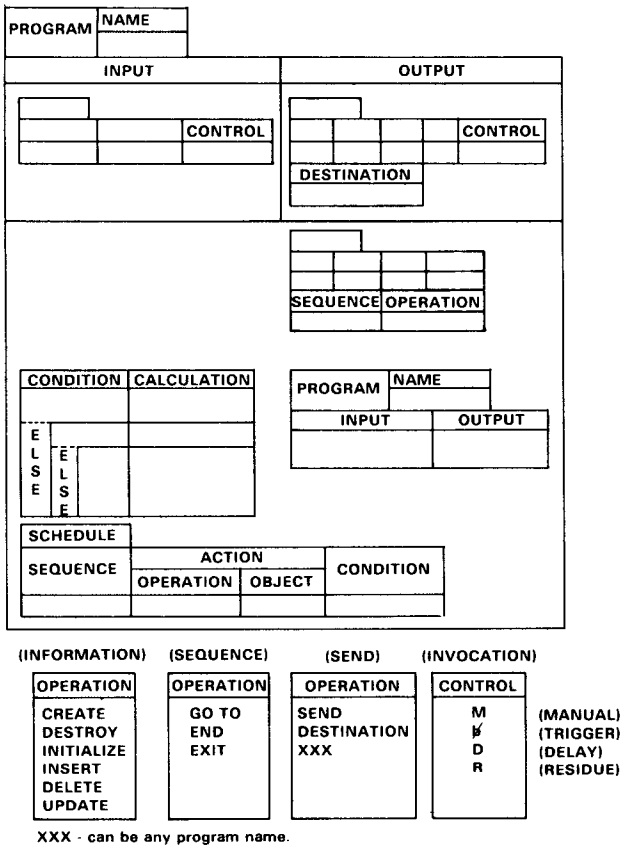
5. Thomas, J.C., and Gould, J.D. A psychological study of query-by-example. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 439-445. (IBM Res. Rep. RC-5124, Nov. 1974).
6. Zloof, M.M. Query-by-example. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 431-438 (IBM Res. Rep. RC-4917, July 1974).
7. Zloof, M.M. Query-by-example: A data base management language. IBM Res. Rep., Dec. 1976; available upon request from Moshé Zloof, T.J. Watson Res. Ctr., Yorktown Heights, N.Y.

Appendix A. SBA Programming Language Summary



Appendix B

In this section we illustrate how typical Query-by-Example queries and SBA programs can be mapped into predicate-calculus-like expressions. The examples chosen are Q1, Q2 from Section 2 and program 4 from Section 4.

Q1: {X: ∃Y(X,RED,Y)∈TYPE}
Q2: {X: ∃Y((X,Y)∈SALES ∧
(Y,PARKER)∈SUPPLY)}

Program 4: {(X,Y): (X,Y)∈CREDIT DECISION
∧∃Z∃W∃U
((U,Z)∈CREDIT RATING
∧ (X,U,W)∈ORDER)
∧ IF (Z = A1 ∧ W <10000)
→ Y = YES}

The formal syntax of the Query-by-Example database language is found in [7].

Data: Abstraction, Definition, and Structure
B. Wegbreit Editor

Abstract Data Types and the Development of Data Structures

John Guttag
University of Southern California

Abstract data types can play a significant role in the development of software that is reliable, efficient, and flexible. This paper presents and discusses the application of an algebraic technique for the specification of abstract data types. Among the examples presented is a top-down development of a symbol table for a block structured language; a discussion of the proof of its correctness is given. The paper also contains a brief discussion of the problems involved in constructing algebraic specifications that are both consistent and complete.

Key Words and Phrases: abstract data type, correctness proof, data type, data structure, specification, software specification
CR Categories: 4.34, 5.24

This work was supported in part by the National Science Foundation under grant number MCS76-06089.
A version of this paper was presented as the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure, Salt Lake City, Utah, March 22-24, 1976.

Author's address: Computer Science Department, University of Southern California, Los Angeles, CA 90007.

1. Introduction

Dijkstra [4] and many others have made the point that the amount of complexity that the human mind can cope with at any instant in time is considerably less than that embodied in much of the software that one might wish to build. Thus the key problem in the design and implementation of large software systems is reducing the amount of complexity or detail that must be considered at any one time. One way to do this is via the process of abstraction.

One of the most significant aids to abstraction used in programming is the self-contained subroutine. At the point where one decides to invoke a subroutine, one can (and most often should) treat it as a "black box." It performs a specific arbitrarily abstract function by means of an unprescribed algorithm. Thus, at the level where it is invoked, it separates the relevant detail of "what" from the irrelevant detail of "how." Similarly, at the level where it is implemented, it is usually unnecessary to complicate the "how" by considering the "why," i.e. the exact reasons for invoking a subroutine often need not be of concern to its implementor. By nesting subroutines, one may develop a hierarchy of abstractions.

Unfortunately, the nature of the abstractions that may be conveniently achieved through the use of subroutines is limited. Subroutines, while well suited to the description of abstract events (operations), are not particularly well suited to the description of abstract objects. This is a serious drawback, for in a great many applications the complexity of the data objects to be manipulated contributes substantially to the overall complexity of the problem.

