PROGRAMMING WITH ABSTRACT DATA TYPES

Barbara Liskov
Massachusetts Institute of Technology
Project MAC
Cambridge, Massachusetts

Stephen Zilles
Cambridge Systems Group
IBM Systems Development Division
Cambridge, Massachusetts

Abstract

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient.

This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstractions are given.

Introduction

This paper describes an approach to computer representation of abstraction. The approach, developed while designing a language to support structured programming, is also relevant to work in very-high-level languages. We begin by explaining its relevance and by comparing work in structured programming and very-high-level languages.

The purpose of structured programming is to enhance the reliability and understandability of programs. Very-high-level languages, while primarily intended to increase programmer productivity by easing the programmer's task, can also be expected to enhance the reliability and understandability of code. Thus, similar benefits can be expected from work in the two areas.

Work in the two areas, however, proceeds along different lines. A very-high-level language attempts to present the user with the abstractions (operations, data structures, and control structures) useful to his application area. The user can use these abstractions without being concerned with how they are implemented -- he is only concerned with what they do. He is thus able to ignore details not relevant to his application area, and to concentrate on solving his problem.

Structured programming attempts to impose a discipline on the programming task so that the resulting programs are "well-structured." In this discipline, a problem is solved by means of a process of successive decomposition. The first step is to write a program which solves the problem but which runs on an abstract machine, one which provides just those data objects and operations which are ideally suited to solving the problem. Some or all of those data objects and operations are truly abstract, i.e., not present as primitives in the programming language being used. We will, for the present, group them loosely together under the term "abstraction."

The programmer is initially concerned with satisfying himself (or proving) that his program correctly solves the problem. In this analysis he is concerned with the way his program makes use of the abstractions, but not with any details of how those abstractions may be realized. When he is satisfied with the correctness of his program, he turns his attention to the abstractions it uses. Each abstraction represents a new problem, requiring additional programs for its solution. The new program may also be written to run on an abstract

Work reported herein was supported in part by the National Science Foundation under research grant GJ-34671.

machine, introducing further abstractions. The original problem is completely solved when all abstractions generated in the course of constructing the program have been realized by further programs.

It is clear now that the approaches of very-high-level languages and structured programming are related to one another: each is based on the idea of making use of those abstractions which are correct for the problem being solved. Furthermore, the rationale for using the abstractions is the same in both approaches: to free the programmer from concern with details not relevant to the problem he is solving.

In very-high-level languages, the designers attempt to identify the set of useful abstractions in advance. A structured programming language, on the other hand, contains no preconceived notions about the particular set of useful abstractions, but, instead, must provide a mechanism whereby the language can be extended to contain the abstractions which the user requires. A language containing such a mechanism can be viewed as a general-purpose, indefinitely-high-level language.

In this paper we describe an approach to abstraction which permits the set of built-in abstractions to be augmented when the need for new abstractions is discovered. We begin by analyzing the abstractions used in writing programs, and identify the need for data abstractions. A language supporting the use and definition of data abstractions is informally described, and some example programs are given. Remaining sections of the paper discuss the relationship of the approach to previous work, and some aspects of the implementation of the language.

The Meaning of Abstraction

The description of structured programming given in the preceding section is vague because it is couched in such undefined terms as "abstraction" and "abstract machine." In this section we analyze the meaning of "abstraction" to determine what kinds of abstraction a programmer requires, and how a structured programming language can support these requirements.

What we desire from an abstraction is a mechanism which permits the expression of relevant details and the suppression of irrelevant details. In the case of programming, the use which may be made of an abstraction is relevant; the way in which the abstraction is implemented is irrelevant. If we consider conventional programming languages, we discover that they offer a powerful aid to abstraction: the function or procedure. When a programmer makes use of a procedure, he is (or should be) concerned only with what it does -- what function it provides for him. He is not concerned with the algorithm executed by the procedure. In addition, procedures provide a means of decomposing a problem -- performing part of the programming task inside a procedure, and another part in the program which calls the procedure. Thus, the existence of procedures goes quite far toward capturing the meaning of abstraction.

Unfortunately, procedures alone do not provide a sufficiently rich vocabulary of abstractions. The abstract data objects and control structures of the abstract machine mentioned above are not accurately represented by independent procedures. Because we are considering abstraction in the context of structured programming, we will omit discussion of control abstractions.

This leads us to the concept of abstract data type which is central to the design of the language. An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type.

We believe that the above concept captures the fundamental properties of abstract objects. When a programmer makes use of an abstract data object, he is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation. The behavior of an object is captured by the set of characterizing operations. Implementation information, such as how the object is represented in storage, is only needed when defining how the characterizing operations are to be implemented. The user of the object is not required to know or supply this information.

