6. Farber, D.J., and Larson, K.C. The structure of a distributed computing system—the communication system. Proc. Symp. Computer-Communications Networks and Traffic, Polytechnic Inst. of Brooklyn, Brooklyn, N.Y., April 1972, pp. 21-27.
7. Fultz, G.L. Adaptive routing techniques for message switching computer communication networks. Ph.D. Th., UCLA-ENG-7252, U. of California, Los Angeles, July 1972.
8. Gerla, M. The design of store- and forward (S/F) networks for computer communication. Ph.D. Th., UCLA-ENG-7319, U. of California, Los Angeles, 1973.
9. Harary, F. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969.
10. Metcalfe, R.M. Packet Communication. Ph.D. Th., Harvard, Proj. Mac Tech. Rep. No. 114, M.I.T., Cambridge, Mass., Dec. 1973.
11. Metcalfe, R.M., and Boggs, D.R., Ethernet: Distributed packet switching for local computer networks. *Comm. ACM 19,* 7 (July 1976), 395–404.
12. McQuillan, J.M. Adaptive routing algorithms for distributed computer networks. Ph.D. Th., Harvard, BBN Rep. 2831, May 1974; available as AD781467, N.T.I.S., Springfield, Va.
13. Paoletti, L.M. AUTODIN. In *Computer Communication Networks,* R.L. Grimsdale and F.F. Kuo, Eds. (Proc. NATO Advanced Study Inst. Comptr. Comm. Networks, Sussex, U.K., Sept. 1973), Noordoff Int. Publ., Leyden, 1975.
14. Roberts, L.G., and Wessler, B.D. The ARPA computer network. In *Computer Communication Networks,* N. Abramson and F. Kuo, Eds., Prentice-Hall, Englewood Cliffs, N.J., 1972.

Corrigendum. Programming Languages

David Gries, An Exercise in Proving Parallel Programs Correct, *Comm. ACM 20,* 12 (Dec. 1977), 921–930.

Dr. Leslie Lamport detected what appeared to be a methodological mistake in the proof of the on-the-fly garbage collector. The assignment *atleastgray*($m[i].left$) of the Collector (see the algorithm labeled (3.6) on page 925) contains references to the *two* shared variables $m[i].left$ and $m[m[i].left].color$, and this clearly violates the restriction (2.10) found on page 923.

The problem is not a methodological error but a missing footnote. The statement *atleastgray*($m[i].left$) in (3.6) does have a footnote number 3 attached to it, and an earlier version of the paper [Springer Lecture Notes in Computer Science 46, 1976, 57–81] contained the footnote

> This should be written as "$t := m[i].left$; *atleastgray*(t)" where t is a local variable. Since the mutator never tests the color of a node and only grays a node using also *atleastgray*, the single statement *atleastgray*($m[i].left$) is equivalent under parallel operation to this sequence of two operations.

Dr. Lamport also noted that the informal discussion of noninterference of assertions (4.5.1) and (4.5.2) in the first four paragraphs of Section 4.5 could be interpreted as using circular reasoning, but that a formal proof of noninterference does indeed work.

My thanks to Dr. Lamport for pointing out these problems and my apologies for any inconvenience they have caused the reader.

Programming Languages — J.J. Horning, Editor

Abstract Data Types and Software Validation

John V. Guttag, Ellis Horowitz, and David R. Musser
University of Southern California

A data abstraction can be naturally specified using algebraic axioms. The virtue of these axioms is that they permit a representation-independent formal specification of a data type. An example is given which shows how to employ algebraic axioms at successive levels of implementation. The major thrust of the paper is twofold. First, it is shown how the use of algebraic axiomatizations can simplify the process of proving the correctness of an implementation of an abstract data type. Second, semi-automatic tools are described which can be used both to automate such proofs of correctness and to derive an immediate implementation from the axioms. This implementation allows for limited testing of programs at design time, before a conventional implementation is accomplished.

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

1. Introduction

The key problem in the design and validation of large software systems is reducing the amount of com-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
This research was supported in part by the National Science Foundation under Grant MCS76-06089, by the Defense Research Projects Agency under Contract DAHC15 72 C 0308, and by the Joint Services Electronics Program Monitored by the Air Force Office of Scientific Research under Contract F44 620-76-C-0061.
Authors' addresses: J.V. Guttag and E. Horowitz, Computer Science Department, University of Southern California, Los Angeles, CA 90007; D.R. Musser, USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291.
© 1978 ACM 0001-0782/78/1200–1048 $00.75

plexity or detail that must be considered at any one time; two common and effective solution methods are decomposition and abstraction. One decomposes a task by factoring it into two or more separable subtasks. Unfortunately, for many problems the separable subtasks are still too complex to be mastered in toto. The complexity of this sort of problem can be reduced via abstraction. By providing a mechanism for separating those attributes of an object or event that are relevant in a given context from those that are not, abstraction serves to reduce the amount of detail that must be comprehended at any one time.

