THE USE OF ABSTRACT DATA TYPES

TO SIMPLIFY PROGRAM MODIFICATIONS

Theodore A. Linden
Institute for Computer Sciences and Technology
National Bureau of Standards

ABSTRACT

If a program is structured using abstract data types as the basic unit of modularity, then that program is much easier to extend or modify. This thesis is illustrated by the staged development of a program to compute prime numbers based on the sieve of Eratosthenes. This paper includes an extensive introduction to the concept of abstract data types and can be used as a tutorial survey. It includes discussions on the use of abstract data types in connection with recent approaches to data abstractions, heirarchical structure, and program design. Abstract data types are an extension and modification of the traditional concept of data type. An abstract data type defines not only a data representation for objects of the type but also the set of operations that can be performed on objects of the type. Furthermore, the abstract data type can protect the data representation from direct access by other parts of the program.

Key Words: abstract data types, data type, type, data abstraction, program modifications, programming methods, class, cluster, module.

CR categories: 4.22, 4.34, 4.43

1. INTRODUCTION

Much recent literature on programming methods has advocated an approach which can be generally characterized as: Construct a simple version of the program, and then evolve it by stages until it is complete and efficient. This general approach takes many different forms such as:

1) Top-down Programming - Construct a top-level program using calls to as yet undefined modules; then repeat this construction for each of the undefined modules until no modules are still undefined. (Wirth [71], Mills [71], Dijkstra [72a].

2) Iterative enhancement - Write a complete program, but in a simple and possibly inefficient way; and then, once it is working, extend it until it accomplishes the entire task (Basili and Turner [75]), or rewrite parts of it to obtain acceptable efficiency (Mills [74]).

3) Very high level languages - Construct the program in a very high level language which makes the expression of the program simple, and then evolve the program to an efficient form (Cheatham and Wegbreit [72]). A related approach is to use a formal specification language and then later select an appropriate algorithmic implementation.

A basic problem common to all the above approaches is that the completion or modification of the program may be almost as difficult as starting over. All too often seemingly simple modifications or extensions require changes throughout the program, and the resulting program may bear little resemblance to the original. This problem is especially acute when data structures have to be modified.

Program structures that localize the effects of most program modifications or extensions would greatly enhance all of the above approaches toward staged development of programs. Such program structures would also relieve the problems of program maintenance;* and furthermore, they might encourage programmers to build on the work of others rather than reinventing the wheel whenever they need one that is slightly different.**

Author's address: Technology Building, Room A-265, National Bureau of Standards, Washington, D.C. 20234

* The distinction between program development and program maintenance is somewhat arbitrary--often it has more to do with the exhaustion of the development budget than it does with whether the program works.

** Reinvented wheels often turn out to have a flat side. Unfortunately, modification of another wheel may not lead to better results for the simplest way to modify the size of a wheel also results in a flat side.

The thesis of this paper is that if the concept of an abstract data type is used as the basic unit of modularity in a program, then modules can be more independent of each other, and the program is likely to be much easier to extend or modify. Section 2 provides a tutorial introduction to the concepts of abstract data types and motivates their use as units of modularity. Section 3 surveys existing implementations and uses of abstract data types. The remaining sections develop a simple example to illustrate the thesis. In the example, a program is evolved by stages in such a way that successive evolutions of the program do not destroy the clarity of the program structure developed in the first stage. The example also illustrates some of the features that should be included with abstract data types to make them more effective in localizing potential modifications and extensions of the program.

2. ABSTRACT DATA TYPES AS UNITS OF MODULARITY

It is not easy to modularize a program so that it can be modified easily. The modularity must satisfy three conflicting requirements:

1) The modules must be relatively independent, their interactions must be understandable, and there must be no unanticipated interactions between them. In short, each module must mind its own business.

2) The modules must be small. The previous requirement can easily be satisfied if the program is broken into only a few large modules. Large modules may be independent, but their size and complexity still make them hard to modify or extend.

3) The modularity must not cause serious inefficiencies. While modules should mind their own business, the modules should not be prevented from accomplishing their legitimate business in the most simple and straight forward way possible.

Structured programming concepts that deal with control flow constructs have improved the modularity of programs by reducing the complexity of the possible control paths through programs. Nevertheless, even with carefully structured control flow, interactions through common data structures can still make it impossible to predict what effects might result from a proposed modification to one module of a program. The problem is that different modules often make quite complex assumptions about the structure or content of common data objects, and modules behave in unpredictable ways if they are ever executed when these assumptions are not true. For example, in a data structure involving pointers, any change in the location of the pointer or any unexpected value of a pointer is likely to cause trouble. The assumptions that each module makes about the data structure are not easily deducible from the program text, and it is not always feasible to have modules check the consistency of a shared data structure before they use it.

2.1 The Concept of an Abstract Data Type.

For any complex data structure, the job of maintaining that structure and guaranteeing the consistency of the data stored there should not be distributed among all the programs that may need access to that data structure. One module should be responsible for maintaining the data structure and guaranteeing its consistency, and other modules should be forced to access the data through this module. This is a good general programming principle. The concept of an abstract data type provides a program structure that makes application of this principle routine--even for relatively small data structures. An abstract data type is a set of operations (procedures) that are grouped around common data structures. These data structures can be hidden from operations that are not defined as part of the abstract data type.