Abstract types are intended to be very much like the built-in types provided by a programming language. The user of a built-in type, such as integer or integer array, is only concerned with creating objects of that type and then performing operations on them. He is not (usually) concerned with how the data objects are represented, and he views the operations on the objects as indivisible and atomic when in fact several machine instructions may be required to perform them. In addition, he is not (in general) permitted to decompose the objects. Consider, for example, the built-in type integer. A programmer wants to declare objects of type integer and to perform the usual arithmetic operations on them. He is usually not interested in an integer object as a bit string, and cannot make use of the format of the bits within a computer word. Also, he would like the language to protect him from foolish misuses of types (e.g., adding an integer to a character) either by treating such a thing as an error (strong typing), or by some sort of automatic type conversion.

In the case of a built-in data type, the programmer is making use of a concept or abstraction which is realized at a lower level of detail -- the programming language itself and its compiler. Similarly, an abstract data type is used at one level and realized at a lower level, but the lower level does not come into existence automatically by being part of the language. Instead, an abstract data type is realized by writing a special kind of program, called an operation cluster, or cluster for short, which defines the type in terms of the operations which can be performed on it. The language facilitates this activity by allowing the use of an abstract data type without requiring its on-the-spot definition. The language processor supports abstract data types by building links between the use of a type and its definition (which may be provided either earlier or later), and by enforcing the view

of a data type as equivalent to a set of operations by a very strong form of data typing.

We observe that a consequence of the concept of abstract data types is that most of the abstract operations in a program will belong to the sets of operations characterizing abstract types. We will use the term functional abstraction to denote those abstract operations which do not belong to any characterizing set. A functional abstraction will be implemented as a composition of the characterizing operations of one or more data types, and will be supported in the usual way by a procedure. A sine routine might be an example of such a functional abstraction. The implementation of the sine routine could be a Taylor series expansion expressed in terms of characterizing operations of the type real.

The Programming Language

We now give an informal description of a programming language which permits the use and definition of abstract data types. This language is a simplified version of a structured programming language that is under development at M.I.T. It is derived primarily from PASCAL[1] and is conventional in many respects, but it differs from conventional languages in several important ways.

The language provides two forms of modules corresponding to the two forms of abstraction: procedures, which support functional abstractions, and operation clusters, which support abstract data types. Each module is translated (compiled) by itself.

The language has no free variables in the conventional sense. Within a module, the only names that are free, and therefore are defined externally, are the names of other modules; that is, cluster names and procedure names. These names are bound at translation time by means of a directory of module names created by the programmer expressly for this purpose. No names remain to be bound in the translated module.

The language has only structured control. There are no goto's or labels, but merely variants of concatenation, selection (if, case) and iteration (while) constructions. A structured error-handling mechanism is under development. In this paper, it is represented only by the presence of the reserved word error.

The way in which the language permits the use and definition of abstract data types can best be illustrated by an example. We have chosen the following problem: Write a program, Polish_gen, which will translate from an infix language to a Polish post-fix language. Polish_gen is to be a general-purpose program which makes no assumptions about input or output devices (or files). It makes only the following assumptions about the input language:

1. The input language has an operator precedence grammar.

2. A symbol of the input language is either an arbitrary string of letters and numbers, or a single, non-alphanumeric character; blanks terminate symbols but are otherwise ignored.

For example, if Polish_gen received the string

a + b * (c + d)

as input, it would produce the string

a b c d + * +

as output. We have chosen this problem as our example because the problem and its solution are familiar to people interested in programming languages, and the problem is sufficiently complex to illustrate the use of many abstractions.

Using Abstract Data Types

The procedure Polish_gen, shown in Figure 1, performs the translation described above. It takes three arguments: input, an object of abstract type infile which holds the sentence of the input language; output, an object of abstract type outfile which will accept a sentence of the output language; and g, an object of abstract type grammar which can be used to recognize symbols of the input language and determine their precedence relations. In addition, Polish_gen makes use of local variables of abstract types stack and token. Note that all the data-type-names appear free in Polish_gen, as does "scan," which names the single functional abstraction used by Polish_gen.