If one is to make full use of abstraction, it is critical to have available a good notation for expressing abstractions. It is obvious that a reasonable language is a prerequisite to communicating something as intangible as an abstraction; it is less obvious, but equally true, that a reasonable language is a prerequisite to the creation of such abstractions. Even if a language need not be available for the initial formulation of an abstraction (an argument we leave to psychologists and linguists), it is certainly necessary if the abstraction is to be retained and developed over any significant period of time.

A recent trend in programming is the development of the abstract data type or data abstraction. In data abstraction, a number of functional abstractions are grouped together. The clustered operations are related by the fact that they, and only they, operate on a particular class or type of object. Some typical data abstractions are a symbol table, a priority queue, and a set.

In this paper we shall use a notation, which we call algebraic axioms, for describing data abstractions. In order to show how these specifications can be used during the design process, we exhibit, in Sections 2 and 3, their use in the creation of a symbol table which allows for block structure.

However, the point we wish to stress in this paper is not the design of data abstractions, but the use of algebraic axioms for proofs of correctness and for program testing. In Section 4 we show how this axiomatic technique can be employed to prove the correctness of an implementation of a data abstraction. The strength of the technique is that it factors the proving process into distinct, manageable stages; further, it simplifies the proof at each stage. In Section 5 we discuss an automated system which processes algebraic axiomatizations of data abstractions in such a way that correctness proofs of implementations can be carried out semi-automatically; in addition, programs may be tested before an implementation in a conventional programming language is achieved. This coupling of testing and correctness is a valuable by-product of the algebraic axiom approach and is a strong argument for its worth.

It is important to note that the techniques developed in this paper are essentially programming language independent. While languages with the compile time facilities of Simula 67 [2] (with the extensions of [18]), CLU [14], or Alphard [24] will make these techniques easier to apply, they are by no means essential. If one exercises enough self- (or project-wide) discipline to ensure the validity of what we have called data type induction (see Section 4.3), the techniques described should prove useful in the development of programs in a wide class of languages.

Related work has been done by a number of other people. The relation of our work to that of Zilles and of Goguen, Thatcher, Wagner, and Wright will be discussed in Sections 2 and 4.5. Spitzen and Wegbreit, [20] and [23], have taken a similar approach to proofs about abstract data types, but most of their specifications were in a form more akin to Hoare's system, than to the conditional equations which we employ. Our proof techniques are related to those of Boyer and Moore [1] (see also Section 4.1) and Suzuki [22], who have made extensive use of axioms and lemmas in the form of rewrite rules in automated proof systems, but have not stressed the organization of rules into specifications of data types.

2. Definitions, Concepts, and Examples

Rather than present formal definitions of our data abstraction mechanism and related concepts, we give informal and (hopefully) intuitively appealing definitions and illustrate the main ideas with a number of examples. We shall view a *data type* T as a class of values and a collection of operations on the values. If the properties of the operations are specified only by axioms, we call T an *abstract data type* or a *data abstraction.* An *implementation* of a data abstraction is an assignment of meaning to the values and operations in terms of the values and operations of another data type or set of data types. A *correct implementation* is an implementation which satisfies the axioms. If this implementation is to be useful, it must also be possible to correctly implement the underlying types used. The danger here is that if one writes an inconsistent specification of one of these underlying types, it will not be implementable, yet technically speaking we still have a correct implementation at the higher level.

An *algebraic axiom specification* of a data type T consists of a *syntactic* and a *semantic* specification. The syntactic specification defines the names, domains, and ranges of the operations of T. The semantic specification contains a set of axioms in the form of equations which relate the operations of T to each other. The term "algebraic" is appropriate because the values and operations can be regarded as an abstract algebra. Goguen [3] and Zilles [25] have strongly emphasized the algebraic approach, developing the theory of abstract data types as an application of many-sorted algebras. Implementations are treated under this approach as other algebras, and the problem of showing the correctness of an implementation is treated as one of showing the existence of a homomorphic mapping from one algebra to another.

We shall in this paper deemphasize the explicit use of algebraic terminology, preferring instead the terminology of programming. In spite of this difference in terminology, there are many similarities between our approach and the more purely algebraic approach. A brief discussion of a technical difference between the two approaches is contained in Section 4.5.