2.2 Data Abstraction.

The simplicity of this basic idea should not obscure its potential power and generality. Abstract data types provide a basis for data abstraction that is potentially as powerful as the concept of a procedure has been for procedural abstraction. Just as a procedure defines an abstract or high level operation that can be called without understanding the details of how the operation is implemented, so an abstract data type defines abstract or high level data objects that can be manipulated by calling operations on these objects without understanding the details of how the data is represented or how the operations are implemented.*

2.3 Generalization of Data Type and Procedure.

While the simple view of an abstract data type is to see it as providing a set of caretaker routines to manage data objects, the concept can also be viewed more broadly as a generalization of both a data type and a procedure. It is a generalization of a data type since its definition includes the declaration of data structures to represent objects of the abstract data type. It is also a generalization of a procedure since the operations of the abstract data type are usually procedures. A procedure can be thought of as a degenerate case of an abstract data type with one operation and a common data structure that is null. (The procedure can still have local variables.)

The general concept of an abstract data type can be used almost universally as the unit of modularity within a program. Indeed, the phrase "abstract data type" may eventually be replaced

---

* A data management system also makes it possible to manipulate data objects without having to worry about all the details of how that data is stored and represented. A data management system is an instance of an abstract data type; however, the importance of abstract data types arises from their use to implement smaller and more specific data objects.

by the simpler word "module"; however, before this happens, some of the connotations of the word "module" must be changed--in particular, modules must not be thought of as very large entities. There are also some undesirable connotations of the term "abstract data type." "Abstract" should not be equated with "esoteric" or "impractical"; and the potential generality of the idea may be missed if too much emphasis is put on the words "data type."

The word "abstract" in the phrase "abstract data type" distinguishes this concept from other concepts of type in current use. The word abstract is appropriate for two reasons:

1) The abstract data type can be used as an appropriate vehicle for data abstraction.

2) A type is thought of from the mathematical viewpoint that sees a new type as being defined by the definition of the operations that are applicable to objects of that type (Morris [73b]). From this viewpoint, the definition of the operations applicable to an object is more basic than the definition of its data representation.

Terminology--and the underlying concepts--in this subject area is still in a state of flux. The term "abstract data type" is a fairly general term, and it is now widely but not universally used. The following terms have all been used with an equivalent or similar meaning: class, cluster, form, extended type, opaque type, type, mode, module, abstract machine, virtual machine, and procedure. The reader is encouraged not to miss the forest because of all the trees.

Abstract data types should be especially useful for software systems that are primarily concenred with the management of data. The difficulty of changing data structures is a primary source of inflexibility in such systems, and the use of abstract data types promises to alleviate this rigidity. Furthermore, by improving the clarity and understandability of the software, abstract data types should contribute to improved reliability and security in such systems. Other papers on the use of abstract data types to achieve better program modularity include papers by Liskov and Zilles [74], Dennis [75], and Flon [75].

3. REALIZATIONS AND USES OF ABSTRACT DATA TYPES

While the basic idea of an abstract data type sounds simple, it is not so simple to make it effective and efficient in a wide variety of programming situations. Procedures would not be a very powerful and general tool for procedural abstraction if they could not be parameterized. To be an effective tool for data abstraction, abstract data types must also be parameterized; and furthermore, they must be able to handle many objects of the same type, and they should provide protection for the representations of their data objects.

3.1 Realizations.

The concept of an abstract data type was first implemented as a class in the language Simula 67 (Dahl et al, [68]). Other versions of this basic concept have been included in many recently proposed languages; for example, there are clusters in the language CLU (Liskov and Zilles [74]), forms in the language Alphard (Wulf [74]), classes in Concurrent Pascal (Brinch Hansen [75a]), and opaque types in PASQUAL (Tennent [75]). There are important differences in generality between these different realizations of abstract data type, but a detailed discussion of these languages is beyond the scope of this paper.

Abstract data types are closely related to extensible data types (Wegbreit [74]) in that both allow for the definition of new types and the subsequent creation of objects of that new type; however, the basic concept of an extensible data type does not include the definition of all the visable operations on objects of the type in a way that may be quite independent of the data representation.* Abstract data types should make it possible to change the data representation of objects while guaranteeing that all visible operations on the objects have the same properties.

When implementing abstract data types it is difficult to make manipulation of the data objects efficient while still guaranteeing that other parts of the program are not dependent on details of the implementation of the data objects. This general problem has been called the uniform reference problem by Geschke and Mitchell [75], and it is also discussed by Parnas et al. [75]. Repeated simple operations on a data object often cannot be implemented as individual procedure calls if an acceptable level of efficiency is to be maintained. The repeated procedure calls can be avoided either by implementing the repetition inside an operation of the type (as is done for the example in Section 5) or by using macros instead of procedures.