The language uses the same syntax to declare variables of abstract data type as to declare variables of primitive type. The syntax distinguishes between declarations which involve the creation of an object and those which do not. For example,

t: token

states that t is the name of a variable which holds an object of abstract type token, but that no token object is to be created, so that the value of t is initially undefined. Thus the variable t is being

```
Polish_gen: procedure(input: infile,
                      output: outfile, g: grammar);

 t: token;
 mustscan: boolean;
 s: stack(token);

 mustscan := true;
 stack$push(s, token(g, grammar$eof(g)));
 while ¬stack$empty(s) do
    if mustscan
       then t := scan(input, g)
       else mustscan := true;
    if token$is_op(t)
       then
          case token$prec_rel(stack$top(s), t) of
             "<":: stack$push(s, t);
             "=":: stack$erasetop(s);
             ">":: begin
                      outfile$out_str(output,
                         token$symbol(stack$pop(s)));
                      mustscan := false;
                   end
          otherwise error;
       else outfile$out_str(output, token$symbol(t));
 end
 outfile$close(output);
 return;
end Polish_gen
```

Figure 1

declared in the same way as mustscan in

```
mustscan: boolean
```

The presence of parentheses following the type name signals creation of an object. For example,

```
s: stack(token)
```

states that s is the name of a variable which holds an object of abstract type stack, and a stack object is to be created and stored in s. Information required for creating the object is passed in a parameter list; in the example, the only parameter, token, defines the type of element which may be placed on the stack s. The declaration of a stack is similar to an array declaration, such as "array[1..10] of characters," in that they both require the type of elements to be specified.

The language is strongly typed; thus there are only three ways in which an abstract object can be used:

1. An abstract object may be operated upon by the operations which define its abstract type.
2. An abstract object may be passed as a parameter to a procedure. In this case, the type of the actual argument passed by the calling procedure must be identical to the type of the corresponding formal parameter in the called procedure.
3. An abstract object may be assigned to a variable, but only if the variable is declared to hold objects of that type.

Application of a defining operation to an abstract object is indicated by an operation call in which a compound name is used: for example,

```
grammar$eof(g)
stack$push(s, t)
token$is_op(t)
```

The first part of the compound name identifies the abstract type to which the operation belongs while the second component identifies the operation. An operation call will always have at least one parameter -- an object of the abstract type to which the operation belongs.

There are several reasons why the type-name is included in the operation call. First, since an operation call may have several parameters of different abstract types, the absence of the type-name may lead to an ambiguity as to which object is actually being operated on. Second, use of the compound name permits different data types to use the same names for operations without any clash of identifiers arising. Third, we believe that the type-name prefix will enhance the understandability of programs, once the reader is used to the notation. Not only is the type of the operation immediately apparent, but operation calls are clearly distinguished from procedure calls.

The statement

```
t := scan(input, g)
```

illustrates both passing abstract objects as parameters, and assigning an abstract object to a variable. The procedure scan, shown in Figure 2, expects objects of type infile and grammar as its arguments, and returns an object of type token,

```
scan: procedure(input: infile, g: grammar)
        returns token;
   newsymb: string;
   ch: char;
   ch := infile$get(input)
   while ch=" " do ch := infile$get(input); end
   if infile$eof(input)
      then return token(g, grammar$eof(g));
   newsymb := unit_string(ch);
   if alphanumeric(ch) then
      while alphanumeric(infile$peek(input)) do
         newsymb := newsymb concat infile$get(input);
      end
   return token(g, newsymb);
end scan
```

Figure 2

which is then stored in the token variable t.

We have explained that objects can be created in conjunction with variable declaration. It is also possible for objects to be created independently of variable declaration. Object creation is specified (whether inside a declaration or not) by the appearance of the type-name followed by parentheses. For example, in the last line of scan

```
token(g, newsymb)
```

states that a token object, representing the symbol just scanned, is to be created; the information required to create the object (the grammar and the symbol just scanned) is passed in a parameter list.

A brief description of the logic of Polish_gen can now be given. Polish_gen uses the functional abstraction scan to obtain a symbol of the grammar from the input string. Scan returns the symbol in the form of a token -- a type introduced to provide efficient execution without revealing information about how the grammar represents symbols. Polish_gen stores the token containing the newly scanned symbol in variable t. If t holds a token representing an identifier (like "a") rather than an operator (like "+"), that identifier is put in the output file immediately. Otherwise, the token on top of the stack is compared with t to determine the precedence relation between them. If the relation is "<", t is pushed on the stack (e.g., "+" < "*"). If the relation is "=", both t and the top-of-stack token are discarded (e.g., "("=")"). If the relation is ">", the operator held in the top-of-stack token is appended to the output file, exposing a new top-of-stack token. Since that operator token may have a higher precedence than t, the boolean variable mustscan is used to prevent a new symbol from being scanned and to insure the next comparison is with the current value of t. Because a grammar-dependent representation of the end of file symbol (grammar$eof(g)) is initially pushed onto the stack, the stack will become empty causing Polish_gen to complete only when a matching eof token is generated by exhausting the input. (We have made the simplifying assumption that the input is a legitimate sentence of the infix language.)

