that it enforces the preservation of I_c. Step 4 guarantees (a) that the concrete operation is applicable whenever the abstract **pre** condition holds and (b) that if the operation is performed, the result corresponds properly to the abstract specifications.

Acknowledgments. We owe a great deal to our colleagues at Carnegie-Mellon University and the University of Southern California Information Sciences Institute, especially Mario Barbacci, Neil Goldman, Donald Good, John Guttag, Paul Hilfinger, David Jefferson, Anita Jones, David Lamb, David Musser, Karla Perdue, Kamesh Ramakrishna, and David Wile. We would also like to thank James Horning and Barbara Liskov and their groups at the University of Toronto and M.I.T., respectively, for their critical reviews of Alphard. We also appreciate very much the perceptive responses that a number of our colleagues have made on an earlier draft of this paper. Finally, we are grateful to Raymond Bates, David Lamb, Brian Reid, and Martin Yonke for their expert assistance with the document formatting programs.

References

1. Dahl, O.-J., and Hoare, C.A.R. Hierarchical program structures. In *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, New York, 1972, pp. 175–220.
2. Hoare, C.A.R. A note on the for statement. *BIT 12* (1972), 334–341.
3. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica 1*, 4 (1972), 271–281.
4. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language Pascal. *Acta Informatica* 2, 4 (1973), 335–355.
5. Igarashi, S., London, R.L., and Luckham, D.C. Automatic program verification I: a logical basis and its implementation. *Acta Informatica 4*, 2 (1975), 145–182.
6. Jensen, K., and Wirth, N. *PASCAL User Manual and Report.* Lecture Notes in Computer Science, No. 18, Springer-Verlag, 1974.
7. Katz, S., and Manna, Z. A closer look at termination. *Acta Informatica 5*, 4 (1975), 333–352.
8. London, R.L., Shaw, M., and Wulf, W.A. Abstraction and verification in Alphard: a symbol table example. Tech. Reports, Inform. Sci. Inst., U. of Southern California, Marina del Rey, Calif., and Carnegie-Mellon U., Pittsburgh, Pa., 1976.
9. Luckham, D.C., and Suzuki, N. Automatic program verification IV: Proof of termination within a weak logic of programs. Memo AIM-269, Stanford University, Stanford, Calif., Oct. 1975.
10. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. *LISP 1.5 Programmer's Manual.* MIT Press, 1962.
11. Newell, A., Tonge, F., Feigenbaum, E.A., Green, B.F. Jr., and Mealy, G.H. *Information Processing Language-V Manual.* Prentice-Hall, Englewood Cliffs, N.J., Sec. Ed. 1964.
12. Shaw, M. Abstraction and verification in Alphard: design and verification of a tree handler. Proc. Fifth Texas Conf. on Computing Systems, 1976, pp. 86–94.
13. Shaw, M., Wulf, W.A., and London, R.L. Abstraction and verification in Alphard: Iteration and generators. Tech. Reports, Inform. Sci. Inst., U. of Southern California, Marina del Rey, Calif., and Carnegie-Mellon U., Pittsburgh, Pa., 1976.
14. Teitelman, W. Interlisp Reference Manual. Xerox Palo Alto Res. Ctr., Palo Alto, Calif., 1975.
15. Weissman, C. *LISP 1.5 Primer,* Dickenson, Encino, Calif., 1967.
16. Wulf, W.A., London, R.L., and Shaw, M. Abstraction and verification in Alphard: Introduction to language and methodology. Tech. Reports, Inform. Sci. Inst., U. of Southern California, Marina del Rey, Calif., and Carnegie-Mellon U., Pittsburgh, Pa., 1976.
17. Wulf, W.A., London, R.L., and Shaw, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans. on Software Eng. SE-2*, 4 (Dec. 1976), 253–265.

Language Design for Reliable Software

S.L. Graham Editor

Abstraction Mechanisms in CLU

Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert
Massachusetts Institute of Technology

CLU is a new programming language designed to support the use of abstractions in program construction. Work in programming methodology has led to the realization that three kinds of abstractions – procedural, control, and especially data abstractions – are useful in the programming process. Of these, only the procedural abstraction is supported well by conventional languages, through the procedure or subroutine. CLU provides, in addition to procedures, novel linguistic mechanisms that support the use of data and control abstractions. This paper provides an introduction to the abstraction mechanisms in CLU. By means of programming examples, the utility of the three kinds of abstractions in program construction is illustrated, and it is shown how CLU programs may be written to use and implement abstractions. The CLU library, which permits incremental program development with complete type checking performed at compile time, is also discussed.

Key Words and Phrases: programming languages, data types, data abstractions, control abstractions, programming methodology, separate compilation
CR Categories: 4.0, 4.12, 4.20, 4.22

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.

A version of this paper was presented at the SIGPLAN/SIGOPS/SICSOFT Conference on Language Design for Reliable Software, Raleigh, N.C., March 28–30, 1977.

Authors' address: Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

1. Introduction

The motivation for the design of the CLU programming language was to provide programmers with a tool that would enhance their effectiveness in constructing programs of high quality—programs that are reliable and reasonably easy to understand, modify, and maintain. CLU aids programmers by providing constructs that support the use of abstractions in program design and implementation.

The quality of software depends primarily on the programming methodology in use. The choice of programming language, however, can have a major impact on the effectiveness of a methodology. A methodology can be easy or difficult to apply in a given language, depending on how well the language constructs match the structures that the methodology deems desirable. The presence of constructs that give a concrete form for the desired structures makes the methodology more understandable. In addition, a programming language influences the way that its users think about programming; matching a language to a methodology increases the likelihood that the methodology will be used.

CLU has been designed to support a methodology (similar to [6, 22]) in which programs are developed by means of problem decomposition based on the recognition of abstractions. 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). The implementation is developed by envisioning a number of subsidiary abstractions (abstract objects and operations) that are useful in the problem domain. Once the behavior of the abstract objects and operations has been defined, a program can be written to solve the original problem; in this program, the abstract objects and operations are used as primitives. Now the original problem has been solved, but new problems have arisen, namely, how to implement the subsidiary abstractions. Each of these abstractions is considered in turn as a new problem; its implementation may introduce further abstractions. This process terminates when all the abstractions introduced at various stages have been implemented or are present in the programming language in use.

In this methodology, programs are developed incrementally, one abstraction at a time. Further, a distinction is made between an abstraction, which is a kind of behavior, and a program, or *module*, which implements that behavior. An abstraction isolates use from implementation: an abstraction can be used without knowledge of its implementation and implemented without knowledge of its use. These aspects of the methodology are supported by the CLU *library*, which maintains information about abstractions and the CLU modules that implement them. The library permits separate compilation of modules with complete type checking at compile time.