2. The Abstraction of Data

The large knot of complexly interrelated attributes associated with a data object may be separated according to the nature of the information that the attributes convey regarding the data objects that they qualify. Two kinds of attributes, each of which may be studied in isolation, are:

(1) those that describe the representation of objects and the implementations of the operations associated with them in terms of other objects and operations, e.g. in terms of a physical store and a processor's order code;

(2) those that specify the names and define the abstract meanings of the operations associated with an object. Though these two kinds of attributes are in practice highly interdependent, they represent logically independent concepts.

The emphasis in this paper is on the second kind of attribute, i.e. on the specification of the operations associated with classes of data objects. At most points in a program one is concerned solely with the behavioral characteristics of a data object. One is interested in what one can do with it, not in how the various operations on it are implemented. The analogy with a closed procedure is exact. More often than not, one need be no more concerned with the underlying representation of the object being operated on than one is with the algorithm used to implement an invoked procedure.

If at a given level of refinement one is interested only in the behavioral characteristics of certain data objects, then any attempt to abstract data must be based upon those characteristics, and only those characteristics. The introduction of other attributes, e.g. a representation, can only serve to cloud the relevant issues. We use the term "abstract data type" to refer to a class of objects defined by a representation-independent specification.

The class construct of SIMULA 67 [3] has been used as the starting point for much of the more recent work on embedding abstract types in programming languages, e.g. [14, 16, 18]. While each of these offers a mechanism for binding together the operations and storage structures representing a type, they offer no representation-independent means for specifying the behavior of the operations. The only representation-independent information that one can supply are the domains and ranges of the various operations. One could, for example, define a type Queue (of Items) with the operations

NEW:	$\rightarrow$ Queue
ADD:	Queue $\times$ Item $\rightarrow$ Queue
FRONT:	Queue $\rightarrow$ Item
REMOVE:	Queue $\rightarrow$ Queue
IS_EMPTY?:	Queue $\rightarrow$ Boolean

Unfortunately, however, short of supplying a representation, the only mechanism for denoting what these operations "mean" is a judicious choice of names. Except for intuitions about the meaning of such words as Queue and FRONT, the operations might just as easily be defining type Stack as type Queue. The domain and range specifications for these two types are isomorphic. To rely on one's intuition about the meaning of names can be dangerous even when dealing with familiar types [19]. When dealing with unfamiliar types it is almost impossible. What is needed, therefore, is a mechanism for specifying the semantics of the operations of the type.

There are, of course, many possible approaches to the specification of the semantics of an abstract data type. Most, however, can be placed in one of two categories: operational or definitional. In an operational specification, instead of trying to describe the properties of the abstract data type, one gives a recipe for constructing it. One begins with some well-understood language or discipline and builds a model for the type in terms of that discipline. Wulf [24], for example, makes good use of sequences in modeling various data structures.

The operational approach to formal specification has many advantages. Most significantly, operational specifications seem to be relatively (compared to definitional specifications) easily constructed by those trained as programmers—chiefly because the construction of operational specifications so closely resembles programming. As the operations to be specified grow complex, however, operational specifications tend to get too long (see, for example, Batey [1]) to permit substantial confidence in their aptness. As the number of operations grows, problems arise because the relations among the operations are not explicitly stated, and inferring them becomes combinatorially harder.

The most serious problem associated with operational specifications is that they almost always force one to overspecify the abstraction. By introducing extraneous detail, they associate nonessential attributes with the type. This extraneous detail complicates the problem of proving the correctness of an implementation by introducing conditions that are irrelevant, yet nevertheless must be verified. More importantly, the introduction of extraneous detail places unnecessary constraints on the choice of an implementation and may potentially eliminate the best solutions to the problem.

Axiomatic definitions avoid this problem. The algebraic approach used here owes much to the work of Hoare [13] (which in turn owes much to Floyd [5]) and is closely related to Standish's "axiomatic specifications" [22] and Zilles' "algebraic specifications" [25]. Its formal basis stems from the heterogeneous algebras of Birkhoff and Lipson [2]. An algebraic specification of an abstract type consists of two pairs: a syntactic specification and a set of relations. The syntactic specification provides the syntactic information that many programming languages already require: the names, domains, and ranges of the operations associated with the type. The set of relations defines the meanings of the operations by stating their relationships to one another.

3. A Short Example

Consider type Queue (of Items) with the operations listed in the previous section. The syntactic specification is as above:

NEW:	→ Queue
ADD:	Queue × Item → Queue
FRONT:	Queue → Item
REMOVE:	Queue → Queue
IS_EMPTY?:	Queue → Boolean

The distinguishing characteristic of a queue is that it is a first in-first out storage device. A good axiomatic definition of the above operations must therefore assert that and only that characteristic. The relations (or axioms) below comprise just such a definition. The meanings of the axioms should be relatively clear. ("=" has its standard meaning, "*q*" and "*i*" are typed free variables, and "error" is a distinguished value with the property that the value of any operation applied to an argument list containing error is error, e.g. $f_n(x_1, \ldots, x_i, \text{error}, x_{i+2}, \ldots, x_n) = \text{error}$.)