Even without new structures in a programming language, procedural abstraction can be used to obtain much of the effect of abstract data types. A data structure can be defined and a set of procedures can be written to manipulate that data. If the rest of the program always accesses the data through those procedures, then a reasonable form of data abstraction is achieved. The primary limitations of this approach are: there is no enforcement of access controls other than the self discipline of the programmers, and the syntax may not make the program as easy to understand. Aiello [74] discussed approaches toward implementing abstract data types in PLI, Pascal, EL1, and Simula. While he found problems with all of these

* Types are called modes in the language EL1. Wegbreit [74] does discuss programmer specified mode behavior which allows EL1 to be used to implement the concept of an abstract data type. Recent evolution of the EL1 mode has made it more suitable for realizing abstract data types (Squires [75]).

languages for implementing the full concept of an abstract data type with protection, many of the benefits of data abstraction can be achieved in any language.

3.2 Protection.

Protection to support abstract data types can be enforced either by a compiler or by an operating system. Both of these approaches to protection in association with abstract data types are being pursued.

Compilers can be used to protect the data representation within a type from direct access by procedures that are not defined as part of the same abstract data type.* They can also be used to enforce controls over the rights of other parts of the program to invoke the operations of a type. Protection in programming languages has been emphasized especially in the work of Morris [73a], Liskov and Zilles [74], and Wulf [74]. Protection is especially useful in large programs where explicit definition and control of access can be used to identify the modules that could be affected by a proposed change.

Protection for abstract data types can also be implemented by an operating system--especially if appropriate hardware support is available. The HYDRA operating system (Wulf et al. [74]), the Plessey System 250 (Cosserat [74]), and a system design by Neumann et al. [75] all support the concept of an abstract data type. The usefulness of integrating system protection mechanisms with abstract data types (also called extended types) in order to enhance system security is discussed in Linden [76].

3.3 Experience.

The system programming area has been the source of most of the experience in using abstract data types. This experience has emphasized the use of abstract data types to implement general system objects such as stacks, buffers, tables, directories, etc.; however, the concept appears to be at least as useful for implementing data objects that are tailored to specific applications. The use of an abstract data type to create and protect applications-oriented objects in a bibliographic system is described in Wulf et al. [74].

As another example, objects of type "bank account record" might be implemented with an abstract data type that provides operations such as withdrawal, deposit, and address change. While these operations appear simple to other programs that call them, the operations might actually incorporate elaborate provisions to check for the reasonableness and consistency of the transaction and to guarantee that appropriate records of the transaction are preserved. If there is a guarantee that the bank account records can only be updated by calling one of these procedures, then it would be much easier to extend or modify the program, and also security and reliability would be greatly improved.

* Protection is not enforced for classes as implemented in most versions of Simila.

Data abstraction and the isolation and protection of the data representation are most important when:

1) the data has a complex structure,
2) the data might get into an inconsistent state, or
3) the data needs to be protected for reasons of privacy or security.

An abstract data type is usually not needed to encapsulate a simple array of ordinary data values. Note, however, that an array is sometimes used as the storage medium for data that is expected to have a much more complex structure. The abstract data type is able to implement and enforce this additional structure. The abstract data type is especially useful when some redundancy is incorporated in the data for then much of the error detection and error recovery can be buried in the operations of the abstract data type.

3.4 Hierarchical Structure.

Abstract data types seem to be emerging as the appropriate unit of modularity for use in systems to be structured according to Dijkstra's idea of levels of abstraction (Dijkstra [72a] [72b]). In hierarchally structured systems a simple procedure is often awkward to use as the unit of modularity because:

1) The Algol scope of variables rule is wrong when procedures are used as the unit of modularity in a hierarchical structure. Variables generally do not need to be global across different levels of a hierarchy. Modules in a hierarchy should not automatically have access to variables declared at higher levels of the hierarchy.

2) A procedure cannot easily preserve information between successive calls, and thus it cannot be used as a unit of modularity to encapsulate a data base by forcing all accesses to the data to go through the procedure.

3) A unit of modularity often needs many entry points. It is awkward to use a parameter to obtain the effect of many entry points.

Procedures and data interact very closely, and an appropriate unit of modularity should incorporate both procedure and data definitions. An abstract data type does this, and it has the following three desirable properties for a unit of modularity in a hierarchically structured system. These three properties correspond (in reverse order) to the three problems with using a procedure as the unit of modularity:

1) An abstract data type includes many operations or entry points.

2) Objects of an abstract data type are preserved independent of calls to operations of the type.

3) The representation of a data object is normally not accessible except by the operations defined as part of its type definition.

Large units of modularity have always had the first two of these properties. The class concept from SIMLUA extended these two properties to small units of modularity by providing programming language support for them. The third property adds protection to these units of modularity so that interactions between modules can be explicitly defined and controlled.

3.5 Program Design.

An important advantage of abstract data types is that they can be used to document decisions made during the design stages of a programming project. One proposed method for programming suggests that the first stage of program design should result in a decomposition of the program into a set of modules with each module defining an abstract data type (or an abstract machine). The first stage of the design is complete when all the operations of each of the abstract data types have been identified. (Neumann [74b], Robinson et al. [75]). The next stage of the program design consists of some form of specification of the operations of each abstract data type. The decomposition of a program into modules in the form of abstract data types makes formal specification of the modules much more feasible (Liskov and Zilles [75], Robinson et al. [75]).