To make effective use of the methodology, it is necessary to understand the kinds of abstractions that are useful in constructing programs. In studying this question, we identified an important kind of abstraction, the data abstraction, that had been largely neglected in discussions of programming methodology.

A data abstraction [8, 12, 20] 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 includes 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 characterizing the behavior of the objects.

If a data abstraction is to be understandable at an abstract level, the behavior of the data objects must be *completely* characterized by the set of operations. This property is ensured by making the operations the *only direct means* of creating and manipulating the objects. One effect of this restriction is that, when defining an abstraction, the programmer must be careful to include a sufficient set of operations, since every action he wishes to perform on the objects must be realized in terms of this set.

We have identified the following requirements that must be satisfied by a language supporting data abstractions:

1. A linguistic construct is needed that permits a data abstraction to be implemented as a unit. The implementation involves selecting a representation for the data objects and defining an algorithm for each operation in terms of that representation.

2. The language must limit access to the representation to just the operations. This limitation is necessary to ensure that the operations completely characterize the behavior of the objects.

CLU satisfies these requirements by providing a linguistic construct called a *cluster* for implementing data abstractions. Data abstractions are integrated into the language through the data type mechanism. Access to the representation is controlled by type checking, which is done at compile time.

In addition to data abstractions, CLU supports two other kinds of abstractions: procedural abstractions and control abstractions. A procedural abstraction per-

forms a computation on a set of input objects and produces a set of output objects; examples of procedural abstractions are sorting an array and computing a square root. CLU supports procedural abstractions by means of procedures, which are similar to procedures in other programming languages.

A control abstraction defines a method for sequencing arbitrary actions. All languages provide built-in control abstractions; examples are the **if** statement and the **while** statement. In addition, however, CLU allows user definitions of a simple kind of control abstraction. The method provided is a generalization of the repetition methods available in many programming languages. Frequently the programmer desires to perform the same action for all the objects in a collection, such as all characters in a string or all items in a set. CLU provides a linguistic construct called an *iterator* for defining how the objects in the collection are obtained. The iterator is used in conjunction with the **for** statement; the body of the **for** statement describes the action to be taken.

The purpose of this paper is to illustrate the utility of the three kinds of abstractions in program construction and to provide an informal introduction to CLU. We do not attempt a complete description of the language; rather, we concentrate on the constructs that support abstractions. The presence of these constructs constitutes the most important way in which CLU differs from other languages. The language closest to CLU is Alphard [24], which represents a concurrent design effort with goals similar to our own. The design of CLU has been influenced by Simula 67 [4] and to a lesser extent by Pascal [23] and Lisp [15].

In the next section we introduce CLU and, by means of a programming example, illustrate the use and implementation of data abstractions. Section 3 describes the basic semantics of CLU. In Section 4, we discuss control abstractions and more powerful kinds of data abstractions. We present the CLU library in Section 5. Section 6 briefly describes the current implementation of CLU and discusses efficiency considerations. Finally, we conclude by discussing the quality of CLU programs.

2. An Example of Data Abstraction

This section introduces the basic data abstraction mechanism of CLU, the cluster. By means of an example, we intend to show how abstractions occur naturally in program design and how they are used and implemented in CLU. In particular, we show how a data abstraction can be used as structured intermediate storage.

Consider the following problem: given some document, we wish to compute, for each distinct word in the document, the number of times the word occurs and its frequency of occurrence as a percentage of the total number of words. The document will be represented as a sequence of characters. A word is any nonempty sequence of alphabetic characters. Adjacent words are separated by one or more nonalphabetic characters such as spaces, punctuation, or newline characters. In recognizing distinct words, the difference between upper and lower case letters should be ignored.

The output is also to be a sequence of characters, divided into lines. Successive lines should contain an alphabetical list of all the distinct words in the document, one word per line. Accompanying each word should be the total number of occurrences and the frequency of occurrence. For example:

a	2	3.509%
access	1	1.754%
and	2	3.509%

Specifically, we are required to write the procedure *count_words*, which takes two arguments: an *instream* and an *outstream*. The former is the source of the document to be processed, and the latter is the destination of the required output. The form of this procedure will be

```
count_words = proc (i: instream, o: outstream);
  . . .
  end count_words;
```

Note that *count_words* does not return any results; its only effects are modifications of *i* (reading the entire document) and of *o* (printing the required statistics).

Instream and *outstream* are data abstractions. An *instream i* contains a sequence of characters. Of the primitive operations on *instreams*, only two will be of interest to us. *Empty* (*i*) returns **true** if there are no characters available in *i* and returns **false** otherwise. *Next* (*i*) removes the first character from the sequence and returns it. Invoking the *next* operation on an empty *instream* is an error.[1] An *outstream* also contains a sequence of characters. The interesting operation on *outstreams* is *put_string* (*s*, *o*), which appends the string *s* to the existing sequence of characters in *o*.

Now consider how we might implement *count_words*. We begin by deciding how to handle words. We could define a new abstract data type *word*. However, we choose instead to use strings (a primitive CLU type), with the restriction that only strings of lower case alphabetic characters will be used.[2]

Next we investigate how to scan the document. Reading a word requires knowledge of the exact way in which words occur in the input stream. We choose to isolate this information in a procedural abstraction, called *next_word*, which takes in the *instream i* and returns the next word (converted to lower case charac-

[1] The CLU error handling mechanism is discussed in [10].

[2] Sometimes it is difficult to decide whether to introduce a new data abstraction or to use an existing abstraction. Our decision to use strings to represent words was made partly to shorten the presentation.

ters) in the document. If there are no more words, *next_word* must communicate this fact to *count_words*. A simple way to indicate that there are no more words is by returning an "end of document" word, one that is distinct from any other word. A reasonable choice for the "end of document" word is the empty string.

It is clear that in *count_words* we must scan the entire document before we can print our results, and therefore we need some receptacle to retain information about words between these two actions (scanning and printing). Recording the information gained in the scan and organizing it for easy printing will probably be fairly complex. Therefore we defer such considerations until later by introducing a data abstraction *wordbag* with the appropriate properties. In particular, *wordbag* provides three operations: *create*, which creates an empty *wordbag*; *insert*, which adds a word to the *wordbag*; and *print*, which prints the desired statistical information about the words in the *wordbag*.[3]

The implementation of *count_words* is shown in Figure 1. The "%" character starts a comment, which continues to the end of the line. The "~" character stands for boolean negation. The notation *variable*: *type* is used in formal argument lists and declarations to specify the types of variables; a declaration may be combined with an assignment specifying the initial value of the variable. Boldface is used for reserved words, including the names of primitive CLU types.

