LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Note on CLU

CSG Memo 112-1
November 1974
Revised June 1975

Barbara Liskov

This work was supported by the National Science Foundation under research grant GJ-34671.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Introduction

Our work on quality software concerns the development of tools and techniques which aid in the production of software that is correct, modifiable, understandable, and transferable. A major research effort of the previous year has been the design of CLU, a language system to support structured programming,by a group of students and faculty led by Professor Liskov. This work is motivated as follows: We believe that a significant improvement in software quality will result from the use of programming methodologies which aid the programmer in discovering correct and understandable problem solutions. Two promising methodologies have been described in the literature: structured programming [1] and modularity [2, 3].

Unfortunately, it is not easy to apply the methodologies to practical software problems. The methodologies are described by means of some "rules of thumb" illustrated by examples of "toy" problems. It is difficult to extrapolate from such descriptions to real situations, and conscientious attempts to apply the methodologies may fail [4]. In addition, even if the programmer succeeds in producing a design, the task of mapping the design into a program can be difficult and often introduces errors.

We believe the best approach to developing a methodology that will serve as a practical tool for program construction is through the design of a programming language such that problem solutions developed using the methodology are programs in the language. Several benefits accrue from this approach. First, since designs produced using the methodology are actual programs, the problems of mapping designs into programs do not require independent treatment. Second, completeness and precision of the language will be of a set of operations [5]. We developed a set of criteria about the way abstract data types should be handled:

1. A data type definition must include definitions of all operations applicable to objects of that type.
2. A user of an abstract data type need not know how objects of the type are represented in storage.
3. A user of an abstract data type may manipulate the objects only through the type's operations, and not through direct manipulation of the storage representation.

This last criterion insures that the operations provide a complete description of the behavior of the type, and enhances the modifiability and provability of programs.

Analysis of Existing Languages

J. Aiello studied a number of existing languages to determine whether they could meet the data abstraction needs of structured programming [6]. The criteria used were those described in the preceding paragraph. Programs were written in the selected languages to define selected data type examples.

The first language analyzed was PL/I. PL/I is described by IBM as an all-purpose language, but like many conventional languages (FORTRAN, ALGOL), fails to provide the user with a means for constructing data types. Analysis of PL/I revealed alternative mechanisms, for example, the multiple-entry procedure, which could be used for type definitions, and abstractions were programmed using these mechanisms. None of the programs satisfied all the criteria. Criterion 3 proved especially difficult, and attempts to satisfy it led to programs whose complexity greatly exceeddd that of the abstraction being programmed.

The next language, Pascal [7], was chosen because it provides numerous data structuring facilities. Pascal does support data type definitions, but

Since Simula came so close to satisfying our criteria, we investigated the possibility that some simple changes to the semantics of the language would result in acceptability. However, we found that this augmented SIMULA 67 failed in other respects (for example, representing operations that take more than one argument of abstract type).

As a result of this study, we concluded that existing languages, even with minor modifications, do not support abstract data types, and therefore that a new language must be developed for this purpose.

The Structured Programming Language, CLU

We are designing a structured programming language, called CLU, to permit the abstractions introduced during program design to be easily defined via CLU modules. Two kinds of modules are supported by CLU: procedures, which support abstract operations, and clusters, which support abstract data types in a way which satisfies the three criteria discussed earlier. An abstract data type is defined to be a set of operations. The cluster supporting the type contains definitions of the operations.

Important features of CLU are:

1. CLU is a modular programming language/system. Each CLU module is an independent entity and is accepted by the system by itself. The separateness of modules is a consequence of the view that modules support abstractions. The purpose of abstraction is to separate use from implementation; therefore, the implementation of an abstraction (a module) should be treated separately from its use (in another module or modules).

2. Although each module is developed and submitted to the system independently, a means must be provided to permit modules to refer to each other (so that one module can make use of the abstraction provided by another module). To provide this facility, the CLU system

Example of a definition of an abstract data type

An example of an abstract data type definition is presented to illustrate thos features of CLU which are most novel. The type to be defined is that of integer sets; a reasonable set of meaningful operations for integer sets are:

create	creates an empty set
insert	inserts an integer in a set
remove	removes an integer from a set
has	tests whether a set contains a particular integer
equal	tests whether two sets are the same
similar	tests whether two sets contain the same integers
copy	copies a set

Ordinary set behavior is desired: a set does not behave as if it contains multiple copies of the same integer.

