. As with the type closure, these closures are ultimately derived from known properties of the basic CLU types. The mutability and access closures can be used to approximate the obscuring property for P and Q. We assume that P obscures Q if the intersection of the mutability closure of P with the access closure of Q is not the empty set.

Use of the obscuring property may permit optimizations that would be forbidden if only type closures were considered. In the example above, if the mutability closure of procedure *p* does not contain *array[t]*, then *p* does not obscure the first array fetch operation and therefore the second array fetch operation can be eliminated. This may occur even if *array[t]* were contained in the union of the type closures of *S* and *R*.

Not all properties useful to the optimizer can be approximated with closures. For example, using the above methods, we may be able to determine that the data abstraction, *stack[t]*, with operations *push*, *pop*, *top*, *size* and *equal*, has the following properties:

stack[t] objects are mutable

stack[t] has an isolated representation

top, size, and equal are side-effect free

push obscures top, size

pop obscures top, size

One additional property of interest would express the fact that push (or pop) only obscures top (or size) if the same stack object is given to both push and top. A further property expresses information about equivalence of symbolic objects. For example, after *push(s, v)*, we know that $v = top(s)$. Information of this sort could be used during program transformation to avoid the *top(s)* computation, and use a previously computed object.

Although closures cannot be used to approximate every property of interest, a considerable amount of information can be obtained from their use. Such information is needed for optimizing languages, like CLU, that provide data abstractions. The information would also be useful for optimizing programs with pointers.

5. Specifications for Data Abstractions

There are three methods for specifying data abstractions [15, 16]: axiomatic, state machine, and abstract model.

The most promising form of axiomatic specification is the algebraic technique, developed by Zilles at M. I. T. [29], using some results in algebra [2]. The technique was investigated further by Guttag at the University of Toronto [6], who worked out a criterion for recognizing a "sufficiently complete" axiomatization of a data type. Further work on verification of data types using this technique is in progress at ISI [7, 8].

The state machine approach was first proposed by Parnas [20]. The approach as originally proposed was informal. Work on formalization of this technique is underway [21, 22].

The abstract model approach has been used informally in [9]. During the past year, we have been studying the formalization of this technique. Some work in this area has also been done by Wulf et al. [28].

In [15], we developed some criteria for judging the desirability of a specification technique for data abstractions. Among the criteria were the ease of construction and understandability of the specifications. We believe that the abstract model specification technique is best with respect to these criteria; this is the motivation for our work on this technique.

In the remainder of this section, we discuss the work of V. Berzins on the abstract model technique. He has worked out the theoretical justification for this technique (which is also algebraic in nature). He has investigated the structure of the specifications, and has arrived at a form that, we believe, makes it easier to build specifications. He has also developed criteria for establishing consistency and completeness of abstract model specifications (analogous to those developed by Guttag [6] for algebraic specifications). These criteria are helpful in evaluating the specification of an abstraction, since a specification that is not well formed cannot define any behavior, let alone the intended behavior.

5.1 Abstract Model Specifications

A sample specification using the abstract model technique is shown in Figure 5. A sequential file data type is defined, which can be written in a restricted way: records can only be

Figure 5. Sample Abstract Model Specification

```
Type FILE[RECORD] is

Interface:
        create() --> FILE,
        append(FILE, RECORD) --> FILE ∪ {error(append-in-middle)},
        reset(FILE) --> FILE ∪ {error(file-empty)},
        skip(FILE, int) --> FILE ∪ {error(skip-past-eof), error(reverse-skip)},
        read(FILE) --> RECORD ∪ {error(file-empty)},
        eof(FILE) --> bool,

Representation:         tuple[ptr: int, s: sequence[RECORD]],

        Invariant:      For all f: FILE;
                0 ≤ f.ptr ≤ length(f.s) & (length(f.s) > 0 ==> f.ptr > 0),

        Equivalence:    For all (f1, f2): FILE;
                f1 = f2 ≡ (f1.ptr = f2.ptr & f1.s = f2.s),

Operations:     For all (f, f1, f2): FILE, r: RECORD, n: int;
        create() = tuple[ptr: 0, s: emptyseq()],
        append(f, r) = if f.ptr = length(f.s)  then tuple[ptr: f.ptr + 1, s: addlast(r, f.s)]
                                               else error(append-in-middle),
        reset(f) = if length(f.s) > 0  then tuple[ptr: 1, s: f.s]
                                       else error(file-empty),
        skip(f, n) =  if n < 0 then error(reverse-skip)
                      else if f.ptr + n > length(f.s) then error(skip-past-eof)
                      else tuple[ptr: f.ptr + n, s: f.s],
        read(f) = if f.ptr = 0  then error(file-empty)
                                else nth(f.ptr, f.s),
        eof(f) ≡ f.ptr = length(f.s),
end type.
```

appended to a file, but not deleted or updated. The files are sequential because they can only be scanned by starting at the beginning and spacing forward.