The scan procedure obtains characters from the input file via the operations defining the abstract type infile. It makes use of the data types char and string, and operations on objects of these types. Although these types are shown as built-in, they

could easily have been abstract types instead. In that case, the built-in predicate alphanumeric, for example, would have been expressed as char$alphanumeric. Only the syntax would change; the meaning and use of the types would be the same in either case.

To sum up, Polish_gen makes use of five data abstractions, infile, outfile, grammar, token and stack, plus one functional abstraction, scan. The power of the data abstractions is illustrated by the types infile and outfile, which are used to shield Polish_gen from any physical facts concerning its input and output, respectively. Polish_gen does not know what input and output devices are being used, when the I/O actually takes place, nor does it know how characters are represented on the devices. What it does know is just enough for its needs: For parameter output it knows how to add a string of characters (outfile$out_str) and how to signify that the output is complete (outfile$close). For parameter input, it knows how to obtain the next character (infile$get), how to look at the next character without removing it from input (infile$peek), and how to recognize the end of input (infile$eof). (Note that for scan to operate correctly, infile must provide a non-blank, non-alphanumeric character on any call on infile$get or infile$peek after the end of file has been reached.) In every case its knowledge consists of the names of the operations which provide these services.

Defining Abstract Data Types

In this section, we describe the programming object -- the operation cluster -- whose translation provides an implementation of a type. The cluster contains code implementing each of the characterizing operations and thereby embodies the idea that a data type is defined by a set of operations.

As an example, consider the abstract data type stack used by Polish_gen. A cluster supporting stacks is shown in Figure 3. This cluster implements a very general kind of stack object in which the type of the stack elements is not known in advance. The cluster parameter element_type indicates the type of element a particular stack object is to contain.

The first part of a cluster definition provides a very brief description of the interface which the cluster presents to its users. The cluster interface defines the name of the cluster, the parameters required to create an instance of the cluster (an object of the abstract type which the cluster implements), and a list of the operations defining the type which the cluster implements, e.g.,

```
stack: cluster(element-type: type)
          is push, pop, top, erasetop, empty
```

The use of the reserved word is underlines the idea of a data type being characterized by a group of operations.

The remainder of the cluster definition, describing how the abstract type is actually supported, contains three parts: the object representation, the code to create objects and the operation definitions.

```
stack: cluster(element_type: type)
          is push, pop, top, erasetop, empty;

   rep(type_param: type) = (tp: integer;
                            e_type: type;
                            stk: array[1..]
                                  of type_param;

   create

      s: rep(element_type);

      s.tp := 0;
      s.e_type := element_type;
      return s;
      end

   push: operation(s: rep, v: s.e_type);

      s.tp := s.tp+1;
      s.stk[s.tp] := v;
      return;
      end

   pop: operation(s: rep) returns s.e_type;

      if s.tp = 0 then error;
      s.tp := s.tp-1;
      return s.stk[s.tp+1];
      end

   top: operation(s: rep) returns s.e_type;

      if s.tp = 0 then error;
      return s.stk[s.tp];
      end

   erasetop: operation(s: rep);

      if s.tp = 0 then error;
      s.tp := s.tp-1;
      return;
      end

   empty: operation(s: rep) returns boolean;

      return s.tp = 0;
      end

   end stack
```

Figure 3

Object Representation. Users of the abstract data type view objects of that type as indivisible entities. Inside the cluster, however, objects are viewed as decomposable into elements of more primitive type. The rep description

rep{(⟨rep-parameters⟩)} = ⟨type-definition⟩

defines a new type, denoted by the reserved word rep, which is accessible only within the cluster and describes how objects are viewed there. The ⟨type-definition⟩ defines a template which permits objects of that type to be built and decomposed. In general, it will make use of the data structuring methods provided by the language: arrays (possibly unbounded) or PASCAL records. The optional ("{}") ⟨rep-parameters⟩ make it possible to delay specifying some aspects of the ⟨type definition⟩ until an instance of the rep is created. Consider the rep

description of the stack cluster:

```
rep(type_param: type) = (tp: integer; e_type: type;
                         stk: array[1..] of type_param)
```

The ⟨type-definition⟩ specifies that a stack object is represented by a record containing three components named tp, stk, and e_type. The parameter, type_param, specifies the type of element which may be stored in the unbounded array named stk which will hold the elements pushed onto a stack object. This same type will also be stored in the e_type component, and is used for type checking as will be described below. The tp component holds the index of the topmost element of the stack.