The use of abstract data types during the stages of program design and specification has substantial similarities with the approach to program specifications advocated by Parnas [72a] [72b] [72c]. Parnas suggests that a module can be defined in terms of a set of operations and a set of value functions supported by the module. (Value functions return a value but do not change the state of an object.) The module is defined abstractly by defining all error conditions and the effect of all the operations on the value-returning functions. Defining a module in this way requires no knowledge of the data representation of an object's state and hides the object's state from other modules.

A variant of Parnas's approach has been used to specify a large, hierarchically structured operating system. (Neumann [74a], Robinson et al. [75], Neumann et al. [75]). It seems to be an appropriate vehicle for defining and specifying early design decision--especially since the approach makes it easy to modify the design with a minimum of effort.

A gradual evolution toward programming languages that incorporate abstract data types may be possible based on the prior use of abstract data types for program design and specification. Abstract data types can be used to design and structure a program even if coding must eventually be done in a language that does not support abstract data types. If abstract data types prove to be effective for program design, then the new concepts can be gradually introduced to the community of programmers. Program design is usually done by the better and more flexible programmers. Evenually it would become clear that a programming language that supports the design structures would be worth the costs of converting to a new language.

3.6. Summary

The concept of an abstract data type is still the subject of intense research. Languages that implement some version of it are not widely available, and the search for the most effective version of the idea is still going on. There are still serious questions about the effectiveness and efficiency of the more general versions of the idea. Nevertheless, there has now been enough successful use of some versions of this concept that it appears ready for experimental usage outside the research community. It should be possible to profit from many aspects of the idea even if coding must be done in a language that does not implement abstract data types. The reader will also note that some aspects of this approach have long been recognized as good programming practice. The newer aspects of the approach involve reducing the size of the unit of modularity and including protection featues to enforce the desired modularity.

4. THE SIEVE OF ERATOSTHENES

While many programming techniques work best on the small examples which are appropriate for a published paper, the modularity provided by abstract data types promises to be more effective in large programs than it is in this small example. The reader must try to generalize the ideas from this example to larger programs.

The example is a program to compute prime numbers based on an algorithm known as the sieve of Eratosthenes. This program is interesting because it has already been used by Hoare [72] to illustrate similar points about the stated development of a program. If abstract data types can simplify the structure of this program and preserve that structure during refinements, then it will be clear that the improvement is due to the use of the abstract data types; it is not due to differences between programmers.

Wulf [74] used this same example in an informal publication on the programming language ALPHARD. He used the example to illustrate how the program's structure could be improved using the Alphard concept of a form. (For the purposes of this paper, Alphard forms can be considered the same as abstract data types.) Wulf also claimed that forms make it easier to prove correctness of the program; and the paper illustrated this with an informal proof of correctness. Unfortunately, the program was not correct. Furthermore, the mistake arose because a subtle, unstated assumption needed about a data representation was not fulfilled by declarations made at a higher level of abstraction.* To be fair, it should be noted

* I found the error while reading the proof and wondered why there was no discussion of this apparently necessary assumption. While a simple fix was possible, the mistake indicated that the program's structure was not as clear and general as desired. Part of the reason the mistake occurred is that Wulf was using the program to illustrate various features of the ALPHARD language, and this resulted in too much of a good thing. Wulf's program defined five abstract

that this mistake occurred at a time when proof methods appropriate for use with abstract data types were just beginning to be worked out.

The sieve of Eratosthenes finds all prime numbers up to a number N by beginning with the set of all numbers up to N and removing all the numbers which are not prime as follows:

1) Remove 0 and 1.
2) For each integer from 2 through the square root of N, remove all multiples of that integer from the set.

The numbers that remain are the primes.

In step 2 it is not necessary to remove the multiples of any number which has already been removed from the set. This leads to the initial program as given in Figure 1. The right column is a natural language version of the program that is sufficiently self-contained so that it can be read by itself. The formal language in the left column uses notation that is fairly common in many recent Algol-based languages. The right column serves to explain any notation that may be unfamiliar, and it also serves as comments on the program.

This program is a direct translation of the English definition of the algorithm. The only operations in the program that are not directly executable are the set operations such as "delete," "full," "print," and test for membership "in." The program library for any language that supports abstract data types would surely contain a module that supports these set operations. With the help of this library module, the above program is now executable.

A program library could not easily define the set operations as separate procedures since the operations must all operate on a common data representation. Abstract data types are more convenient units for building program libraries since the abstract types can encapsulate data structures that otherwise would have had to be global. The abstract types exhibit a more easily defined external interface. This means that program libraries could contain many abstract data types that could then be used as building blocks to get an initial version of a program running with relatively little effort.

A set module chosen from a program library might be prohibitively inefficient to support this algorithm. For example, a library module might represent membership in the set by storing the values of all members of the set. If this algorithm is to be used to generate reasonably large prime numbers, then the set should be represented as a bit string. Such a set module might also be available from a library. If not, then such a module might be constructed as a useful, general-purpose byproduct of the programming effort. The parts of such a module that are relevant to primefinder are presented in Figure 2. This module only handles initial sets of integers, but it could easily be generalized to handle any indexed sets.

The set module is parameterized by an integer so that it can be used to implement any set as long as the universe of possible values is an initial set of integers (0 through N). The common data structures of the module consist of an integer constant (Asize) and an array of words (A). These data structures are created each time a variable is declared to be of the type set.* All operations of the module have a parameter (s) that is used to identify the particular set to be operated on. Within the operations, the data structure associated with the appropriate set is selected by the selector notation s.Asize or s.A where s is the parameter that identifies the set.