An abstract model specification has three major parts, describing the interface, the abstract representation, and the operations of the data type.

The **interface** of a data type consists of the names, domains, and ranges of its operations. This information is singled out because the operations provide the sole access to the abstract objects of the type. Thus a program, a proof, or even the rest of the specification can be checked for type correctness using only the information contained in the interface specifications of the data types that are used. (This is precisely the information that must be provided whenever abstractions are added to the CLU system, and the CLU compiler checks all uses and implementations of an abstraction for consistency with this information.)

The **abstract representation** is introduced into the specification solely to provide a framework in which to define the behavior of the operations of the type, and does *not* constrain the class of representations that may be used in the implementation. The types used in the abstract representation are chosen for simplicity rather than for efficiency. The primary use of specifications is for communication, and (perhaps) in proofs of program properties; how well they run as programs is of secondary interest. Therefore simplicity and clarity are important, while hypothetical time and space requirements are not.

The **abstract representation** has three subcomponents in its specification: the representation type, the abstract **invariant**, and the abstract **equivalence** relation. The representation type must be composed from previously defined types. We favor using finite sets, sequences, and tuples to put together known types into new ones. (Although we have not included them in this report, formal, axiomatic definitions of these families of types have been developed.)

Every meaningful abstract object should have a unique abstract representation, and conversely. The **invariant** describes a restriction on the representation type which excludes those elements that do not represent any meaningful abstract object. (It is similar in this respect to the invariant of the concrete representation [9] used in proving the correctness of an implementation of a data abstraction.) The **equivalence** is a relation stating which pairs of the representation type represent the same abstract object. If there are multiple meaningful representations for each abstract object, we can take the entire set (equivalence class) of elements representing an abstract object to be its unique abstract representation. The abstract equivalence is important because it specifies precisely which properties of the representation are being used to model the abstract type.

In the example, the state of a file is represented by a sequence of records, and a pointer into that sequence to indicate which record is currently being scanned. Note that the pointer is a natural number, which by definition cannot be negative, although it can be zero. The invariant says that the pointer can never get past the end of the sequence, and that provided the file is not empty, the pointer will always point at some record of the sequence (the first record has index 1). The equivalence tells us that each object of the representation type satisfying the invariant represents a unique file object.

The operations are defined as functions on the representation type, in as simple and clear a way as possible (efficiency does not matter). Any formal method for defining functions is acceptable. We will use both McCarthy's recursive conditional expressions [19], and input/output constraints expressed in the predicate calculus, as we find most convenient.

In the example, all of the operations except for *eof* are defined using conditional expressions, none of which need be recursive because of the simplicity of the data abstraction. *Eof* is defined as a predicate on the representation type, which happens not to require conditionals or quantifiers.

5.2 Consistency and Completeness of Abstract Model Specifications

A specification describes the behavior of some abstraction, and it is important that it describe that behavior correctly. While it is clearly not possible to *prove* that the specification is correct, it is possible, by analyzing properties of the specification, to identify problems, or alternatively to gain confidence in the correctness of the specification. Guttag [6] has done some work along these lines for algebraic specifications. We discuss below some criteria for abstract model specifications that we have developed for this purpose.

A well formed abstract model specification must satisfy the following requirements:

1. Type Correctness. The definitions of the operations must be consistent with the interface specifications, and all expressions of previously defined types must be consistent with the interface specifications of those types.

2. Representation consistency.
A. The invariant must be a well formed unary predicate on the representation type.
B. The equivalence must be a well formed binary predicate on the representation type, and it must define an equivalence relation (it must be reflexive, symmetric, and transitive).

3. Totality. Every operation mentioned in the interface specification must be uniquely defined for all elements of the representation type satisfying the invariant relation.

4. Closure. Every element in the intersection of the range of an operation with the representation type must satisfy the invariant relation.

5. Congruence. Every operation must be consistent with the representation equivalence, which means that equivalent inputs must result in equivalent outputs.

Some of these requirements are easier to check than others. The bulk of the type correctness check can be performed by a fairly simple algorithm, such as the one used by the CLU compiler. (Showing that no error values are produced, except for those described in the interface specifications, may require some program analysis.) At the other extreme, deciding whether a recursive function is total is undecidable in the general case, although there are well known techniques for proving termination, which apply to most programs that are designed to terminate [25]. A moderately powerful theorem proving facility is needed to demonstrate that all the requirements are met, comparable to the facility required for verifying that programs meet their specifications.