A cluster implementing integer sets is shown in Figure 1. A cluster definition consists of three parts:

1. interface description
2. object description
3. operation definitions

The *interface description* of a cluster definition provides a very brief description of the interface which the cluster presents to its users. It consists of the name of the cluster and a list of the operations defining the type which the cluster implements: e.g.,

```
intset = cluster is create, insert remove, has, equal, similar, copy
```

The use of the reserved word *is* underlines the idea of a data type being equivalent to a group of operations; the group of operations following *is* is called the *is-list*.

```
        equal = oper(s, t: cvt) returns (boolean);
            return (rep $ equal(s, t));
        end equal;

        similar = oper(s, t: cvt) returns (boolean);
            if rep $ size(s) ~= rep $ size(t) then return (false);
            for i: int := rep $ low(s) to rep $ high(s) by 1 do
                if search(t, s[i]) > rep $ high(t) then return (false);
            return (true);
        end similar;

        copy = oper(s: cvt) returns (cvt);
            return (rep $ copy(s));
        end copy;

    end intset
```

Figure 1. The intset cluster (continued).

1. array limits. Each array has an upper and a lower bound and a size. All array elements between the bounds are defined (have values); no array elements are defined outside the bounds. Three operations give limit information:

    low(a)   returns the index of the lowest defined element, or the initially defined lower bound if the array is empty.

    high(a)   returns the index of the highest defined element, or low(a)-1 if the array is empty.

    size(a)   returns high(a) - low(a) + 1.

2. growing arrays. Arrays are empty when initially created. They may grow in either direction one element at a time:

    create(i)   returns a new empty array a with lower bound i. low(a) =
high(a) = i - 1, size(a) = 0.

    extendh(a, v)   a grows in the high direction by one element, and v is stored in that element. high(a) and size(a) increase by 1.

    extendl(a, v)   like extendh, but growth is in low direction.

3. accessing arrays. Arrays may be accessed and updated in the usual way, but only elements between the bounds may be referenced. The index is interpreted absolutely (not relative to low(a))

    fetch(a, i)   returns the value in the ith element of a if low(a) ≤ i ≤ high(a), else error. Syntactic "sugar" is provided: fetch(a, i) may be written a[i].

    store(a, i, v)   stores v in the ith element of a if low(a) ≤ i ≤ high(i), else error. Syntactic sugar is provided: store (a, i, v) may be written a[i] :== v

Figure 2. CLU arrays.

Some operations create new objects of the cluster type; create is an example of such an operation. The first thing create does is to bring into existence a variable r of the representing type

```
r: rep
```

It then initializes r to an object of the representing type; it creates the object by calling on a creating operation of that type:

```
rep $ create(0)
```

This line is an example of an operation call which requires a compound name to be used to specify the operation. The first part of the name identifies the type of the operation, while the second part identifies the operation. Since rep has been defined to be equal to array of int, the above operation call is the same as

```
array of int $ create(0)
```

Thus a call on the create operation for arrays is made.

Finally, create returns this object by the statement

```
return (r)
```

However, the type of r is the representing type, while the user of intset expects an object of type intset. Therefore, the create operation must cause the type of r to change before r is passed to the user of intset. The heading of the create operation specifies that this conversion is to occur:

```
create = oper ( ) returns (cvt)
```

This line states that the create operation expects no input parameters, and returns a single value. The use of the reserved word cvt states that this return value will be of the cluster type (intset in this case), and that the value being returned should be converted to the cluster

Uses of intset look very similar to the uses of array objects which appear in the intset operations. Variables may be declared of type intset:

```
s: intset
```

and intset objects created and assigned to such variables:

```
s := intset $ create( )
```

Operations may then be applied to intset objects:

```
intset $ insert(s, 3);
if intset $ has (s, 7) then intset $ remove(s, 7);
```

Also intset objects may be passed to procedures and to operations of other clusters. In every case, the CLU translator will check that the called procedure or operation expects an intset object in the position in which s occurs; any other expectation will cause a type error to be detected, and the translation will not complete. Therefore, it is impossible for any procedure or operation to receive an intset object as anything but an intset object.