(1) IS_EMPTY? (NEW) = true
(2) IS_EMPTY? (ADD(q,i)) = false
(3) FRONT(NEW) = error
(4) FRONT (ADD(q,i)) = **if** IS_EMPTY? (q)
then i
else FRONT(q)
(5) REMOVE(NEW) = error
(6) REMOVE (ADD(q,i)) = **if** IS_EMPTY? (q)
then NEW
else ADD(REMOVE(q),i)

Note that this set of axioms involves no assumption about the attributes of type Item. In effect Item is a parameter of type Type, and the specification may be viewed as defining a type schema rather than a single type. This will be the case for many algebraic type specifications.

With some practice, one can become quite adept at reading algebraic axiomatizations. Practice also makes it easier to construct such specifications; see Guttag [11]. Unfortunately, it does not make it trivial. It is not always immediately clear how to attack the problem. Nor, once one has constructed an axiomatization, is it always easy to ascertain whether or not the axiomatization is consistent and sufficiently complete. The meaning of the operations is supplied by a set of individual statements of fact. If any two of these are contradictory, the axiomatization is inconsistent. If the combination of statements is not sufficient to convey all of the vital information regarding the meaning of the operations of the type, the axiomatization is not sufficiently complete.[1]

Experience indicates that completeness is, in a practical sense, a more severe problem than consistency. If one has an intuitive understanding of the type being specified, one is unlikely to supply contradictory axioms. It is, on the other hand, extremely easy to overlook one or more cases. Boundary conditions, e.g. REMOVE(NEW), are particularly likely to be overlooked.

In an attempt to ameliorate this problem, we have devised heuristics to aid the user in the initial presentation of an axiomatic specification of the operations of an abstract type and a system to mechanically "verify" the sufficient-completeness of that specification. As the first step in defining a new type, the user would supply the system with the syntactic specification of the type and an axiomatization constructed with the aid of the heuristics mentioned above. Given this preliminary specification, the system would begin to prompt the user to supply the additional information necessary for the system to derive a sufficiently complete axiom set

[1] Sufficiently complete is a technical notion first developed in Guttag [8]. It differs considerably from both the notion of completeness commonly used in logic and that used in Zilles [25].

for the operations. A detailed look at sufficient-completeness is contained in Guttag [8, 9].

4. An Extended Example

A common data structuring problem is the design of the symbol table component of a compiler for a block structured language. Many sources contain good discussions of various symbol table organizations. Setting aside variations in form, the basic operations described vary little from source to source. They are:

INIT:	Allocate and initialize the symbol table.
ENTERBLOCK:	Prepare a new local naming scope.
LEAVEBLOCK:	Discard entries from the most recent scope entered, and reestablish the next outer scope.
IS_INBLOCK?:	Has a specified identifier already been declared in this scope? (Used to avoid duplicate declarations.)
ADD:	Add an identifier and its attributes to the symbol table.
RETRIEVE:	Return the attributes associated (in the most local scope in which it occurs) with a specified identifier.

Though many references provide insights into how these operations can be implemented, none presents a formal definition (other than implementations) of exactly what they mean. The abstract concept "symbol table" thus goes undefined. Those who attempt to write compilers in a top-down fashion suffer from a similar problem. Early refinements of parts of the compiler make use of the basic symbol table operations, but the "meaning" of these operations is provided only by subsequent levels of refinement. This is infelicitous in that the clear separation of levels of abstraction is lost and with it many of the advantages of top-down design. By providing axiomatic semantics for the operations, this problem can be avoided.

The thought of providing rigorous definitions for so many operations may, at first, seem a bit intimidating. Nevertheless, if one is to understand the refinement, one must know what each operation means. The following specification of abstract type Symboltable supplies these meanings.

Type: Symboltable

Operations:

INIT:	→ Symboltable
ENTERBLOCK:	Symboltable → Symboltable
LEAVEBLOCK:	Symboltable → Symboltable
ADD:	Symboltable × Identifier × Attributelist → Symboltable
IS_INBLOCK?:	Symboltable × Identifier → Boolean
RETRIEVE:	Symboltable × Identifier → Attributelist

Axioms:

(1) LEAVEBLOCK(INIT) = error

(2) LEAVEBLOCK(ENTERBLOCK(symtab)) = symtab

(3) LEAVEBLOCK(ADD(symtab, id, attrs)) = LEAVEBLOCK(symtab)

(4) IS_INBLOCK? (INIT, id) = false

(5) IS_INBLOCK? (ENTERBLOCK(symtab), id) = false

(6) IS_INBLOCK? (ADD(symtab, id, attrs), idl) =
  if IS_SAME? (id, idl)[2]
    then true
    else IS_INBLOCK? (symtab, id)

(7) RETRIEVE(INIT, id) = error

(8) RETRIEVE(ENTERBLOCK(symtab), id) =
  RETRIEVE(symtab, id)

(9) RETRIEVE(ADD(symtab, id, attrs), idl)=
  if IS_SAME? (id, idl)
    then attrs
    else RETRIEVE(symtab, idl)

This set of relations serves a dual purpose. Not only does it define an abstract type that can be used in the specification of various parts of the compiler, but it also provides a complete self-contained specification for a major subsystem of the compiler. If one wished to delegate the design and implementation of the symbol table subsystem, the algebraic characterization of the abstract type would (unlike the informal description in, say, McKeeman [15]) be a sufficient specification of the problem. In fact, the procedure discussed earlier can be used to formally prove the sufficient-completeness of this specification.