Object Creation. The reserved word create marks the create_code, the code to be executed when an object of the abstract type is created. The cluster may be viewed as a procedure whose procedure body is the create-code. When a user indicates that an object of abstract type is to be created, for example,

```
s: stack(token)
```

one thing that happens (at execution time) is a call on the create-code, causing that procedure body to be executed. The parameters of the cluster are actually parameters of the create-code. Since free variables, other than references to externally defined modules, are not provided, these parameters are not accessible either to the operations or to the ⟨type definition⟩ in the rep. Therefore, any information about the parameters that is to be saved must be explicitly inserted into each instance of the rep.

The code shown in the stack cluster is typical of create-code. First, an object of type rep is created; that is, space is allocated to hold the object as defined by the rep. Then, some initial values are stored in the object. Finally, the object is returned to the caller. When the object is returned, its type is changed from type rep to the abstract type defined by the cluster.

Operations. The remainder of the cluster consists of a group of operation definitions, which provide implementations of the permissible operations on the data type. Operation definitions are like ordinary procedure definitions except that they have access to the rep of the cluster, which permits them to decompose objects of the cluster type. Operations are not themselves modules; they will be accepted by the translator only as part of a cluster.

Operations always have at least one parameter -- of type rep. Because the cluster may simultaneously support many objects of its defined type, this parameter tells the operation the particular object on which to operate. Note that the type of this parameter will change from the abstract type to type rep as it is passed between the caller and the operation.

Because the language is strongly typed, the type of objects pushed on a given stack must be checked for consistency with the type of elements the stack can hold. This consistency requirement is specified syntactically by declaring that the type of the second argument of push is to be the same as the e_type component of the rep of the stack object which is the first argument of push. The translator can generate code to verify that the types match at run time and to raise an error if they don't.

Controlling the Use of Information

Abstract data types were introduced as a way of freeing a programmer from concern about irrelevant details in his use of data abstractions. But in fact we have gone further than that. Because the language is strongly typed, the user is unable to make use of any implementation details. In this section we discuss the benefits that accrue from this limitation: the programs which result are more modular, and easier to understand, modify, maintain and prove correct.

Token is a good example of a type created to control access to implementation details. Instead of introducing a new type, Polish_gen could have been written to accept strings from scan, to store strings on the stack, and to compare strings to determine the precedence relation (via an appropriate operation grammar$prec_rel). Such a solution would be inefficient. Since the precedence matrix can be indexed by the positions of the operators in the reserved word table of the grammar, an efficient implementation would look up the character string only once to find out if it is an operator symbol and, if so, use the index of the operator in Polish_gen.

This, however, exposes information about the representation of the grammar. If Polish_gen or some other module which uses the grammar makes use of this information, normal maintenance and modification of the grammar cluster can introduce errors which are difficult to track down.[2] Therefore, the new type, token, is introduced to limit the distribution of information about how the grammar is represented. Now a redefinition of the grammar cluster can affect only the token cluster -- which makes no assumptions about the index it receives from grammar. If an error occurs while looking up a precedence relation (like an index out of bounds), the error can only have been caused by something in the token or grammar cluster.

Actually, the selection of an implementation of tokens -- for example, whether a token is represented by an integer or a character string -- involves a design decision. This decision can be delayed until the cluster for tokens is defined and need not be made during the coding of Polish_gen. Therefore, the programming of Polish_gen can be done according to one of Dijkstra's programming principles: build the program one decision at a time.[3] Following this principle leads to a simplified logic for Polish_gen, making it easier to understand and maintain.

Making the representation inaccessible also results in a program which is easier to prove correct. The proof of a program is divided into two parts: a proof that the cluster correctly implements the type, and a proof that the program using the type is correct. Only in the former proof need details of the implementation of type objects be considered; the latter proof is based only on the abstract properties of the types, which may be expressed in terms of relations among the characterizing operations for each type.

Relationship to Previous Work

Much work has been done in the area of creating suitable mechanisms for defining data types. There is no hope of surveying all that work here, nor is it all relevant to this paper. In this section we outline the areas of work that are most closely related to clusters in that they provide some tools for defining abstract data types, and we discuss how the cluster approach differs from that work. The related work can be roughly divided into three categories: extensible languages, implementation specifications for a set of standard abstract operators, and SIMULA 67 class definitions.