The *count_words* procedure declares four variables: *i*, *o*, *wb*, and *w*. The first two denote the *instream* and *outstream* that are passed as arguments to *count_words*. The third, *wb*, denotes the *wordbag* used to hold the words read so far, and the fourth, *w*, the word currently being processed.

Operations of a data abstraction are named by a compound form that specifies both the type and the operation name. Three examples of operation calls appear in *count_words*: *wordbag$create*( ), *wordbag$insert (wb, w)* and *wordbag$print (wb, o)*. The CLU system provides a mechanism that avoids conflicts between names of abstractions; this mechanism is discussed in Section 5. However, operations of two different data abstractions may have the same name; the compound form serves to resolve this ambiguity. Although the ambiguity could in most cases be resolved by context, we have found in using CLU that the compound form enhances the readability of programs.

The implementation of *next_word* is shown in Figure 2. The *string$append* operation creates a new string by appending a character to the characters in the string argument (it does *not* modify the string argument). Note the use of the *instream* operations *next* and *empty*. Note also that two additional procedures have been used: *alpha* (*c*), which tests whether a character is alphabetic or not, and *lower_case* (*c*), which returns the lower case version of a character. The implementations

[3] The *print* operation is not the ideal choice, but a better solution requires the use of control abstractions. This solution is presented in Section 4.

Fig. 1. The *count_words* procedure.

```
count_words = proc (i: instream, o: outstream);
  % create an empty wordbag
  wb: wordbag := wordbag$create ( );
  % scan document, adding each word found to wb
  w: string := next_word (i);
  while w ~= " " do
    wordbag$insert (wb, w);
    w := next_word (i);
    end;
  % print the wordbag
  wordbag$print (wb, o);
  end count_words;
```

Fig. 2. The *next_word* procedure.

```
next_word = proc (i: instream) returns (string);
  c: char := '';
  % scan for first alphabetic character
  while ~alpha (c) do
    if instream$empty (i)
      then return " ";
      end;
    c := instream$next (i);
    end;
  % accumulate characters in word
  w: string := " ";
  while alpha (c) do
    w := string$append (w, c);
    if instream$empty (i)
      then return (w);
      end;
    c := instream$next (i);
    end;
  return (w);      % the nonalphabetic character c is lost
  end next_word;
```

of these procedures are not shown in the paper.

Now we must implement the type *wordbag*. The cluster will have the form

```
wordbag = cluster is create, insert, print;
  . . .
  end wordbag;
```

This form expresses the idea that the data abstraction is a set of operations as well as a set of objects. The cluster must provide a representation for objects of the type *wordbag* and an implementation for each of the operations. We are free to choose from the possible representations the one best suited to our use of the *wordbag* cluster.

The representation that we choose should allow reasonably efficient storage of words and easy printing, in alphabetic order, of the words and associated statistics. For efficiency in computing the statistics, maintaining a count of the total number of words in the document would be helpful. Since the total number of words in the document is probably much larger than the number of distinct words, the representation of a *wordbag* should contain only one "item" for each distinct word (along with a multiplicity count), rather than one "item" for each occurrence. This choice of representation requires that, at each insertion, we check whether

the new word is already present in the *wordbag*. We would like a representation that allows the search for a matching "item" and the insertion of a not previously present "item" to be efficient. A binary tree representation [9] fits our requirements nicely.

Thus the main part of the *wordbag* representation consists of a binary tree. The binary tree is another data abstraction, *wordtree*. The data abstraction *wordtree* provides operations very similar to those of *wordbag*: *create* ( ) returns an empty *wordtree*; *insert* (*tr*, *w*) returns a *wordtree* containing all the words in the *wordtree tr* plus the additional word *w* (the *wordtree tr* may be modified in the process); and *print* (*tr*, *n*, *o*) prints the contents of the *wordtree tr* in alphabetic order on *outstream o* along with the number of occurrences and the frequency (based on a total of *n* words).

The implementation of *wordbag* is given in Figure 3. Following the header, we find the definition of the representation selected for *wordbag* objects:

rep = **record** [contents: wordtree, total: **int**];

The reserved type identifier **rep** indicates that the type specification to the right of the equal sign is the representing type for the cluster. We have defined the representation of a *wordbag* object to consist of two pieces: a *wordtree*, as explained above, and an integer, which records the total number of words in the *wordbag*.

A CLU record is an object with one or more named components. For each component name, there is an operation to select and an operation to set the corresponding component. The operation *get_n* (*r*) returns the *n* component of the record *r* (this operation is usually abbreviated *r.n*). The operation *put_n* (*r*, *x*) makes *x* the *n* component of the record *r* (this operation is usually abbreviated *r.n* := *x*, by analogy with the assignment statement). A new record is created by an expression of the form type${name$_1$: value$_1$, . . .}.

There are two different types associated with any cluster: the abstract type being defined (*wordbag* in this case) and the representation type (the record). Outside of the cluster, type checking ensures that a *wordbag* object will always be treated as such. In particular, the ability to convert a *wordbag* object into its representation is not provided (unless one of the *wordbag* operations does so explicitly).

Inside the cluster, however, it is necessary to view a *wordbag* object as being of the representation type, because the implementations of the operations are defined in terms of the representation. This change of viewpoint is signalled by having the reserved word **cvt** appear as the type of an argument (as in the *insert* and *print* operations). **Cvt** may also appear as a return type (as in the *create* operation); here it indicates that a returned object will be changed into an object of abstract type. Whether **cvt** appears as the type of an argument or as a return type, it stipulates a "conversion" of viewpoint between the external abstract type and the internal representation type. **Cvt** can be used only within a cluster, and conversion can be done only between the single abstract type being defined and the (single) representation type.[4]

Fig. 3. The *wordbag* cluster.

```
wordbag = cluster is
   create,     % create an empty bag
   insert,     % insert an element
   print;      % print contents of bag
   rep = record [contents: wordtree, total: int];
create = proc ( ) returns (cvt);
         return (rep${contents: wordtree$create ( ), total: 0});
         end create;
insert = proc (x: cvt, v: string);
         x.contents := wordtree$insert (x.contents, v);
         x.total := x.total + 1;
         end insert;
print = proc (x: cvt, o: outstream);
        wordtree$print (x.contents, x.total, o);
        end print;
end wordbag;
```

The procedures in *wordbag* are very simple. *Create* builds a new instance of the **rep** by use of the record constructor

rep${contents: wordtree$create ( ), total: 0}

Here *total* is initialized to 0 and *contents* to the empty *wordtree* (by calling the *create* operation of *wordtree*). This **rep** object is converted into a *wordbag* object as it is being returned. *Insert* and *print* are implemented directly in terms of *wordtree* operations.