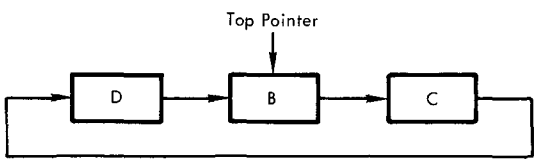
The next step in the design process is to further refine type Symboltable, i.e. to provide implementations of the operations of the type. These implementations will implicitly furnish representation for values of type Symboltable.

A representation of a type T consists of (i) any interpretation (implementation) of the operations of the type that is a model for the axioms of the specification of T, and (ii) a function Φ that maps terms in the model domain onto their representatives in the abstract domain. (This is basically the abstraction function of Hoare [12].)

It is important to note that Φ may not have a proper inverse. Consider, for example, type Bounded Queue (with a maximum length of three). A reasonable representation of the values of this type might be based on a ring-buffer and top pointer. Given this representation, the program segment:

```
x := EMPTY.Q
x := ADD.Q(x, A)
x := ADD.Q(x, B)
x := ADD.Q(x, C)
x := REMOVE.Q(x)
x := ADD.Q(x, D)
```

would translate to a representation for x of the form:

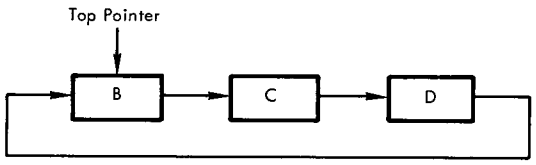


[2] The definition of IS_SAME? is part of the specification of an independently defined type Identifier.

Similarly:

```
x := EMPTY.Q
x := ADD.Q(x, B)
x := ADD.Q(x, C)
x := ADD.Q(x, D)
```

would yield a representation for x of the form:



It is clear that these two representations though not identical, refer to the same abstract value. That is to say, the mapping from values to representations, Φ^{-1}, may be one-to-many.

The representation of type Symboltable will make use of the abstract data types Stack (of arrays) and Array (of attributelists) as defined below.

Type: Stack

Operations:

NEWSTACK:	→ Stack
PUSH:	Stack × Array → Stack
POP:	Stack → Stack
TOP:	Stack → Array
IS_NEWSTACK?:	Stack → Boolean
REPLACE:	Stack × Array → Stack

Axioms:

(10) IS_NEWSTACK? (NEWSTACK) = true
(11) IS_NEWSTACK? (PUSH(stk, arr)) = false
(12) POP(NEWSTACK) = error
(13) POP(PUSH(stk, arr)) = stk
(14) TOP(NEWSTACK) = error
(15) TOP(PUSH(stk, arr)) = arr
(16) REPLACE(stk, arr) = **if** IS_NEWSTACK? (stk)
then error
else PUSH(POP(stk), arr)

Type: Array

Operations:

EMPTY:	→ Array
ASSIGN:	Array × Identifier × Attributelist → Array
READ:	Array × Identifier → Attributelist
IS_UNDEFINED?:	Array × Identifier → Boolean

Axioms:

(17) IS_UNDEFINED? (EMPTY, id) = true
(18) IS_UNDEFINED? (ASSIGN(arr, id, attrs), idl) =
if IS_SAME? (id, idl)
then false
else IS_UNDEFINED? (arr, idl)
(19) READ(EMPTY, id) = error
(20) READ(ASSIGN(arr, id, attrs), idl) = **if** IS_SAME? (id, idl)
then attrs
else READ(arr, idl)

The general scheme of the representation of type Symboltable is to treat a value of the type as a stack of arrays (with index type Identifier), where each array contains the attributes for the identifiers declared in a single block. For every function f in the more abstract domain (e.g. type Symboltable), a function f' is defined in the lower-level domain; thus we have:

INIT':	→ Stack
ENTERBLOCK':	Stack → Stack
LEAVEBLOCK':	Stack → Stack
ADD':	Stack × Identifier × Attributelist → Stack
IS_INBLOCK?':	Stack × Identifier → Boolean
RETRIEVE':	Stack × Identifier → Attributelist

The "code" for each of these functions is ("::" means "is defined as"):

```
INIT' :: PUSH(NEWSTACK, EMPTY)
ENTERBLOCK'(stk) :: PUSH(stk, EMPTY)
LEAVEBLOCK'(stk) :: if IS_NEWSTACK? (POP(stk))
                    then error
                    else POP(stk)
ADD'(stk, id, attrs) :: REPLACE(stk, ASSIGN(TOP(stk), id,
  attrs))
IS_INBLOCK?'(stk, id) :: if IS_NEWSTACK? (stk)
                         then false
                         else ¬ IS_UNDEFINED? (TOP(stk),
                           id)
RETRIEVE'(stk, id) :: if IS_NEWSTACK? (stk)
                      then error
                      else ¬IS_UNDEFINED?(TOP(stk),id)
                        then RETRIEVE'(POP(stk), id)
                        else READ(TOP(stk), id)
```