Extensible Languages

Much of the work and much of the success with extensible languages[4] has been in the area of data type definition. This work, however, has been primarily oriented toward defining representations rather than abstract types. New data representations, or modes as they are frequently called, are created by constructing the representation in terms of existing modes using the primitive mode construction facilities of the language. Mode construction facilities provided by an extensible language typically include mechanisms for defining pointers to objects, for defining unions of distinct mode classes, and for constructing aggregates (arrays and records) of objects. These correspond closely to the facilities used in this paper to define reps. The use of these mode definition mechanisms implies the definition of a set of constructors, selectors and predicates which may be applied to objects of the mode being defined. In some languages, the mode definitions may allow this set of operations to be augmented by certain operations, such as assignment, which are expressly provided for in the language.

The main problem with extensible languages is that they do not encourage the use of data abstractions. It is, in general, impossible to define all the operations characterizing an abstract data type within the mode definition. As we noted, only the representation of a data type is defined using the mode extension mechanism. Any abstract operation which is not equivalent to a constructor, selector or predicate for the representation must be defined outside the mode definition by a procedure or macro which can be made to appear like an operator by using the syntax extension facility. Therefore, a user must learn two different mechanisms; and the definition, instead of being collected in one place, as it is in an operation cluster, is split into distinct parts. Furthermore, it is difficult to restrict access to the representation solely to the characterizing operations of the abstract data type.

Standard Abstract Operations

The work derived from the earlier work of Mealy[5] and Balzer[6] is much closer in spirit to the approach taken here. Mealy established the view that a data collection is a map from a set of selectors to a set of values, and that operations on data collections are either transformations on the map or uses of the map to access elements. This view has led to attempts to standardize a set of abstract operators for data collections. For example, Balzer proposed a particular abstraction for such collections which defines a set of four abstract operators to create, access, modify, and destroy abstract data collections. The user would define a particular collection by specifying how each abstract operation was to be implemented. This work has been extended (e.g., Earley[7]), but its primary emphasis has remained on defining a standard set of abstract operations. More complex operations are defined as procedures written in terms of these abstract operations.

Although it is useful to distinguish some abstract operations, such as "create," which have a high probability of being applicable to every abstract data type, it seems unreasonable to expect that a predetermined set of operations will suffice to manipulate every abstract data object. Therefore, leaving the selection of the operations to the creator of the type, as is done with operation clusters, provides a more closely tailored abstraction.

SIMULA Classes

The language which most closely resembles, in form, the language presented here is SIMULA 67.[8] SIMULA class definitions have many similarities with cluster definitions. There is, however, a very important philosophical difference in these two languages which leads to several important linguistic differences. The classes of SIMULA were designed to represent and provide full accessibility to data objects. Every attribute and function in a class is accessible in the block in which the class definition is embedded. Therefore, the actual form of the representation is always known to the user.

In contrast to this, the rep of a cluster is not accessible outside the cluster. Operations in the cluster provide the only way to access the contents of the rep and, even then, only a subset of the operations defined in the cluster may be externally accessible. As a result of this philosophical difference, the mechanisms for referencing data, the use of non-local variable references, and the use of blocks and block structuring are quite different in the two languages.

Implementation Considerations

Most aspects of the implementation of clusters will be handled in a conventional manner. There are, however, several aspects of the implementation which deserve special mention because they are nonstandard or have a significant impact on the practicality of using clusters to represent abstract data.

Modules and Module-Names

The compiler accepts a module as input. A module will usually be a cluster, but will sometimes be a procedure like Polish_gen or scan. In the course of module translation, externally defined module-names, used to refer to procedures and data types, will be encountered. (Note that the references to operations on abstract data types do not introduce any additional external references because they are relative to the abstract type with which the operation name is prefixed.)

When the compiler processes a module it builds or adds to a description-unit containing information about the module. Information held in the description-unit includes:

1. The location of the object code generated by the compiler.

2. A description of the interface which the module makes available to its users. In particular, complete information about types of all parameters and values expected by the module is maintained. If the module is a cluster, information will be kept for each operation in the cluster.

3. A list of all modules which use the module.

Obviously much more information can be stored in the description-unit: debugging information in the form of symbol tables, etc., documentation information, specification information in the form of predicate calculus descriptions of input/output relationships, and even an analysis of the rationale for the decisions made in designing the module.

The description unit is the focus for all information about a module. It can be created when the module is processed or it can be created to be the target of references from other modules. Creating a description unit before the module it represents is processed supports top-down design and provides a simple way to define recursion. Since the description unit holds a list of all uses of the module, the consistency of the uses and the definition can be checked when the module is actually defined and appropriate error messages can be generated at that time. The actual definition can be delayed for quite some time as the description unit can be used to locate code to simulate the behavior of the module for debugging purposes.