The implementation of *wordtree* is shown in Figure 4. In the *wordtree* representation, each node contains a word and the number of times that word has been inserted into the *wordbag*, as well as two subtrees. For any particular node, the words in the "lesser" subtree must alphabetically precede the word in the node, and the words in the "greater" subtree must follow the word in the node. This information is described by

node = **record** [value: **string**, count: **int**,
lesser: wordtree, greater: wordtree];

which defines "node" to be an abbreviation for the information following the equal sign. (The reserved word **rep** is used similarly as an abbreviation for the representation type.)

Now consider the representation of *wordtrees*. A nonempty *wordtree* can be represented by its top node. An empty *wordtree*, however, contains no information. The ideal type to represent an empty *wordtree* is the CLU type **null**, which has a single data object **nil**. So the representation of a *wordtree* should be either a node or **nil**. This representation is expressed by

rep = **oneof** [empty: **null**, non_empty: node];

Just as the record is the basic CLU mechanism to

[4] **Cvt** corresponds to Morris' seal and unseal [16] except that **cvt** represents a change in viewpoint only; no computation is required.

form an object that is a collection of other objects, the oneof is the basic CLU mechanism to form an object that is "one of" a set of alternatives. Oneof is CLU's method of forming a discriminated union, and is somewhat similar to a variant component of a record in Pascal [23].

An object of the type **oneof** [s_1: T_1 . . . s_n: T_n] can be thought of as a pair. The "tag" component is an identifier from the set {s_1 . . . s_n}. The "value" component is an object of the type corresponding to the tag. That is, if the tag component is s_i, then the value is some object of type T_i.

Objects of type **oneof** [s_1: T_1 . . . s_n: T_n] are created by the operations *make_*$s_i(x)$, each of which takes an object x of type T_i and returns the pair $\langle s_i, x \rangle$. Because the type of the value component of a oneof object is not known at compile time, allowing direct access to the value component could result in a run-time type error (e.g. assigning an object to a variable of the wrong type). To eliminate this possibility, we require the use of a special **tagcase** statement to decompose a oneof object:

```
tagcase e
  tag s1 (id1: T1):   statements . . .
      . . .
  tag sn (idn: Tn):   statements . . .
  end;
```

This statement evaluates the expression e to obtain an object of type **oneof** [s_1: T_1 . . . s_n: T_n]. If the tag is s_i, then the value is assigned to the new variable id_i and the statements following the ith alternative are executed. The variable id_i is local to those statements. If, for some reason, we do not need the value, we can omit the parenthesized variable declaration.

The reader should now know enough to understand Figure 4. Note, in the *create* operation, the use of the construction operation *make_empty* of the representation type of *wordtree* (the discriminated union **oneof** [empty: **null**, nonempty: node]) to create the empty *wordtree*. The **tagcase** statement is used in both *insert* and *print*. Note that if *insert* is given an empty *wordtree*, it creates a new top node for the returned value, but if *insert* is given a nonempty *wordtree*, it modifies the given *wordtree* and returns it.[5] The *insert* operation depends on the dynamic allocation of space for newly created records (see Section 3).

The *print* operation uses the obvious recursive descent. It makes use of procedure *print_word* (w, c, t, o), which generates a single line of output on o consisting of the word w, the count c, and the frequency of occurrence derived from c and t. The implementation of *print_word* has been omitted.

We have now completed our first discussion of the *court_words* procedure. We return to this problem in Section 4, where we present a superior solution.

[5] It is necessary for *insert* to return a value in addition having a side effect because in the case of an empty *wordtree* argument side effects are not possible. Side effects are not possible because of the representation chosen for the empty *wordtree* and because of the CLU parameter passing mechanism (see Section 3).

Fig. 4. The *wordtree* cluster.

```
wordtree = cluster is
  create,     % create empty contents
  insert,     % add item to contents
  print;      % print contents
  node = record [value: string, count: int,
                 lesser: wordtree, greater: wordtree];
  rep = oneof [empty: null, non_empty: node];
create = proc ( ) returns (cvt);
  return (rep$make_empty (nil));
  end create;
insert = proc (x: cvt, v: string) returns (cvt);
  tagcase x
    tag empty:
        n: node := node${value: v, count: 1,
                        lesser: wordtree$create ( ),
                        greater: wordtree$create ( )};
        return (rep$make_non_empty (n));
    tag non_empty (n: node):
        if v = n.value
            then n.count := n.count + 1;
          elseif v < n.value
            then n.lesser := wordtree$insert (n.lesser, v);
          else n.greater := wordtree$insert (n.greater, v);
          end;
        return (x);
    end;
  end insert;
print = proc (x: cvt, total: int, o: outstream);
  tagcase x
    tag empty: ;
    tag non_empty (n: node):
      wordtree$print (n.lesser, total, o);
      print_word (n.value, n. count, total, o);
      wordtree$print (n.greater, total, o);
    end;
  end print;
end wordtree;
```

3. Semantics

All languages present their users with some model of computation. This section describes those aspects of CLU semantics that differ from the common Algol-like model. In particular, we discuss CLU's notions of objects and variables and the definitions of assignment and argument passing that follow from these notions. We also discuss type correctness.

3.1 Objects and Variables

The basic elements of CLU semantics are *objects* and *variables*. Objects are the data entities that are created and manipulated by CLU programs. Variables are just the names used in a program to refer to objects.

In CLU, each object has a particular *type*, which characterizes its behavior. A type defines a set of operations that create and manipulate objects of that type. An object may be created and manipulated only via the operations of its type.

An object may *refer* to objects. For example, a

record object refers to the objects that are the components of the record. This notion is one of logical, not physical, containment. In particular, it is possible for two distinct record objects to refer to (or *share*) the same component object. In the case of a cyclic structure, it is even possible for an object to "contain" itself. Thus it is possible to have recursive data structure definitions and shared data objects without explicit reference types. The *wordtree* type described in the previous section is an example of a recursively defined data structure. (This notion of object is similar to that in Lisp.)

CLU objects exist independently of procedure activations. Space for objects is allocated from a dynamic storage area as the result of invoking constructor operations of certain primitive CLU types. For example, the record constructor is used in the implementation of *wordbag* (Figure 3) to acquire space for new *wordbag* objects. In theory, all objects continue to exist forever. In practice, the space used by an object may be reclaimed when the object is no longer accessible to any CLU program.[6]

An object may exhibit time-varying behavior. Such an object, called a *mutable* object, has a state which may be modified by certain operations without changing the identity of the object. Records are examples of mutable objects. The record update operations (*put_s* (r, v), written as $r.s := v$ in the examples), change the state of record objects and therefore affect the behavior of subsequent applications of the select operations (*get_s* (r), written as $r.s$). The *wordbag* and *wordtree* types are additional examples of types with mutable objects.