The interpretation function Φ is defined by:

(a) Φ(error) = error
(b) Φ(NEWSTACK) = error
(c) Φ(PUSH(stk, EMPTY)) = **if** IS_NEWSTACK? (stk)
then INIT
else ENTERBLOCK(Φ(stk))
(d) Φ(PUSH(stk, ASSIGN(arr, id, attrs))) = ADD(ΦPUSH(stk, arr)), id, attrs))

Before continuing to refine these operations, i.e. before supplying representations for types Array and Stack, let us consider the problem of proving that the above implementation of type Symboltable is correct.

In the course of such a proof two kinds of invariants may have to be verified: inherent invariants and representation invariants. The inherent invariants represent those invariant relationships that must be maintained by any representation of the type. They correspond to the axioms used in the specification of the type. A representation invariant, on the other hand, is peculiar to a particular representation of a type.

The basic procedure followed in verifying the inherent invariants is to take each axiom for type Symboltable and replace all instances of each function appearing in the axiomatization with its interpretation. Then, by using the axiomatizations of the operations used in constructing the representations, it is shown that the left-hand side of each axiom is equivalent to the right-hand side of that axiom. That is to say, they represent the same abstract value.

What must be shown therefore is that for every relation $f'(x^*) = z$ (where x^* is a list, possibly empty, of arguments), derived from the axiomatization of type Symboltable,

(a) if the range of f is the type being defined (i.e., Symboltable), $\Phi(f'(x^*)) = \Phi(z)$ for all legal assignments to the free variables of x^* and z, or

(b) if the range of f is a type other than that being defined, $f'(x^*) = z$ for all legal assignments to the free variables of x^* and z.

To show this, we have at our disposal a proof system consisting of the axioms and rules of inference of our programming language plus the axioms defining the abstract types used in the representation.

The proof depends upon the assumption that objects of type Symboltable are created and manipulated only via the operations defined in the specification of that type. (The use of classes as described in Palme [18] makes this assumption relatively easy to verify.) All that need be shown is that INIT′ establishes the invariants and that if on entry to an operation all invariants hold for all objects of type Symboltable to be manipulated by that operation, then all invariants on those objects hold upon completion of that operation. More complete discussions of how this may be done are contained in Guttag [8], Spitzen [21], and Wegbreit [23] (where it is called generator induction).

To verify that the implementation is consistent with Axioms 1 through 8 is quite straightforward. (It has, in fact, been done completely mechanically by David Musser [17] using the program verification system at the University of Southern California Information Sciences Institute [7]. Thus the proofs will not be presented here. Axiom 9, on the other hand, presents some problems that make the portion of the proof pertinent to that axiom worth examining.

The proof that the implementation satisfies Axiom 9 is based upon an assumption about the environment in which the operations of the type are to be used. In effect, the assumption asserts that an identifier is never added to an empty symbol table, i.e. a scope must have been established (on a more concrete level, an array must have been pushed onto the stack) before an identifier can be added. The concrete manifestation of this assumption is formally expressed:

Assumption 1. For any term, ADD′(symtab, id, attrs), IS_NEWSTACK? (symtab) = false.

The validity of the above assumption can be assured by adding to the implementation of ADD′ a check for this condition and having it execute an ENTERBLOCK′ if necessary. This would make it possible to construct a completely self-contained proof of the correctness of the representation. In most cases, however, it would also introduce needless inefficiency. The compiler must somewhere check for mismatched (i.e. extra) "end" statements. Any check in ADD′ would therefore be redundant.

This observation leads to a notion of conditional correctness, i.e. the representation of the abstract type is correct if the enclosing program obeys certain constraints. In practice, this is often an extremely useful notion of correctness, especially if the constraint is easily checked. If, on the other hand, the environment in which the abstract type is to be used is unknown (e.g. if the type is to be included in a library), this is probably unacceptably dangerous. Given the above assumption, the verification of Axiom 9 is straightforward but lengthy and will therefore not be presented here. It does appear in Guttag [8].