In the course of translating a module, the translator must give a meaning to each module name by binding it to the code of the corresponding module. This is done via the description unit. The translator obtains access to description units by means of a directory, containing a set of module-name/description unit pairs, which it receives as an argument. All external references must be resolved by means of this directory; if they cannot be resolved, an appropriate error message is generated.

The directory is a user-constructed object which is, in general, built to control the translation of a specific set of related modules. The actual description units are stored in a multilevel, tree structured file system similar to the MULTICS file system,[9] and the references to description units in a directory are actually references into this file system. The primitives for constructing directories and for manipulating the file system are independent of the language, forming the "file system cluster" and the "directory cluster."

Type Checking

The language described in this paper is based on the idea of strong type checking, and the language translator is supposed to enforce strong type checking even across the interface between two separately compiled procedures. In this section we discuss some of the problems arising from strong type checking.

Strong type checking means that whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function. If the called function is a procedure, the types must match identically. If the called function is an operation, then the types must match identically unless the object is of the abstract type defined by the cluster to which the operation belongs. In this case, the type of the object is changed to the type rep for that cluster. Thus, the type checking mechanism controls whether the representation of an object is visable to a given operation. If a type error were undetected in this case, information supposed to have been inaccessible outside of the cluster, will become accessible, and program modularity will be destroyed.

Type checking in this language is more complex than in most conventional languages. This is because user-defined abstractions, both data types and procedures, may have types as parameters. Consider the data type stack defined above. We have noted the similarity between stacks and arrays: In each case, a type specification for the components of the structure must be supplied before an instance can be created. Constructs, such as stack and array, are called type generators because they define a class of types rather than a single type. Each individual type in the class is generated by supplying type definitions for each of the type parameters of the type generator. A type generator, like stack, which is built to serve the needs of future users, defines an open-ended class of types, and the members of its type class are not known at the time the stack cluster is compiled.

One of the effects of allowing user defined type generators is that some of the operations in the cluster for that "type" are polymorphic; that is, the operations may be defined over many different type domains, subject to the constraint that the types of any given set of arguments are type-consistent. An example of such an operation is push in the stack cluster. Push takes as its operands a stack and a value. The type consistency requirement for push is that, if the type of the stack is "stack of T," the value pushed must be of type T; thus, strong type checking for the operation push involves determining that its stack argument really is a stack, determining the type of the stack argument, determining the type of the value being pushed and determining that they satisfy the consistency requirement.

It is desirable to do compile-time type checking, since type errors are detected as early as possible. Because of the freedom with which types can be used in the language, however, it is not clear how complete the compile time type checking can be. Therefore, the design of the language is based on a run-time type checking mechanism which is augmented by as much compile-time checking as is possible.

It is clear that given a suitable representation of types, a run time check for identically matching types can be programmed. The kind of type checking which results in the representation of an object being exposed to an operation can be handled at run-time by a technique described by Morris[10] which is an outgrowth of the work on protection in operating systems. (There is a strong correlation between clusters and protected subsystems; clusters provide a natural mechanism for encapsulating private information.)[11]

In the future we may be able to dispense with the run time mechanism, since recent work by John Reynolds[12] indicates that complete compile-time type checking may be possible. We look forward to the completion of Reynolds' work, and intend to design a version of the language based on compile-time type checking in the near future.

Retention

The language has been designed to permit activations of clusters, procedure and operation to be implemented using a stack discipline. Clusters, procedures and operations have no free variables at execution time, and all variables defined therein are purely local. All information that is to be retained or shared must be stored in the rep of an object. The objects are allocated in a heap where the retention strategy is used. In practice, there are a number of easily identified cases where objects need not be placed in the heap but can instead be allocated on the stack, either because the object is not shared or because once it has been allocated its content never changes. These cases may be optimized by the language translator.

Efficiency

We believe it is helpful to associate two structures with a program: its logical structure and its physical structure. The primary business of a programmer is to build a program with a good logical structure -- one which is understandable and leads to ease in modification and maintenance.[13] However, a good logical structure does not necessarily imply a good physical structure -- one which is efficient to execute. In fact, the techniques employed to achieve good logical structure (hierarchy, access to data only through functions, etc.) in many cases seem to imply bad physical structure.

We believe it is the business of the compiler to map good logical structure into good physical structure. The fact that the two structures may diverge is acceptable provided that the compiler is verified, and that all programming tools (for example, the debugging aids) are defined to hide the divergence.

The language is intended to be compiled by an optimizing compiler which achieves a good physical structure in the output code. An important efficiency can be obtained from the fact that the language is flexible with respect to the meaning of an operation call. Each operation call may be replaced either by an actual call upon the corresponding operation or by inline code for the operation. Two aspects of the language design make this flexibility possible:

1. Because the syntax for an operation call is identical in both cases, it is possible to change the compiling technique that is used without rewriting the procedure in which the operator is used.