If a mutable object m is shared by two other objects x and y, then a modification to m made via x will be visible when m is examined via y. Communication through shared mutable objects is most beneficial in the context of procedure invocation, described below.

Objects that do not exhibit time-varying behavior are called *immutable* objects, or *constants*. Examples of constants are integers, booleans, characters, and strings. The value of a constant object can not be modified. For example, new strings may be computed from old ones, but existing strings do not change. Similarly, none of the integer operations modify the integers passed to them as arguments.

Variables are names used in CLU programs to *denote* particular objects at execution time. Unlike variables in many common programming languages, which *are* objects that *contain* values, CLU variables are simply names that the programmer uses to refer to objects. As such, it is possible for two variables to denote (or *share*) the same object. CLU variables are much like those in Lisp and are similar to pointer variables in other languages. However, CLU variables are *not* objects; they cannot be denoted by other variables or referred to by objects. Thus variables are completely private to the procedure in which they are declared and cannot be accessed or modified by any other procedure.

[6] An object is accessible if it is denoted by a variable of an active procedure or is a component of an accessible object.

3.2 Assignment and Procedure Invocation

The basic actions in CLU are *assignment* and *procedure invocation*. The assignment primitive $x := E$, where x is a variable and E is an expression, causes x to denote the object resulting from the evaluation of E. For example, if E is a simple variable y, then the assignment $x := y$ causes x to denote the object denoted by y. The object is *not* copied; after the assignment is performed, it will be *shared* by x and y. Assignment does not affect the state of any object. (Recall that $r.s := v$ is not a true assignment, but an abbreviation for *put_s* (r, v).)

Procedure invocation involves passing argument objects from the caller to the called procedure and returning result objects from the procedure to the caller. The formal arguments of a procedure are considered to be local variables of the procedure and are initialized, by assignment, to the objects resulting from the evaluation of the argument expressions. Thus argument objects are shared between the caller and the called procedure. A procedure may modify mutable argument objects (e.g. records), but of course it cannot modify immutable ones (e.g. integers). A procedure has no access to the variables of its caller.

Procedure invocations may be used directly as statements; those that return objects may also be used as expressions. Arbitrary recursive procedures are permitted.

3.3 Type Correctness

Every variable in a CLU module must be declared; the declaration specifies the type of object that the variable may denote. All assignments to a variable must satisfy the variable's declaration. Because argument passing is defined in terms of assignment, the types of actual argument objects must be consistent with the declarations of the corresponding formal arguments.

These restrictions, plus the restriction that only the code in a cluster may use **cvt** to convert between the abstract and representation types, ensure that the behavior of an object is indeed characterized completely by the operations of its type. For example, the type restrictions ensure that the only modification possible to a record object that represents a *wordbag* (Figure 3) is the modification performed by the *insert* operation.

Type checking is performed on a module by module basis at compile time (it could also be done at run time). This checking can catch all type errors—even those involving intermodule references—because the CLU library maintains the necessary type information for all modules (see Section 5).

4. More Abstraction Mechanisms

In this section we continue our discussion of abstraction mechanisms in CLU. A generalization of the *wordbag* abstraction, called *sorted_bag*, is presented as an illustration of parameterized clusters, which are a means for implementing more generally applicable data abstractions. The presentation of *sorted_bag* is also used to motivate the introduction of a control abstraction called an *iterator*, which is a mechanism for incrementally generating the elements of a collection of objects. Finally, we show an implementation of the *sorted_bag* abstraction and illustrate how *sorted_bag* can be used in implementing *count_words*.

4.1 Properties of the Sorted_bag Abstraction

In the *count_words* procedure given earlier, a data abstraction called *wordbag* was used. A *wordbag* object is a collection of strings, each with an associated count. Strings are inserted into a *wordbag* object one at a time. Strings in a *wordbag* object may be printed in alphabetical order, each with a count of the number of times it was inserted.

Although *wordbag* has properties that are specific to the usage in *count_words*, it also has properties in common with a more general abstraction, *sorted_bag*. A bag is similar to a set (it is sometimes called a multiset) except that an item can appear in a bag many times. For example, if the integer 1 is inserted in the set {1, 2}, the result is the set {1, 2}, but if 1 is inserted in the bag {1, 2}, the result is the bag {1, 1, 2}. A *sorted_bag* is a bag that affords access to the items it contains according to an ordering relation on the items.

The concept of a *sorted_bag* is meaningful not only for strings but for many types of items. Therefore we would like to parameterize the *sorted_bag* abstraction, the parameter being the type of item to be collected in the *sorted_bag* objects.

Most programming languages provide built-in parameterized data abstractions. For example, the concept of an array is a parameterized data abstraction. An example of a use of arrays in Pascal is

array 1..n **of integer**

These arrays have two parameters, one specifying the array bounds (1..n) and one specifying the type of element in the array (integer). In CLU we provide mechanisms allowing user-defined data abstractions (like *sorted_bag*) to be parameterized.

In the *sorted_bag* abstraction, not all types of items make sense. Only types that define a total ordering on their objects are meaningful since the *sorted_bag* abstraction depends on the presence of this ordering. In addition, information about the ordering must be expressed in a way that is useful for programming. A natural way to express this information is by means of operations of the item type. Therefore we require that the item type provide less than and equal operations (called *lt* and *equal*). This constraint is expressed in the header for *sorted_bag*:

```
sorted_bag = cluster [t: type] is create, insert, . . .
  where t has
    lt, equal: proctype (t, t) returns (bool);
```

The item type *t* is a *formal parameter* of the *sorted_bag* cluster; whenever the *sorted_bag* abstraction is used, the item type must be specified as an *actual parameter*, e.g.

```
sorted_bag[string]
```

The information about required operations informs the programmer about legitimate uses of *sorted_bag*. The compiler will check each use of *sorted_bag* to ensure that the item type provides the required operations. The **where** clause specifies exactly the information that the compiler can check. Of course, more is assumed about the item type *t* than the presence of operations with appropriate names and functionalities: these operations must also define a total ordering on the items. Although we expect formal and complete specifications for data abstractions to be included in the CLU library eventually, we do not include in the CLU language declarations that the compiler cannot check. This point is discussed further in Section 7.

Now that we have decided to define a *sorted_bag* abstraction that works for many item types, we must decide what operations this abstraction provides. When an abstraction (like *wordbag*) is written for a very specific purpose, it is reasonable to have some specialized operations. For a more general abstraction, the operations should be more generally useful.

The *print* operation is a case in point. Printing is only one possible use of the information contained in a *sorted_bag*. It was the only use in the case of *wordbag*, so it was reasonable to have a *print* operation. However, if *sorted_bags* are to be generally useful, there should be some way for the user to obtain the elements of the *sorted_bag*; the user can then perform some action on the elements (for example, print them).