Now we know that, given implementations of types Stack and Array that are consistent with their specifications, the implementation of type Symboltable is "correct." Assuming PL/I-like based variables, pointers, and structures, the implementation of type Stack is trivial. The basic scheme is to represent a stack as a pointer to a list of structures of the form:

```
1. stack elem based,
   2. val Array,
   2. prev pointer.
```

The operations may be implemented as follows (PL/I keywords have been boldfaced):

```
NEWSTACK′ :: null
PUSH′(symtab, newblock) ::
  procedure(symtab: pointer, newblock: Array)returns(pointer)
    declare elem_ptr pointer
    allocate(stack_elem) set(elem_ptr)
    elem_ptr → prev := symtab
    elem_ptr → val := newblock
    return(elem_ptr)
  end
POP′(symtab) ::
  procedure(symtab: pointer) returns(pointer)
    if symtab = null
      then return(error)
      else return(symtab → prev)
  end
TOP′(symtab) ::
  procedure(symtab: pointer) returns(Array)
    if symtab = null
      then return(error)
      else return(symtab → val)
  end
IS_NEWSTACK?′(symtab) :: symtab = null
REPLACE′(symtab, newblock) ::
  procedure(symtab: pointer, newblock: Array) returns(pointer)
    if symtab = null
      then return(error)
      else symtab → val := newblock
        return(symtab)
  end
```

Φ is defined by the mapping:

```
Φ(symtab) :: if symtab = null
               then NEWSTACK
               else PUSH(Φ(symtab → prev), symtab → val))
```

The implementation chosen for type Array is a bit more complicated. The basic scheme is to represent an array as a PL/I-like array, hash_tab, of n pointers to lists of structures of the form:

```
1. entry based,
   2. id Identifier
   2. attributes Attributelist,
   2. next pointer.
```

The correct element of hash_tab is selected by performing a hash on values of type Identifier. Therefore, in

addition to the operations used in the code above, the implementation of type Array uses an operation

HASH:Identifier → {1, 2, . . . , n}

which is assumed to be defined in the type Identifier specification. The "code" implementing type Array is:

```
declare hash_tab(n) pointer based

EMPTY' ::
  procedure returns(pointer)
    declare new_hash_tab pointer
    allocate (hash_tab) set (new_hash_tab)
    do i := 1 to n
      new_hash_tab → hash_tab(i) := null
    end
    return(new_hash_tab)
  end
ASSIGN'(arr, indx, atr) ::
  procedure(arr: pointer, indx: Identifier, atr: Attributelist)
        returns(pointer)
    declare new_entry pointer
    allocate(entry) set (new_entry)
      new_entry → id := indx
      new_entry → attributes := atr
      new_entry → next := arr → hash_tab(HASH(indx))
      arr → hash_tab(HASH(indx)) := new_entry
    return(arr)
  end
READ'(arr, indx) ::
  procedure(arr: pointer, indx: Identifier) returns(Attributelist)
    declare bucket_ptr pointer
      bucket_ptr := arr → hash_tab(HASH(indx))
    do while(bucket_ptr ≠ null & ¬ IS_SAME?(bucket_ptr → id,
        indx))
      bucket_ptr := bucket_ptr → next
    end
    if bucket_ptr = null
      then return(error)
      else return (bucket_ptr → attributes)
  end
IS_UNDEFINED?'(arr, indx) ::
  procedure(arr: pointer, indx: Identifier) returns(Boolean)
    declare bucket_ptr pointer
      bucket_ptr := arr → hash_tab(HASH(indx))
    do while (bucket_ptr ≠ null & ¬ IS_SAME? (bucket_ptr → id,
        indx))
      bucket_ptr := bucket_ptr → next
    end
    return (bucket_ptr = null)
  end
```

As one might expect, Φ is a bit more complex for this representation. It is defined by using two intermediate functions: Φ1 to construct a union over all the entries in the hash table, and Φ2 to construct a union over the elements of an individual bucket.

```
(a) Φ(hash_tab_ptr) = Φ1(hash_tab_ptr, EMPTY, 1)
(b) Φ1(hash_tab_ptr, arr, i) =
      if i > n
        then arr
        else Φ1(hash_tab_ptr, Φ2(hash_tab_ptr → hash_tab(i), arr),
          i + 1)
(c) Φ2(bucket_ptr,arr) =
      if bucket_ptr = null
        then arr
        else ASSIGN(Φ2(bucket_ptr → next, arr), bucket_ptr → id,
          bucket_ptr → attributes)
```

The design of the symbol table subsystem of the compiler is now essentially complete. Given implementations of types Identifier and Attributelist and some obvious syntactic transformations, the above code could be compiled by a PL/I compiler. Before doing so, however, it would be wise to prove that the implementations of types Stack and Array are consistent with the specifications of those types. While such a proof would involve substantial issues related to the general program verification problem (e.g. vis à vis the integrity of the pointers and the question of modifying shared data structures), it would not shed further light on the role of abstract data types in program verification and is not presented in these pages.

The ease with which algebraic specifications can be adapted for different applications is one of the major strengths of the technique. Because the relationships among the various operations appear explicitly, the process of deciding which axioms must be altered to effect a change is straightforward. Let us consider a rather substantial change in the language to be compiled. Assume that the language permits the inheritance of global variables only if they appear in a "knows list," which lists, at block entry, all nonlocal variables to be used within the block [6]. The symbol table operations in a compiler for such a language would be much like those already discussed. The only difference visible to parts of the compiler other than the symbol table module would be in the ENTERBLOCK operation: It would have to be altered to include an argument of abstract type Knowlist. Within the specification of type Symboltable, all relations, and only those relations, that explicitly deal with the ENTERBLOCK operation would have to be altered. An appropriate set of axioms would be:

```
IS_INBLOCK?(ENTERBLOCK(symtab, klist), id) = false
LEAVEBLOCK(ENTERBLOCK(symtab, klist)) = symtab
RETRIEVE(ENTERBLOCK(symtab, klist), id) =
  if IS_IN?(klist, id)
    then RETRIEVE(symtab, id)
    else error
```