2. The invariant portion of the cluster -- the code for the operations -- has been carefully separated from the rep, which holds the object dependent information; thus, inline insertion of the code is possible.

Inline insertion of the code for an operation allows that code to be subject to the optimization transformations available in the compiler. Optimizing transformations, such as compile-time evaluation and common subexpression elimination, remove redundant computations, thereby decreasing the time needed to execute the operation. For example, all error checks in the stack cluster operations could be eliminated if those operations were inserted inline in Polish_gen. These standard optimization techniques should be extremely effective because the compiler is dealing with a structured program; the lack of free variables, and of goto's and other confusing control structures implies that a thorough data and control flow analysis can be performed. In other words, the compiler can benefit from the good logical structure of the program to obtain a thorough understanding of it, just as a person can.

The price paid to obtain this execution time optimization is an increase in the cost of redefining or modifying a module. Each such modification may require the recompilation of the modules which use the modified functions inline. Since the decision to use inline code can be delayed until performance measurements indicate which sections of a system are critical, one need relinquish the flexibility of easy program modification only where a positive performance benefit would result from inline code. Note that the list of the uses of the module, kept in the description-unit, can be used to cause automatic recompilation when changes are made.

Conclusions

This paper described a new kind of abstraction, the abstract data type, which augments our ability to make use of abstraction in building programs. The approach was discussed both as a concept and as a part of a programming language. Several examples of its use were given. An abstract data type was defined to be a class of objects which is completely characterized by the operations which may be performed on those objects. A new linguistic construct, the operation cluster, was introduced to provide programming language support for abstract data types.

The rationale behind undertaking to develop the language was to make the practice of structured programming more understandable by providing a langauge in which the abstractions uncovered in the course of program design could be expressed. We believe that the concept of abstract data type provides data abstraction in a form most useful to the programmer: he need only be aware of the behavior of an abstract object, which is precisely the information he needs to write his program, and irrelevant details about how the object is represented in storage and how the operations are implemented, are hidden from him. In fact, he is unable to make use of implementation details, leading to an improvement in program quality: programs will be more modular, and easier to understand, modify, maintain, and prove correct.

Of course, program quality is most dependent on good program design. Although a language can never teach a programmer what constitutes a well-designed program, it can guide him into thinking about the right things. We believe that abstraction is the key to good design,[13] and we have discovered in our experiments in using the language that it

encourages the programmer to consciously search for abstractions, especially data abstractions, and to think very hard about their use and definition.

We believe that the approach to abstraction discussed in the paper can be usefully incorporated in many different kinds of languages. It is unlikely that any language, no matter how high-level, contains all the abstractions which any person working in it would require. Therefore, the abstraction-building-mechanism described in this paper would be a useful feature of a very-high-level language.

Acknowledgements

The authors gratefully acknowledge the helpful comments on the content and structure of the paper made by Jack Dennis, Austin Henderson, Greg Pfister, and the referees.

References

1. Wirth, N. The programming language PASCAL. Acta Informatica, Vol. 1 (1971), pp 35-63.

2. Parnas, D. L. Information distribution aspects of design methodology, Proceedings of the IFIP Congress, August 1971.

3. 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.

4. Schuman, S. A. and P. Jorrand. Definition mechanisms in extensible programming languages. Proceedings of the AFIPS, Vol. 37, 1970, pp 9-19.

5. Mealy, G. Another look at data. Proceedings of the AFIPS, Vol. 31, 1967, pp 525-534.

6. Balzer, R. M. Dataless programming. Proceedings of the AFIPS, Vol. 31, 1967, pp 557-566.

7. Earley, J. Toward an understanding of data structures. Comm. of the ACM, Vol. 14, No. 10 (October 1971), pp 617-627.

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

9. Daley, R. C., and P. G. Neumann. A general-purpose file system for secondary storage. Proceedings of the AFIPS, Vol. 27, 1965, pp 213-229.

10. Morris, J. H., Jr. Protection in programming languages. Comm. of the ACM, Vol. 16, No. 1 (January 1973), pp 15-21.

11. Zilles, S. N. Procedural encapsulation: a linguistic protection technique. SIGPLAN Notices, Vol. 8, No. 9 (September 1973), pp 140-146.

12. Reynolds, J. Personal communication.

13. Liskov, B. H. A design methodology for reliable software systems. Proceedings of the AFIPS, Vol. 41, 1972, pp 191-199.