The procedure primfinder never creates more than one object of type set. Since this is true, it would not be necessary to use a parameter to identify the set to be operated on by each operation. Elimination of the parameter would simplify the notation; however, in this example, the parameter is retained to illustrate how to handle the more general case where several distinct objects of type set might be created.

In many cases it is useful to define explicit operations for creating and deleting objects of the type. This is especially useful when a newly created object must always be initialized to some consistent state that is not achieved by simple variable declarations. For example, in the set module, the last word of the array A usually contains bits that would represent integers larger than N. It might be desired that these bits always be set to zero when a new set is created. This can be done by a create operation that is invoked each time a new object is declared to be of type set. In this example, the problem of handling the extra bits in the last word is avoided by writing the other operations to guarantee that their value is never disclosed.

The syntax of the type definition follows Parnas [72b] and distinguishes between value functions which return information but do not change the object, and operations which do change the object. "Words" is assumed to be a primitive or previously defined type that is treated as an array of bits indexed by integers from 0 through wordlength-1.

5. STRUCTURE-PRESERVING EVOLUTION OF THE PROGRAM

Hoare [72] approached the construction of Primefinder by beginning with a simple expression of the algorithm which is equivalent to the one

---

data types. If the program is written with only a single abstract data type to support the basic algorithm, then the program has an especially clear structure that can be preserved through a variety of optimizations.

* If the language includes reference variables so that there is no guarantee of a one-to-one correspondence between variables of the type set and objects of type set, then it would be necessary to distinguish declaration of the variable from creation of the object (see Liskov and Zilles [75]).

```
Procedure primefinder (N:int);
  Declare next  : int := 2;
  Declare sqrootN : int = int(sqroot(N));
  Declare sieve : set 0..N;
  full(sieve);
  delete(0,sieve);delete(1,sieve);

  while next <= sqrootN do
    begin
      for mult from 2*next by next to N do
        delete(mult,sieve);
      repeat next := next + 1
        until in(next,sieve)
    end while;

  print (sieve);
  end primefinder;
```

next is an integer initialized to 2.
sqrootN is a constant with obvious value.
sieve is a set of integers over the range
  0 to N and is initialized to be full.
0 and 1 are removed from the sieve.

While next <= sqrootN repeat steps
  1) and 2):
1) Remove all multiples of next from sieve
   beginning with 2 x next.
2) Increment next until it has a value in
   the sieve.

Print the numbers that remain in the sieve.

Figure 1 - Initial Version of Primefinder

```
Type set (N:int) =

  Declare Asize : int = N ÷ wordsize;

  Declare A : array[0..Asize] of words;

  Value function in (x:int,s:set) returns
          Boolean;
    If x < 0 or x > N then errorcall;
    If s.A[x ÷ wordsize][x mod wordsize] = 1
      then return TRUE
      else return FALSE
    end in;

  Operation delete (x:int,s:set);
    If x < 0 or x > N then errorcall;
    s.A[x ÷ wordsize][x mod wordsize] := 0
    end delete;

  Operation full (s:set);
    for word from 0 to Asize do
      s.A[word] := 1
    end full;

  Value function print (s:set);
    for x from 0 to N do
      If in(x,s) then print(x)
    end print;

  end set;
```

set is a new type parameterized by an
  integer that defines the largest integer
  in the universe.
Asize is the constant length for the storage
  array (defined for each set variable);
  wordsize is the wordlength of the target
  machine.
A is the array of words used to store sets;
  words is assumed to be primitive--an array
  of bits indexed by 0 through wordsize - 1.
in tests whether the integer x is a
  member of the set s.
x is checked for expected bounds.
s.A is the array that represents the set s.
  It is indexed by a word index and a bit
  index. If the appropriate bit is 1,
  then return true, else return false.

delete removes the integer x from the set s.
If x is within bounds, then s.A is indexed
  by the appropriate word and bit indices
  and this bit is set to 0.

full turns s into the universal set.
All bits of all words s.A[word] are set
  to 1.

print lists the integers that belong to s
  by trying all possible integers to see if
  they are in s.

Figure 2 - The Type "set"

given in Figures 1 and 2.* The criteria for efficiency that Hoare imposes is that divisions should be eliminated as the way of translating from integers to the indices of the bit which represents whether that integer is in the set. Hoare evolves his program to meet this criteria, but by the time he arrives at the final version of his program (using a language without abstract data types), the simple structure of the algorithm has been badly obscured.

In this section an equivalent program that is structured using an abstract data type is evolved to meet the same criteria for efficiency without obscuring the structure of the original program.

In the program in Figures 1 and 2, each call to the operations "in" and "delete" requires division and modulo operations to translate from the integer parameter to the indices of the bit associated with it. These calls can be eliminated from the inner loops by adding two new operations to the type definition. These new operations are designed to provide more efficient support for this particular algorithm. If the type definition is viewed as defining an abstract machine that supports the execution of the algorithm, then these additions can be seen as tailoring the abstract machine to provide more effective support for a specific application. In this case the two inner loops in the basic program in Figure 1 can be replaced with calls to the functions in Figure 3 which are to be added to the type definition. Note that these two new operations eliminate repeated calls to operations of the type by moving the iteration inside the type definition.