Note that the above relations are not well defined. The undefined symbol IS_IN?, an operation of the abstract type Knowlist, appears in the third axiom. The solution to this problem is simply to add another level to the specification by supplying an algebraic specification of the abstract type Knowlist. An appropriate set of operations might be:

```
CREATE:          → Knowlist
APPEND:  Knowlist × Identifier → Knowlist
IS_IN?:  Knowlist × Identifier → Boolean
```

These operations could then be precisely defined by the following axioms:

```
IS_IN?(CREATE) = false
IS_IN?(APPEND(klist, id), idl) = if IS_SAME?(id, idl)
                                   then true
                                   else IS_IN?(klist, idl)
```

The implementation of abstract type Knowlist is trivial. The changes necessary to adapt the previously presented implementation of abstract type Symboltable would be more substantial. The kind of changes necessary can, however, be inferred from the changes made to the axiomatization.

5. Conclusions

We have not yet applied the techniques discussed in this paper to realistically large software projects. Nevertheless, there is reason to believe that the techniques demonstrated will "scale up." The size and complexity of a specification at any level of abstraction are essentially independent of both the size and complexity of the system being described and of the amount of mechanism ultimately used in the implementation. The independence springs in large measure from the ability to separate the precise meaning of a complex abstract data type from the details involved in its implementation. It is the ability to be precise without being detailed that encourages the belief that the approach outlined here can be applied even to "very large" systems can and perhaps reduce systems that were formerly "very large" (i.e. incomprehensible) to more manageable proportions.

Abstract types may thus play a vital role in the formulation and presentation of precise specifications for software. Many complex systems can be viewed as instances of an abstract type. A database management system, for example, might be completely characterized by an algebraic specification of the various operations available to users. For those systems that are not easily totally characterized in terms of algebraic relations, the use of algebraic type specifications to abstract various complex subsystems may still make a substantial contribution to the design process. The process of functional decomposition requires some means for specifying the communication among the various functions—data often fulfills this need. The use of algebraic specifications to provide abstract definitions of the operations used to establish communication among the various functions may thus play a significant role in simplifying the process of functional abstraction.

The extensive use of algebraic specifications of abstract types may also lead to better-designed data structures. The premature choice of a storage structure and set of access routines is a common cause of inefficiencies in software. Because they serve as the main means of communication among the various components of many systems, the data structures are often the first components designed. Unfortunately, the information required to make an intelligent choice among the various options is often not available at this stage of the design process. The designer may, for example, have poor insight into the relative frequency of the various operations to be performed on a data structure. By providing a representation-free, yet precise, description of the operations on a data structure, algebraic type definitions enable the designer to delay the moment at which a storage structure must be designed and frozen.

The second area in which we expect the algebraic specification of abstract types to have a substantial impact is on proofs of program properties. For verifications of programs that use abstract types, the algebraic specification of the types used provides a set of powerful rules of inference that can be used to demonstrate the consistency of the program and its specification. That is to say, the presence of axiomatic definitions of the abstract types provides a mechanism for proving a program to be consistent with its specifications, provided that the implementations of the abstract operations that it uses are consistent with their specifications. Thus a technique for factoring the proof is provided, for the algebraic type definitions serve as the specification of intent at a lower level of abstraction. For proofs of the correctness of representations of abstract types, the algebraic specification provides exactly those assertions that must be verified. The value of having such a set of assertions available should be apparent to any one who has attempted to construct, *a posteriori*, assertions appropriate to a correctness proof for a program. A detailed discussion of the use of algebraic specifications in a semiautomatic program verification system is contained in Guttag [10].

Given suitable restrictions on the form that axiomatizations may take, a system in which implementations and algebraic specifications of abstract types are interchangeable can be constructed. In the absence of an implementation, the operations of the algebra may be interpreted symbolically. Thus, except for a significant loss in efficiency, the lack of an implementation can be made completely transparent to the user. Such a system should prove valuable as a vehicle for facilitating the testing of software.

The ability to use specifications for testing is closely related to the policy of restricted information flow advocated in Parnas [20]. If a programmer is supplied with algebraic definitions of the abstract operations available to him and forced to write and test his module with only that information available to him, he is denied the opportunity to rely intentionally or accidentally upon information that should not be relied upon. This not only serves to localize the effect of implementation errors, but also to increase the ease with which one implementation may be replaced by another. This should, in general, serve to limit the danger of choosing a poor representation and becoming inextricably locked into it.

Before ending this paper, it seems fitting to mention some of the failings and problems associated with the work described. The specification technique presented here requires that all operations be specified as functions, i.e. as mappings from a cross product of values to

a single value. Most programs, on the other hand, are laden with procedures that return several values (via parameters) or no value at all. (The latter kind of procedure is invoked purely for its side effects.) The inability to specify such procedures is a serious problem, but one that we believe can be solved with only minor changes to the specification techniques [10].