Access to the rep of an abstract object can occur only within a cluster operation in which a parameter or result is marked by the indicator cvt. This indicator specifies that the argument or result is considered to be of the cluster's abstract type outside the body of the operation, but of type rep inside the operation body. Thus intset objects can only be accessed as objects of type rep inside the bodies of operations of the intset cluster.

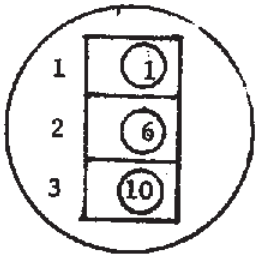
As was mentioned earlier, hiding an object's representation (criterion 3) is necessary to ensure that the behavior of the object is completely defined in terms of the type's operations. In addition, it is beneficial to software quality. Programs produced in this way are easy to modify: all changes to the implementation of a particular abstraction are guaranteed to be limited

Assignment and Parameter Passing

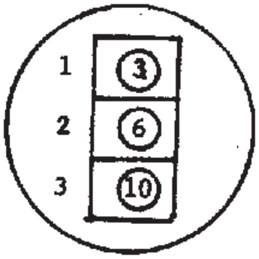
The semantics of CLU is based on a sharp distinction between variables and objects. CLU objects are the values which are manipulated by CLU programs. Each CLU object has a unique type associated with it. CLU objects may be simple, e.g., integer objects

⑤ ⑦

or complicated, e.g., an array containing integers 1, 6, 10 in elements 1, 2, 3



However, the complicatedness or simplicity of an object can't be observed directly; all that can be done with an object is to manipulate it using the operations defining the object's type. These operations provide the only means for making observations about objects of the type, and the operations completely define the behavior of the objects of the type. The objects of some types exhibit mutable behavior: some operations exist which will change the interior of an object without changing the object's identity. Array objects have mutable behavior; for example, the store operation, if asked to change the first element of the array above to ③ , will modify the array object itself, so that at the completion of the operation the object looks like



the procedure or operation heading are considered to be variables; thus

```
f = proc (s: array of int, i: int)
```

contains the declaration of two variables s and i. When a procedure or operation is invoked, the declarations take effect, and the variables are initialized by assigning the actual parameters to them. For example, if t is an array containing 3, 6, and 10 in elements 1, 2, 3 then

```
f(t, 2)
```

is a legal call of f; it causes variables s and i to be created, and assignments

```
s := t
i := 2
```

to be executed. The result of the call of f is illustrated in Figure 3a.

The reason that parameter passing in CLU is unusual is that assignment to the formal parameters of a procedure or operation does *not* affect the actual parameters. If x is an array containing 4 in element 1, and the assignment

```
s := x
```

occurs inside f, the result is that s now denotes a different object, but t is unaffected. Figure 3b illustrates the effect of s := x.

Because assignment to formal parameters inside of procedures cannot affect the actual parameters, CLU parameter passing is not *call by reference*, and one kind of side-effect is eliminated in CLU. We call our parameter passing *call by sharing*, because the object being passed is shared, as illustrated in Figure 3a. Information can be exchanged between calling and called procedures by changing the state of the shared object (if its type exhibits state behavior). Changing the state of an object received as input is the only kind of side effect a CLU procedure can have.

Equality

In addition to the primitive notion of assignment, a primitive notion of equality is often required in order to write meaningful programs. However, unlike assignment, which has a type-independent meaning and can be implemented automatically, equality has a type-dependent meaning. Therefore, it is not possible to provide an automatic implementation for equality. Instead, each cluster must include an equal operation (the operation which is named "equal") to provide an implementation of equality which is meaningful for the type being defined.

Although the meaning of equality is type-dependent, some general statements can be made about the meaning of equality which will help the cluster definer provide the correct definition of the equal operation. First, we can state what we expect equality to mean. Intuitively, two objects are equal if, at any time in the future, one can be substituted for the other without any resulting detectable difference in program behavior.