What we would like is an operation on *sorted_bags* that makes all of the elements available to the caller in increasing order. One possible approach is to map the elements of a *sorted_bag* into a sequence object, a solution potentially requiring a large amount of space. A more efficient method is provided by CLU and is discussed below. This solution computes the sequence one element at a time, thus saving space. If only part of the sequence is used (as in a search for some element), then execution time can be saved as well.

4.2 Control Abstractions

The purpose of many loops is to perform an action on some or 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.

Fig. 5. 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 = iter (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;
```

CLU provides a control abstraction 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 iterates over ranges of integers. The CLU **for** statement can iterate 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); each object is consumed by the **for** statement in turn.

Figure 5 gives an example of a simple iterator called *string_chars*, which produces the characters in a string in the order in which they appear. This iterator uses string operations *size*(s), which tells how many characters are in the string s, and *fetch* (s, n), which returns the nth character in the string s (provided the integer n is greater than zero and does not exceed the size of the string).[7]

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 5), 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[8] are assigned to the variables declared in the **for** statement (following the reserved word **for**) in corresponding order, and the body of the **for** statement is executed. Then 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, by either an implicit or explicit **return**, the invoking **for** statement terminates. The iteration may also be prematurely terminated by a **return** in the body of the **for** statement.

For example, suppose that *string_chars* is invoked 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, *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. The implicit **return** terminates both the iterator and 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, arrays, and *sorted_bags*). Iterators afford users of such abstractions access to all objects in the collection without exposing irrelevant details. Several iterators may be included in a data abstraction. When the order of obtaining the objects is important, different iterators may provide different orders.

4.3 Implementation and Use of Sorted_bag

Now we can describe a minimal set of operations for *sorted_bag*. The operations are *create*, *insert*, *size*, and *increasing*. *Create*, *insert*, and *size* are procedural abstractions that, respectively, create a *sorted_bag*, insert an item into a *sorted_bag*, and give the number of items in a *sorted_bag*. *Increasing* is a control abstraction that produces the items in a *sorted_bag* in increasing order; each item produced is accompanied by an integer representing the number of times the item appears in the *sorted_bag*. Note that other operations might also be useful for *sorted_bag*, for example, an iterator yielding the items in decreasing order. In general, the definer of a data abstraction can provide as many operations as seems reasonable.

In Figure 6, we give an implementation of the *sorted_bag* abstraction. It is implemented by using a sorted binary tree, just as *wordbag* was implemented. Thus a subsidiary abstraction is necessary. This abstraction, called *tree*, is a generalization of the *wordtree* abstraction (used in Section 2), which has been parameterized to work for all ordered types. An implementation of *tree* is given in Figure 7. Notice that both the *tree* abstraction and the *sorted_bag* abstraction place the same constraints on their type parameters.

[7] A **while** loop is used in the implementation of *string_chars* so that the example will be based on familiar concepts. In actual practice, such a loop would be written by using a **for** statement invoking a primitive iterator.

[8] Zero 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.

Fig. 6. The *sorted_bag* cluster.

```
sorted_bag = cluster [t: type] is create, insert, size, increasing
  where t has equal, lt: proctype (t, t) returns (bool);
  rep = record [contents: tree[t], total: int];
create = proc ( ) returns (cvt);
  return (rep${contents: tree[t]$create ( ), total: 0});
  end create;
insert = proc (sb: cvt, v: t);
  sb.contents := tree[t]$insert (sb.contents, v);
  sb.total := sb.total + 1;
  end insert;
size = proc (sb: cvt) returns (int);
  return (sb.total);
  end size;
increasing = iter (sb: cvt) yields (t, int);
  for item: t, count: int
    in tree[t]$increasing (sb.contents) do
      yield (item, count);
      end;
  end increasing;
end sorted_bag;
```

Fig. 7. The *tree* cluster.

```
tree = cluster [t: type] is create, insert, increasing
  where t has equal, lt: proctype (t, t) returns (bool);
  node = record [value: t, count: int,
                 lesser: tree[t], greater: tree[t]];
  rep = oneof [empty: null, non_empty: node];
create = proc ( ) returns (cvt);
  return (rep$make_empty (nil));
  end create;
insert = proc (x: cvt, v: t) returns (cvt);
  tagcase x
    tag empty:
      n: node := node${value: v, count: 1,
                      lesser: tree[t]$create ( ),
                      greater: tree[t]$create ( )};
      return (rep$make_non_empty (n));
    tag non_empty (n: node):
      if t$equal (v, n.value)
          then n.count := n.count + 1;
        elseif t$lt (v, n.value)
          then n.lesser := tree[t]$insert (n.lesser, v);
        else n.greater := tree[t]$insert (n.greater, v);
        end;
      return (x);
    end;
  end insert;
increasing = iter (x: cvt) yields (t, int);
  tagcase x
    tag empty: ;
    tag non_empty (n: node):
      for item: t, count: int
        in tree[t]$increasing (n.lesser) do
          yield (item, count);
          end;
      yield (n.value, n.count);
      for item: t, count: int
        in tree[t]$increasing (n.greater) do
          yield (item, count);
          end;
    end;
  end increasing;
end tree;
```

An important feature of the *sorted_bag* and *tree* clusters is the way that the cluster parameter is used in places where the type **string** was used in *wordbag* and *wordtree*. This usage is especially evident in the implementation of *tree*. For example, *tree* has a representation that stores values of type *t*: the *value* component of a *node* must be an object of type *t*.

In the *insert* operation of *tree*, the *lt* and *equal* operations of type *t* are used. We have used the compound form, e.g. *t$equal* (*v*, *n.value*), to emphasize that the *equal* operation of *t* is being used. The short form, *v* = *n.value*, could have been used instead.

The *increasing* iterator of *tree* works as follows: first it yields all items in the current tree that are less than the item at the top node; the items are obtained by a recursive use of itself, passing the *lesser* subtree as an argument. Next it yields the contents of the top node, and then it yields all items in the current tree that are greater than the item at the top node (again by a recursive use of itself). In this way it performs a complete walk over the tree, yielding the values at all nodes, in increasing order.

Finally, we show in Figure 8 how the original procedure *count_words* can be implemented in terms of *sorted_bag*. Note that the *count_words* procedure now uses *sorted_bag*[**string**] instead of *wordbag*. *Sorted_bag*[**string**] is legitimate since the type **string** provides both *lt* and *equal* operations. Note that two **for** statements are used in *count_words*. The second **for** statement prints the words in alphabetic order, using the *increasing* iterator of *sorted_bag*. The first **for** statement inserts the words into the *sorted_bag*; it uses an iterator