Now that all divisions have been eliminated from the inner loops, one would probably be content with its efficiency. Further improvements should probably be concerned about storage requirements rather than about the execution time; however, it is instructive to see what happens when Hoare's original goal is pursued.

The next step in the program's evolution involves maintaining the value of next in two forms (next and xindex). Next is the integer whose multiples are being removed from the sieve. Within the type definition, xindex is used to store the indices of the bit which represents the current value of next. The translation from next to xindex that was done on each call can now be transformed into an assertion that they bear the proper relationship. This illustrates a suggestion of Good [74] that inefficient functions should be pushed out of the code and into assertions about the program.

This step of the evolution is accomplished as follows:

1) Make xindex global to the type definition and initialize it to (0,2).
2) In removemult and nextmember, change the initial assignments to xindex into assertions.

The final version of the program is given in Figures 4 and 5. In the final version, the print operation is also rewritten so it does not call the "in" operation. A more specific name is now given to the type since there is no longer any intention of using it in any other program. Nevertheless, program generality is preserved because previous versions of this type were more general-purpose.

Other modifications are possible to improve the performance of this algorithm. For example, the number 2 could be treated as a special case so that storage would only be allocated for the odd integers at the beginning. This modification changes the correspondence between integers and the bit positions that represent them; nevertheless, this change could be made with only a limited number of modifications to the sieveset type definition.*

In all, four modifications to the basic program in Fegure 1 have now been suggested:

1) Represent the sets as bit strings.
2) Remove divisions from the inner loops.
3) Remove divisions from all loops.
4) Eliminate the even numbers from ever being considered.

Two of these modifications (the first and last) involved changes in the data representation; yet a program using an abstract data type to support the set operations can be evolved through all of these modifications without obscuring the basic structure of the algorithm.

A program structure based on abstract data types also simplifies a proof about properties of the program. While no proof has been attempted in this paper, the above program structure makes a proof relatively simple. Properties of the operations of the abstract data type can be proven individually, and modifications to the program would not require a completely new proof. The use of abstract data types appears to make the complexity of a proof grow in a way which is more nearly linear with the size and complexity of the program.

6. CONCLUSION

A simple program has been structured using an abstract data type. Various modifications to improve the performance of the program have resulted in changes to the abstract data type, but the modifications have not obscured the structure

---

* The program primefinder in Hoare [72] is based on a slightly different version of the sieve of Eratosthenes--when a prime number is found it is moved into a different set. This doubles the storage requirement, and it means that iteration through the set cannot be terminated with the square root of N; however, this difference has no significant effect on the complexity of the program.

* It is even possible to eliminate 3 and its multiples from the initial set, but in this case, the removemultiples operations becomes significantly more complex.

Remove all multiples of next

```
Operation removemults(x:int,s:set);

  If x <= 0 then errorcall;
  Declare xindex : pair[word,bit:int] =
            [x ÷wordsize,x mod wordsize];

  Declare mindex : pair[word,bit:int] :=
                   xindex;
  while TRUE do
    begin
      mindex := mindex + xindex;
      If mindex.bit >= wordsize then
        begin mindex.word := mindex.word + 1;
          mindex.bit := mindex.bit - wordsize
        end;
      exit if mindex.word >= s.Asize;

      s.A[mindex.word][mindex.bit] := 0;

    end while
  end removemults;
```

removemults removes all multiples of x from s beginning with 2x, 3x, etc.
Errorcall if x=0 (or x<0).
xindex (the indices of x) is a constant pair; xindex.word is the word index of x and xindex.bit is the bit index of x.
mindex (the indices of a multiple of x) is a variable pair and is initialized equal to xindex

mindex is incremented to be the indices of the next multiple of x. Addition of pairs is assumed. If mindex.bit gets too big, an adjustment is made.

exit the loop when mindex represents an integer somewhat larger than N.
The multiple of x is removed from the set by setting the bit indexed by mindex to 0.
Repeat the last three steps until the exit occurs.

Find the next member in s

```
Value function  nextmember(x:int,s:set)
          returns int;

  Declare xindex : pair[word,bit:int] :=
            [x ÷ wordsize,x mod wordsize];
  Declare newx : int := x;

  repeat
    newx := newx + 1;
    xindex.bit := xindex.bit + 1;
    If xindex.bit >= wordsize then
      begin xindex.word := xindex.word + 1;
        xindex.bit := 0
      end;
    until s.A[xindex.word][xindex.bit]
          = 1 or xindex.word > s.Asize;
  return newx
  end nextmember;
```

nextmember returns the first integer larger than x which is in s.

xindex is a variable pair that is otherwise the same as above.
newx is the value to be returned, it is initialized to be x.

newx is incremented
xindex is incremented and adjusted if xindex.bit gets too big.

Repeat the two previous steps unless newx is in the set s or xindex.word gets too big. Primefinder should never cause the latter to happen.
Return newx.

Figure 3 - Additions to the Type "set"

of the program. If the abstract data type is viewed as providing an abstract machine for the execution of the program, then all of these modifications can be seen as changes made to the abstract machine to provide better support for this particular algorithm.