REFERENCES

[1] Atkinson, R. R. *Optimization Techniques for a Structured Programming Language.* S.M. Thesis, Dept. of Electrical Engineering and Computer Science, M. I. T., Cambridge, Mass., June 1976.

[2] Birkhoff, G. and Lipson, J. D. Heterogeneous algebras. *Journal of Combinatorial Theory 8* (1970), 115-133.

[3] Dahl, O. J., Myhrhaug, B., and Nygaard, K. *The SIMULA 67 Common Base Language.* Publication S-22, Norwegian Computing Center, Oslo, 1970.

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

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

[6] Guttag, J. V. *The Specification and Application to Programming of Abstract Data Types.* Ph. D. Thesis, University of Toronto Report CSRG-59, Toronto, Canada, 1975.

[7] Guttag, J. V. Abstract data types and the development of data structures. *Supplement to the Proceedings of the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure*, 1976, 37-46.

[8] Guttag, J. V., Horowitz, E. and Musser, D. R. *Abstract Data Types and Software Validation.* Report ISI/RR-76-48, University of Southern California, Los Angeles, Calif., 1976.

[9] Hoare, C. A. R. Proof of correctness of data representations. *Acta Informatica 1*, 4 (1972), 271-281.

[10] Jones, A. K. *Protection in Programming Systems.* Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburg, Pa., 1973

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

[12] Jones, A. K., and Liskov, B. H. *An Access Control Facility for Programming Languages.* Computation Structures Group Memo 137, Laboratory for Computer Science, M. I. T. , Cambridge, Mass., April 1976.

[13] King, J. C. *Symbolic Execution and Program Testing.* Report RC 5082, IBM Thomas J. Watson Research Center, Yorktown Heights, N. Y., October 1973.

[14] Lampson, B. W. Protection. *Proc. of the Fifth Annual Princeton Conference on Information*

Sciences and Systems, 1971, 437-443.

[15] Liskov, B. H., and Zilles, S. N. Specification techniques for data abstractions. *IEEE Trans. on Software Engineering, SE-1*, 1975, 7-19.

[16] Liskov, B. H., and Berzins, V. *An Appraisal of Program Specifications*. Computation Structures Group Memo 141, Laboratory for Computer Science, M. I. T., Cambridge, Mass., July 1976.

[17] Liskov, B. H., Snyder, A., Atkinson, R. R. and Schaffert, J. C. *Abstraction Mechanisms in CLU*. Computation Structures Group Memo 144, Laboratory for Computer Science, M. I. T., Cambridge, Mass., October 1976.

[18] McCarthy, J., et al. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.

[19] McCarthy, J. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, (Braffort, Hirschberg, Eds.), North Holland Publishing Co., Amsterdam-London 1963, 33-70.

[20] Parnas, D. L. A technique for software specification with examples. *Comm. of the ACM 15* (1972), 330-336.

[21] Parnas, D. L., and Handzel, G. *More on Specification Techniques for Software Modules.* Fachbereich Informatik Technische Hochschule Darmstadt, 1975.

[22] Robinson, L., Levitt, K., Neumann, P. G. and Saxena, A. R. On attaining reliable software for a secure operating system. *Proc. of the Interntional Conference on Reliable Software*, 1975, 267-2, 267-284.

[23] Scheifler, R. W. *An Analysis of Inline Substitution for the CLU Programming Language.* Computation Structures Group Memo 139, Laboratory for Computer Science, M. I. T., Cambridge, Mass., June 1976.

[24] Sturgis, H. E. *A Postmortem for a Time-Sharing System.* Ph. D. Thesis, University of California, Berkeley, Calif., 1974.

[25] Sites, R. L. *Proving That Computer Programs Terminate Cleanly.* Report STAN-CS-74-418, Stanford University, Computer Science Department, Stanford, Calif., 1974.

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

[27] Wulf, W., Johnsson, R. K., Weinstock, C. B., Hobbs, S. O., and Geschke, C. M. *The Design of an Optimizing Compiler.* American Elsevier Publishing Co., New York, 1975.

[28] Wulf, W. A., London, R., and Shaw, M. An introduction to the construction and verification

of Alphard programs. *IEEE Trans. on Software Engineering SE-2* (December 1976), 253-265.

[29] Zilles, S. *Algebraic Specification of Data Types.* Progress Report XI, Laboratory for Computer Science, M. I. T., Cambridge, Mass., 1974, 52-58.