```
words = iter (i: instream) yields (string);
        . . .
        end words;
```

The definition of *words* is left as an exercise for the reader.

5. The CLU Library

So far, we have shown CLU modules as separate pieces of text, without explaining how they are bound together to form a program. This section describes the CLU library, which plays a central role in supporting intermodule references.

The CLU library contains information about abstractions. The library supports incremental program development, one abstraction at a time, and, in addition, makes abstractions that are defined during the construction of one program available as a basis for subsequent program development. The information in the library permits the separate compilation of single modules with complete type checking of all external references (such as procedure invocations).

The structure of the library derives from the funda-

mental distinction between abstractions and implementations. For each abstraction, there is a *description unit* which contains all system-maintained information about that abstraction. Included in the description unit are zero or more modules that implement the abstraction.[9]

The most important information contained in a description unit is the abstraction's *interface specification*, which is that information needed to type-check uses of the abstraction. For procedural and control abstractions, this information consists of the number and types of parameters, arguments, and output values, plus any constraints on type parameters (i.e. required operations, as described in Section 4). For data abstractions, it includes the number and types of parameters, constraints on type parameters, and the name and interface specification of each operation.

An abstraction is entered in the library by submitting the interface specification; no implementations are required. In fact, a module can be compiled before any implementations have been provided for the abstractions that it uses; it is necessary only that interface specifications have been given for those abstractions. Ultimately, there can be many implementations of an abstraction; each implementation is required to satisfy the interface specification of the abstraction. Because all uses and implementations of an abstraction are checked against the interface specification, the actual selection of an implementation can be delayed until just before (or perhaps during) execution. We imagine a process of binding together modules into programs, prior to execution, at which time this selection would be made.

An important detail of the CLU system is the method by which CLU modules refer to abstractions. To avoid problems of name conflicts that can arise in large systems, the names used by a module to refer to abstractions can be chosen to suit the programmer's convenience. When a module is submitted for compilation, its external references must be bound to description units so that type checking can be performed. The binding is accomplished by constructing an *association list*, mapping names to description units, which is passed to the compiler along with the source code when compiling the module. The mapping in the association list is stored by the compiler in the library as part of the module. A similar process is involved in entering interface specifications of abstractions, as these will include references to other (data) abstractions.

When the compiler type-checks a module, it uses the association list to map the external names in the module to description units and then uses the interface specifications in those description units to check that the abstractions are used correctly. The type correctness of the module thus depends upon the binding of names to description units and the interface specifications in those description units, and could be invalidated if changes to the binding or the interface specifications were subsequently made. For this reason, the process of compilation permanently binds a module to the abstractions it uses, and the interface description of an abstraction, once defined, is not allowed to change. (Of course, a new description unit can be created to describe a modified abstraction.)

[9] Other information that may be stored in the library includes information about relationships among abstractions, as might be expressed in a module interconnection language [5, 21].

Fig. 8. The *count_words* procedure using iterators.