The value of abstraction in general and abstraction of data types in particular has been stressed throughout this paper. Nevertheless, the process is not without its dangers. It is all too easy to create abstractions that ignore crucial distinctions or attributes. The specification technique presented here, for example, provides no mechanism for specifying performance constraints and thus encourages one to ignore distinctions based on such criteria. In some environments, such considerations are crucial, and to abstract them out can be disastrous.

Another problem with algebraic specifications is that they supply little direction to implementors. Only experience will tell how easy it is to go from an algebraic specification to an implementation. It is clear, however, that the transition is less easy than from an operational specification.

Our most important reservation pertains to the ease with which algebraic specifications can be constructed and read. They should present no problem to those with formal training in computer science. At present, however, most people involved in the production of software have no such training. The extent to which the techniques described in this paper are generally applicable is thus somewhat open to conjecture.

Acknowledgment. The author is greatly indebted to J.J. Horning of the University of Toronto, who, as the author's thesis supervisor, provided three years of good advice.

References

1. Batey, M., Ed. Working Draft of ECMA/ANSI PL/I Standard Tenth Rev., ANSI, New York, (Sept. 1973).
2. Birkhoff, G., and Lipson, J.D. Heterogeneous algebras. *J. Combinatorial Theory 8* (1970), 115-133.
3. Dahl, O.-J., Nygaard, K., and Myhrhaug, B. The SIMULA 67 Common Base Language. Norwegian Comptng. Centre, Oslo, 1968.
4. Dijkstra, E.W. Notes on structured programming. In *Structured Programming*, Academic Press, New York, 1972.
5. Floyd, R.W. Assigning Meaning to Programs. Proc. Symp. in Applied Math., Vol. XIX, AMS, Providence, R.I., 1967, pp. 19-32.
6. Gannon, J.D. Language design to enhance programming reliability. Ph.D. Th., Comptr. Syst. Res. Group Tech. Rep. CSRG-47, Dept. Comptr. Sci., U. of Toronto, Ontario, 1975.
7. Good, D.I., London, R.L., and Bledsoe, W.W. An interactive program verification system. *IEEE Trans. on Software Engineering SE-1*, 1 (March 1975), 59-67.
8. Guttag, J.V. The specification and application to programming of abstract data types. Ph.D. Th., Comptr. Syst. Res. Group Tech. Rep. CSRG-59, Dept. Comptr. Sci. 1975, U. of Toronto, Ontario, 1975.
9. Guttag, J.V. and Horning, J.J., The algebraic specifications of abstract data types. *Acta Informatica* (to appear).
10. Guttag, J.V., Horowitz, E., and Musser, D.R. Abstract data types and software validation. Tech. Rep., Inform. Sci. Inst., U. of Southern California, Los Angeles, 1976.
11. Guttag, J.V., Horowitz, E., and Musser, D.R. The design of data type specifications. Proc. Second Int. Conf. on Software Eng., San Francisco, Oct. 1976, pp. 414-420.
12. Hoare, C.A.R., Proof of correctness of data representations. *Acta Informatica 1* (1972), 271-281.
13. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica 2* (1973), 335-355.
14. Liskov, B.H., and Zilles, S.N. Programming with abstract data types. Proc. ACM SIGPLAN Symp. on Very High Level Languages, SIGPLAN Notices (ACM) *9*, 4 (April 1974), 50-59.
15. McKeeman, W.M., Symbol Table Access. In *Compiler Construction, An Advanced Course*, T.L. Bauer, and J. Eichel, Eds., Springer-Verlag, New York, 1974.
16. Morris, J.H. Types are not sets. Conf. Rec. ACM Symp. on the Principles of Programming Languages, Boston, Mass., Oct. 1973, pp. 120-124.
17. Musser, D. Private communication, 1975.
18. Palme, J. Protected program modules in SIMULA 67. FOAP Rep. C8372-M3(E5), Res. Inst. of National Defense, Stockholm, 1973.
19. Parnas, D.L. A technique for the specification of software modules with examples. *Comm. ACM 15*, 5 (May 1973), 330-336.
20. Parnas, D.L. Information distribution aspects of design methodology. Information Processing 71, North Holland Pub. Co., Amsterdam, 1971, pp. 339-344.
21. Spitzen, J., and Wegbreit, B. The verification and synthesis of data structures. *Acta Informatica 4* (1975), 127-144.
22. Standish, T.A. Data structures: An axiomatic approach. BBN Rep. No. 2639, Bolt, Beranek and Newman, Cambridge, Mass., (1973).
23. Wegbreit, B., and Spitzen, J. Proving properties of complex data structures. *J. ACM 23*, 2 (April 1976), 389-396.
24. Wulf, W.A., London, R.L., and Shaw, M. Abstraction and verification in Alphard: Introduction to language and methodology. USC Inform. Sci. Tech. Rep., U. of Southern California, Los Angeles, 1976.
25. Zilles, S.N. Abstract specifications for data types. IBM Res. Lab., San Jose, Calif., 1975.