The concept of an abstract data type is needed so that the program modifications do not obscure or destroy the program structure. In particular, the data types must be user-definable, and the basic operations to be performed on the data must be defined as part of the data type definition. Furthermore, the representation of data inside an abstract data type should not necessarily be visable outside the data type--and for large programs these visibility restrictions should be explicitly defined and enforced by a protection mechanism.

Acknowledgments. The author is grateful to Stuart W. Katzke, whose careful readings of this paper, have improved its clarity and accuracy. Henry F. Ledgard's suggestions provided a useful orientation for revising the paper.

REFERENCES

Aiello [74] Aiello, J.M. An investigation of current language support for the data requirements of structured programming. Report No. MAC TM-51 (CSG-105), Project Mac, MIT (September, 1974).

Basili and Turner [75] Basila, V.R., and Turner, A.J. Iterative enhancement: A practical technique for software development. First National Conf. on Software Engineering, IEEE Computer Society, Washington, D.C. (Sept. 1975)

Brinch Hansen [75a] Brinch Hansen, P. The Programming Language Concurrent Pascal. IEEE Trans. of Software Engineering 1, 2 (June 1975).

Cheatham and Wegbreit [72] Cheatham, T.E., and Wegbright, B. A laboratory for the study of automating programming, AFIPS Conf. Proc., SJCC 1972, Vol. 40, pp. 11-21.

Cosserat [74] Cosserat, D.C. A data model based on the capability protection mechanism. IRIA Internat. Workshop on Protection in Operating Systems, Rocquencourt, France (August 1974) pp. 35-54.

Dahl et al. [68] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. The Simula 67 common base language, Norwegian Computing Center, Oslo, 1968.

Dennis [75] Dennis, J.B. An example of programming with abstract data types, SIGPLAN notices, 10, 7 (July 1975).

Dijkstra [72a] Dijkstra, E.W. Notes on structured programming, Structured Programming, by O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, 1972.

```
Procedure primefinder (N:int);
  Declare next  : int := 2;
  Declare sqrootN : int = int(sqroot(N));
  Declare sieve : sieveset(N);
  full(sieve);
  delete(0,sieve);delete(1,sieve);

  while next <= sqrootN do
    begin
      removemults (next,sieve);

      next := nextmember(next,sieve)
    end while;

  print (sieve)
  end primefinder;
```

next is an integer initialized to 2.
sqrootN is a constant with obvious value.
sieve is a set over integers over the range 0 to N and is initialized to be full.
0 and 1 are removed from the sieve.

While next <= sqrootN repeat steps 1) & 2):

1) Remove all multiples of next from sieve beginning with 2 x next.
2) Increment next until it has a value in the sieve

Print the numbers that remain in the sieve.

Figure 4 - Final Version of Primefinder

```
Type sieveset (N:int) =

Declare Asize : int = N ÷ wordsize;


Declare A : array[0..Asize] of words;


Declare xindex : pair[word,bit:int] :=
                    [0,2];



Operation removemults (x:int,s:sieveset);
  If x <= 0 then errorcall;
  Assert x = (s.xindex.word * wordsize)
             + s.xindex.bit;
  Declare mindex : pair[word,bit:int] :=
                     s.xindex;
  while TRUE do
    begin
      mindex := mindex + s.xindex;
      If mindex.bit >= wordsize then
        begin
          mindex.word := mindex.word + 1;
          mindex.bit := mindex.bit - wordsize
        end;
      exit if mindex.word >= s.Asize;

      s.A[mindex.word][mindex.bit] := 0

    end while
  end removemults;



Value function nextmember (x:int,
      s:sieveset) returns int;
  Assert x = (s.xindex.word * wordsize)
             + s.xindex.bit;
  Declare newx : int := x;
  repeat
    newx := newx + 1;
    s.xindex.bit := s.xindex.bit + 1;
    If s.xindex.bit >= wordsize then
      begin
        s.xindex.word := s.xindex.word + 1;
        s.xindex.bit := 0
      end
    until s.A[s.xindex.word][s.xindex.bit]
          = 1 or s.xindex.word > s.Asize;
  return newx
  end nextmember;



Value function  print (s:sieveset);
  Declare xval : int;
  for word from 0 to Asize do
    for bit from 0 to wordsize - 1 do
      begin
        xval := (word * wordsize) + bit;
        If xval <= N and s.A[word][bit] = 1
          then print(xval)
      end
  end print;

Operation  delete (x:int,s:sieveset);
Operation  full (s:sieveset)
end sieveset;
```

sieveset is a new type parameterized by an integer that defines the largest integer in the universe.

Asize is the constant length for the storage array (defined for each set variable); wordsize is the wordlength of the target machine.

A is the array of words used to store sets; words is assumed to be primitive--an array of bits indexed by 0 through wordsize - 1.

xindex (indices of x) is a pair initially [0,2]; it is a pointer to a location within the sets; s.xindex.word is the word index of x in s, and s.xindex.bit is the bit index of x in s.

removemults removes all multiples of x from s beginning with 2x, 3x, etc.

Errorcall if x=0 (or x<0). Assert that x has the expected relation to xindex.

mindex (the indices of a multiple of x) is a variable pair and is initialized equal to xindex.

mindex is incremented to be the indices of the next multiple of x. Addition of pairs is assumed. If mindex.bit gets too big, an adjustment is made.