```
count_words = proc (i: instream, o: outstream);
  wordbag = sorted_bag[string];
  % create an empty wordbag
  wb: wordbag := wordbag$create ( );
  % scan document, adding each word found to wb
  for word: string in words (i) do
    wordbag$insert (wb, word);
    end;
  % print the wordbag
  total: int := wordbag$size (wb);
  for w: string, count: int in wordbag$increasing (wb) do
    print_word (w, count, total, o);
    end;
  end count_words;
```

6. Implementation

This section briefly describes the current implementation of CLU and discusses its efficiency.

The implementation is based on a decision to represent all CLU objects by *object descriptors*, which are fixed-size values containing a type code and some type-dependent information.[10] In the case of mutable types, the type-dependent information is a pointer to a separately allocated area containing the state information. For constant types, the information either directly contains the value (if the value can be encoded in the information field, as for integers, characters, and booleans) or contains a pointer to separately allocated space (as for strings). The type codes are used by the garbage collector to determine the physical representation of objects so that the accessible objects can be traced; they are also useful for supporting program debugging.

The use of fixed-size object descriptors allows variables to be fixed-size cells. Assignment is efficient: the object descriptor resulting from the evaluation of the expression is simply copied into the variable. In addition, a single size for variables facilitates the separate compilation of modules and allows most of the code of a parameterized module to be shared among all instantiations of the module. The actual parameters are made available to this code by means of a small parameter-dependent section, which is initialized prior to execution.

[10] Object descriptors are similar to capabilities [11].

Procedure invocation is relatively efficient. A single program stack is used, and argument passing is as efficient as assignment. Iterators are a form of coroutine; however, their use is sufficiently constrained that they are implemented using just the program stack. Using an iterator is therefore only slightly more expensive than using a procedure.

The data abstraction mechanism is not inherently expensive. No execution-time type checking is necessary. Furthermore, the type conversion implied by **cvt** is merely a change in the view taken of an object's type and does not require any computation.

A number of optimization techniques can be applied to a collection of modules if one is willing to give up the flexibility of separate compilation. The most effective such optimization is the inline substitution of procedure (and iterator) bodies for invocations [18]. The use of data abstractions tends to introduce extra levels of procedure invocations that perform little or no computation. As an example, consider the *wordbag$insert* operation (Figure 3), which merely invokes the *wordtree$insert* operation and increments a counter. If data abstractions had not been used, these actions would most likely have been performed directly by the *count_words* procedure. The *wordbag$insert* operation is thus a good candidate for being compiled inline. Once inline substitution has been performed, the increase in context will enhance the effectiveness of conventional optimization techniques [1–3].

7. Discussion

Our intent in this paper has been to provide an informal introduction to the abstraction mechanisms in CLU. By means of programming examples, we have illustrated the use of data, procedural, and control abstractions and have shown how CLU modules are used to implement these abstractions. We have not attempted to provide a complete description of CLU, but, in the course of explaining the examples, most features of the language have appeared. One important omission is the CLU exception handling mechanism (which does support abstractions); this mechanism is described in [10].

In addition to describing constructs that support abstraction, previous sections have covered a number of other topics. We have discussed the semantics of CLU. We have described the organization of the CLU library and discussed how it supports incremental program development and separate compilation and type checking of modules. Also we have described our current implementation and discussed its efficiency.

In designing CLU, our goal was to simplify the task of constructing reliable software that is reasonably easy to understand, modify, and maintain. It seems appropriate, therefore, to conclude this paper with a discussion of how CLU contributes to this goal.

The quality of any program depends upon the skill of the designer. In CLU programs, this skill is reflected in the choice of abstractions. In a good design, abstractions will be used to simplify the connections between modules and to encapsulate decisions that are likely to change [17]. Data abstractions are particularly valuable for these purposes. For example, through the use of a data abstraction, modules that share a system database rely only on its abstract behavior as defined by the database operations. The connections among these modules are much simpler than would be possible if they shared knowledge of the format of the database and the relationship among its parts. In addition, the database abstraction can be reimplemented without affecting the code of the modules that use it. CLU encourages the use of data abstractions and thus aids the programmer during program design.

The benefits arising from the use of data abstractions are based on the constraint, inherent in CLU and enforced by the CLU compiler, that only the operations of the abstraction may access the representations of the objects. This constraint ensures that the distinction made in CLU between abstractions and implementations applies to data abstractions as well as to procedural and control abstractions.

The distinction between abstractions and implementations eases program modification and maintenance. Once it has been determined that an abstraction must be reimplemented, CLU guarantees that the code of all modules using that abstraction will be unaffected by the change. The modules need not be reprogrammed or even recompiled; only the process of selecting the implementation of the abstraction must be redone. The problem of determining what modules must be changed is also simplified because each module has a well-defined purpose—to implement an abstraction—and no other module can interfere with that purpose.

Understanding and verification of CLU programs is made easier because the distinction between abstractions and implementations permits this task to be decomposed. One module at a time is studied to determine that it implements its abstraction. This study requires understanding the behavior of the abstractions it uses, but it is not necessary to understand the modules implementing those abstractions. Those modules can be studied separately.

A promising way to establish the correctness of a program is by means of a mathematical proof. For practical reasons, proofs should be performed (or at least checked) by a verification system, since the process of constructing a proof is tedious and error-prone. Decomposition of the proof is essential for program proving, which is practical only for small programs (like CLU modules). Note that when the CLU compiler does type checking, it is, in addition to enforcing the constraint that permits the proof to be decomposed, also performing a small part of the actual proof.

We have included as declarations in CLU just the information that the compiler can check with reasonable efficiency. We believe that the other information required for proofs (specifications and assertions) should be expressed in a separate "specification" language. The properties of such a language are being studied [7, 13, 14, 19]. We intend eventually to add formal specifications to the CLU system; the library is already organized to accommodate this addition. At that time various specification language processors could be added to the system.

We believe that the constraints imposed by CLU are essential for practical as well as theoretical reasons. It is true that data abstractions can be used in any language by establishing programming conventions to protect the representations of objects. However, conventions are no substitute for enforced constraints. It is inevitable that the conventions will be violated—and are likely to be violated just when they are needed most, in implementing, maintaining, and modifying large programs. It is precisely at this time, when the programming task becomes very difficult, that a language like CLU will be most valuable and appreciated.

Acknowledgments. The authors gratefully acknowledge the contributions made by members of the CLU design group over the last three years. Several people have made helpful comments about this paper, including Toby Bloom, Dorothy Curtis, Mike Hammer, Eliot Moss, Jerry Saltzer, Bob Scheifler, and the referees.

References

1. Allen, F.E., and Cocke, J. A catalogue of optimizing transformations. Rep. RC 3548. IBM Thomas J. Watson Res. Ctr., Yorktown Heights, N.Y., 1971.
2. Allen, F.E. A program data flow analysis procedure. Rep. RC 5278, IBM Thomas J. Watson Res. Ctr., Yorktown Heights, N.Y., 1975.
3. Atkinson, R.R. Optimization techniques for a structured programming language. S.M. Th., Dept. of Electr. Eng. and Comptr. Sci., M.I.T., Cambridge, Mass., June 1976.
4. Dahl, O.J., Myhrhaug, B., and Nygaard, K. The SIMULA 67 common base language. Pub. S-22, Norwegian Comptng. Ctr., Oslo, 1970.
5. DeRemer, F., and Kron, H. Programming-in-the-large versus programming-in-the-small. Proc. Int. Conf. on Reliable Software, SIGPLAN Notices 10, 6 (June 1975), 114–121.
6. Dijkstra, E.W. Notes on structured programming. *Structured Programming, A.P.I.C. Studies in Data Processing No. 8*, Academic Press, New York, 1972, pp. 1–81.
7. Guttag, J.V., Horowitz, E., and Musser, D.R. Abstract data types and software validation. Rep ISI/RR-76-48, Inform. Sci. Inst., U. of Southern California, Marina del Rey, Calif., Aug. 1976.
8. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica 4* (1972), 271–281.
9. Knuth, D. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison Wesley, Reading, Mass., 1973.
10. Laboratory for Computer Science Progress Report 1974–1975. Comput. Structures Group. Rep. PR-XII, Lab. for Comptr. Sci., M.I.T. To be published.
11. Lampson, B.W. Protection. Proc. Fifth Annual Princeton Conf. on Inform. Sci. and Syst., Princeton U., Princeton, N.J., 1971, pp. 437–443.
12. Liskov, B.H., and Zilles, S.N. Programming with abstract data types. Proc. ACM SIGPLAN Conf. on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974), 50–59.
13. Liskov, B.H., and Zilles, S.N. Specification techniques for data abstractions. *IEEE Trans. Software Eng., SE-1* (1975), 7–19.
14. Liskov, B.H., and Berzins, V. An appraisal of program specifications. Comput. Structures Group Memo 141, Lab. for Comptr. Sci., M.I.T., Cambridge, Mass., July 1976.
15. McCarthy, J., et al. *LISP 1.5 Programmer's Manual.* M.I.T. Press, Cambridge, Mass., 1962.
16. Morris, J.H. Protection in programming languages. *Comm. ACM 16*, 1 (Jan. 1973), 15–21.
17. Parnas, D.L. Information distribution aspects of design methodology. Information Processing 71, Vol. 1, North-Holland Pub. Co., Amsterdam, 1972, pp. 339–344.
18. Scheifler, R.W. An analysis of inline substitution for the CLU programming language. Comput. Structures Group Memo 139, Lab. for Comptr. Sci., M.I.T., Cambridge, Mass., June 1976.
19. Spitzen, J., and Wegbreit, B. The verification and synthesis of data structures. *Acta Informatica 4* (1975), 127–144.
20. Standish, T.A. Data structures: an axiomatic approach. Rep. 2639, Bolt, Beranek and Newman, Cambridge, Mass., 1973.
21. Thomas, J.W. Module interconnection in programming systems supporting abstraction. Rep. CS-16, Comptr. Sci. Prog., Brown U., Providence, R. I., 1976.
22. Wirth, N. Program development by stepwise refinement. *Comm. ACM 14*, 4 (1971), 221–227.
23. Wirth, N. The programming language PASCAL. *Acta Informatica 1* (1971), 35–63.
24. Wulf, W.A., London, R., and Shaw, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Eng. SE-2* (1976), 253–264.