> Suppose T is a type, s1 and s2 are objects of type T, and o1, ..., on are operations of type T. If s1 has been determined to be equal to s2 by an application of the equal operation for T at time t, then at any time t' > t, any application of operation o1, ..., on to object s1 must provide "precisely the same results" as that operation applied to s2, where "precisely the same results" is measured by using the equal operation for the type of the resulting object.

In trying to apply the above criterion when defining a type, it is helpful to distinguish between constant and mutable types. For constant types, two objects are equal if the values inside them are equal insofar as the other operations of the type are able to distinguish. For example, two complex numbers are equal if their real and imaginary components are equal; two strings are equal if they contain the same characters in the same order.

For example, in the search operation of intset, the expression

```
i = s[j]
```

means

```
int $ equal(i, s[j])
```

Since the meaning of equality is so constrained for mutuable types, it is useful to have other concepts of equivalence supported by other cluster operations. One such definition is associated with the operation name "similar": two objects are similar if their contents are similar, insofar as the other operations of the type are able to distinguish. Thus, for a1 and a2 above,

```
array of int $ similar(a1, a2) = true
```

Another example is the similar operation of intset (Figure 1); two intset objects are similar if they contain the same integers, regardless of the order in which the integers are stored. Note that for both constant and mutable types, equality of objects implies similarity. The definer of a cluster has no obligation to provide a "similar" operation.

Copying

Often a user does not wish to have two variables share an object, or to share an object with a procedure he calls. Sharing of objects between two variables is dangerous because a change to the object through one of the variables affects the other variable. For example, starting from the situation in Figure 3b, i

```
s[1] := 5
```

is executed, the result is that x[1] will now return ⑤ . Copying objects is much safer than sharing because such anomolies don't arise. However, the meaning of copy is not defined by the CLU semantics; instead copy (like equal) is an operation which must be defined for each abstraction by giving an operation definition in the cluster. The reason for this is that the meaning of copy may be abstraction dependent; in fact, some abstractions may not even have a copy operation.

Type-generators differ from ordinary clusters in that they define a whole class of types, rather than a single type. Conventional programming languages contain one or more built-in type-generators. An example of such a type generator is the array. An array defines an access mechanism which is independent of the type of data which is stored in the array. Whenever an array is to be used, the program must specify what type of data the array is to contain; e.g.,

```
array of int
array of string
```

Type definitions like these can be viewed as selecting a particular array-type from the class of such types which the array type-generator defines.

CLU permits the programming of clusters which define type-generators rather than types. An example of the set type-generator is shown in Figure 4. The set cluster is very similar to the intset cluster shown in Figure 1. The two clusters differ only in that the set cluster makes use of a type parameter to define the type of element in the set, and everywhere the intset cluster used the type int to define the type of set element, the set cluster uses the type parameter.

The interface description for set identifies it as a type-generator by the presence of the cluster parameter

```
set = cluster[etype: type] is create, insert, remove, has, equal, copy
```

All clusters defining type-generators take one or more cluster parameters. The rep for set is now

```
rep = array of etype
```

The rep still makes use of the array type-generator, but it selects the particular array-type using the type parameter of the cluster.

In addition to appearing in the cluster interface definition and in the rep definition, the cluster parameter is also used to define the types of input and output parameters of operations; for example

```
insert = oper(s: cvt, i: etype)
```

Finally, the set cluster makes use of some of the etype operations. For example, in the search operation, the equal operation of etype is used:

```
etype $ equal(i, s[j])
```

A user-defined type-generator defines a whole class of types just like the built-in type-generator array does, and the rules for using type-generators are the same in either case. First it is necessary to state precisely what type is desired. This is done by using a type definition in which values are specified for the cluster parameters of the type generator; for example

```
intset = set[int]
newset = set[set[int]]
```

As with primitive type generators, such definitions can be viewed as selecting particular set-types from the class of types defined by the set type-generator.

Once a type has been defined, it can be used to declare variables and make operation calls, e.g.,

```
 s: intset :=  intset $ create( );
 t: intset := intset $ copy(s);
ss: newset :=  newset $ create( );
       .
       .
       .
newset $ insert(ss, s);
```