Exit the loop when mindex represents an integer somewhat larger than N.

The multiple of x is removed from the set by setting the bit indexed by mindex to 0.

Repeat the last three steps until the exit occurs.

nextmember returns the first integer larger than x which is in s.

Assert that x has the expected relation to xindex.

newx is the value to be returned, it is initialized to be x.

Newx is incremented.

xindex is incremented and adjusted if xindex.bit gets too big.

Repeat the two previous steps unless newx is in the set s or xindex.word gets too big. Primefinder should never cause the latter to happen.

Return newx.

print lists the integers that belong to s by iterating through the word and bits indices, computing xval, the integer value corresponding to the indices, and printing it only if the indices indicate it belongs to s and it is N.

delete and full are used only for initialization and are the same as in Figure 2.

Figure 5 - Final Version of "sieveset"

Dijkstra [72b] Dijkstra, E.W. The humble programmer, Comm. of the ACM, 15, 10 (Oct. 1972).

Flon [75] Flon, L. Program design with abstract data types, Dept. of Computer Science, Carnegie Mellon Univ. (June 1975).

Geschke and Mitchell [75] Geschke, C.M. and Mitchell, J.G. On the problem of uniform references to data structures. International Conf. on Reliable Software, SIGPLAN Notices, 10, 6 (June 1975) pp. 31-42.

Good [74] Good, D.I. Private communication, 1974.

Hoare [72] Hoare, C.A.R. Notes on data structureing. Structured Programming by O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, 1972.

Linden [76] Linden, T.A. Operating system structures to support reliable and secure software. In preparation (1976).

Liskov and Zilles [74] Liskov, B. and Zilles, S. An approach to abstraction. Proc. of a Symposium on Very High Level Languages, SIGPLAN Notices, 9, 4 (April 1974).

Liskov and Zilles [75] Liskov, B. and Zilles, S. Specification techniques for data abstractions. International Conf. on Reliable Software, SIGPLAN Notices, 10, 6, (June 1975) pp. 72-85.

Mills [71] Mills, H.D. Top down programming in large systems. Debugging Techniques in Large Systems, Prentice-Hall, 1971, pp. 41-55.

Mills [74] Mills, H.D. Techniques for the specification and design of complex programs, Proc. of the Third Texas Conf. on Computing Systems, U. of Texas (Nov. 1974) pp 8-1-1 - 8-1-4.

Morris [73a] Morris, J.H. Protection in programming languages. Comm. of the ACM, 16, 1 (Jan. 1973) pp. 15-21.

Morris [73b] Morris, J.H. Types are not sets. ACM Symposium on Principles of Programming Languages, Boston (1973) pp. 120-124.

Neumann et al [74a] Neumann, P.G., Fabry, R.S., Levitt, K.M., Robinson, L., Wensley, J.H. On the design of a provably secure operating system. IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France (August 1974) pp. 161-176.

Neumann [74b] Neumann, P.G. Towards a methodology for designing large systems and verifying their properties. Gesellschaft fur Informatik, Berlin, Oct. 9-12, 1974.

Neumann et al. [75] Neumann, P.G., Robinson, L., Levitt, K.N., Boyer, R.S. and Saxena, A.R. A provably secure operating system. SRI Report, Menlo Park (June 1975).

Parnas [72a] Parnas, D.L. A technique for software module specification with examples. Comm. of the ACM, 15, 5 (May 1972) pp. 330-336.

Parnas [72b] Parnas, D.L. Some conclusions from an experiment in software engineering techniques. AFIPS Conf. Proc. 1972 FJCC, pp. 325-329.

Parnas [72c] Parnas, D.L. On the criteria to be used in decomposing systems in modules. Comm. of the ACM, 15, 12 (Dec. 1972) pp. 1053-1058.

Parnas et al. [75] Parnas, D.L, Shore, J.E., and Elliott, W.D. On the need for fewer restrictions in changing compile-time environments. SIGPLAN Notices, 10, 5 (May 1975) pp. 29-36.

Robinson et al. [75] Robinson, J., Levitt, K.N., Neumann, P.G., Saxena, A.R. On attaining reliable software for a secure operating system. International Conf. on Reliable Software. SIGPLAN Notices, 10, 6 (June 1975) pp. 267-284.

Squires [75] Squires, S. private communication, 1975.

Tennent [75] Tennent, R.D., PASQUAL: A proposed generalization of PADCAL. Tech. Rpt. 75-32, Dept. of Computing and Info. Sc., Queens University, Kingston, Ontario.

Wegbreit [74] Wegbreit, B. The treatment of data types in EL1. Comm. of the ACM, 17, 5 (May 1974).

Wirth [71] Wirth, N. Program development by stepwise refinement, Comm. of the ACM, 14, 4 (Apr. 1971) pp. 221-227.

Wulf [74] Wulf, W.A., ALPHARD: Toward a language to support structured programs, Carnegie-Mellon University, 1973.

Wulf et al. [74] Wulf, W.A., et al. HYDRA: The kernel of a multiprocessor operating system. Comm. of the ACM, 17, 6 (June 1974).