The choice of a language in which to express the specifications is important. We must be able to express the relationships among the operations precisely and clearly. In addition, the specification language itself must be axiomatically defined to facilitate correctness proofs. We begin by assuming a base language with five primitives: functional composition, an equality relation (=), two distinct constants (TRUE and FALSE), and an unbounded supply of free variables. From these primitives, one can construct an arbitrarily complex specification language, for once an operation has been defined in terms of the primitives, it may be added to the specification language. An IF-THEN-ELSE operation, for example, may be defined by the axioms:

IF-THEN-ELSE(TRUE, q, r) = q,
IF-THEN-ELSE(FALSE, q, r) = r.

We shall assume that the expression IF-THEN-ELSE(b, q, r), which we shall write as IF b THEN q ELSE r, is part of the specification language. We shall also assume the availability of infix Boolean operators such as $\wedge$, $\vee$, $\supset$, $\equiv$, and the prefix operator $\neg$. Finally, we allow for the conventional operations on integers: PLUS, MINUS, TIMES, DIV, MOD, and use the conventional infix operators when convenient.

2.1 Stack Example

One of the simplest examples of an abstract data type is the unbounded Stack. In the example of Figure 1 we have defined a data type Stack with six operations via a syntactic specification of these operations, and a semantic specification which is a set of seven equations relating the operations. Certain notational conventions exhibited by this example will be used throughout. Operation names are written using all capital letters. The name of a data type begins with a capital. In the equations the lowercase symbols are free variables ranging over the domains indicated, e.g. stk ranges over the Stack type. The symbol elementtype is a variable ranging over the set of types and elm ranges over elementtype. This says that we can have a Stack of any type of elements (but all must be of the same type); what we have defined is thus not a single abstract type but rather a *type schema.* The binding of elementtype to a particular type, e.g. Stack[Integer], reduces the schema to a specification of a single abstract type. Using the syntactic specification of the operations, one can check that each of the expressions in the axiomatic equations is well-formed in the sense that each operator is applied to the correct number of arguments and each argument is of the correct type.

The equations are statements of fact (axioms) relating the values which are created by the operations, e.g. the equation

POP(NEWSTACK) = NEWSTACK

states that an attempt to POP the empty stack will always yield NEWSTACK. (The decision to return NEWSTACK, rather than, say, UNDEFINED, is an arbitrary one, and may not coincide with some readers' preconceptions about the behavior of stacks.)

TOP(PUSH(stk, elm)) = elm

means that for any Stack value stk and any elementtype value elm, the result of PUSH(stk, elm) is a Stack value, stk1, such that TOP(stk1) yields the value elm. In viewing the equations in this way, we are not required to give any particular interpretation to the values; the "useful" properties of the values can be derived solely from the relations determined by the axioms. Thus in designing computer implementations of the operations, we are free to represent the values in many different ways.

Fig. 1. Stack data type.

type Stack[elementtype:Type]

syntax
  NEWSTACK → Stack,
  PUSH(Stack, elementtype) → Stack,
  POP(Stack) → Stack,
  TOP(Stack) → elementtype ∪ {UNDEFINED},
  ISNEW(Stack) → Boolean,
  REPLACE(Stack, elementtype) → Stack.

semantics
 declare stk:Stack, elm:elementtype;
  POP(NEWSTACK) = NEWSTACK,
  POP(PUSH(stk, elm)) = stk,
  TOP(NEWSTACK) = UNDEFINED,
  TOP(PUSH(stk, elm)) = elm,
  ISNEW(NEWSTACK) = TRUE,
  ISNEW(PUSH(stk, elm)) = FALSE,
  REPLACE(stk, elm) = PUSH(POP(stk), elm).

The (unbounded) Stack data type can be implemented in terms of an (Array, Integer) pair. Each Stack value is represented by a structure with two components: an (unbounded) array, whose components are of type elementtype, and an integer indicating the position in the array of the top element of the stack. The specifications for an Array data type are given in Figure 2.

ASSIGN(arr, t, elm) means the array identical to arr except possibly in the t-th position where the value is elm [16]. ACCESS(arr, t) returns the value in position t of the array arr. Note the assumption (in the Boolean expression dval = dval1) that there exists an operation, =, from domaintype × domaintype → Boolean. It is a requirement that this operation exist for the actual value of domaintype (see Alphard's *requires* clause [24]).

The implementation of the Stack data type with (Array, Integer) pairs is given in Figure 3. We have divided the implementation into a *representation* part and a *programs* part. In this paper the language used to express programs is the same as the language used to

Fig. 2. Array data type.

```
type Array[domaintype:Type, rangetype:Type]
syntax
    NEWARRAY → Array,
    ASSIGN(Array, domaintype, rangetype) → Array,
    ACCESS(Array, domaintype) → rangetype ∪ {UNDEFINED}
semantics
    declare arr:Array, dval, dval1:domaintype, rval:rangetype;
    ACCESS(NEWARRAY, dval) = UNDEFINED,
    ACCESS(ASSIGN(arr, dval, rval), dval1)
      = IF dval = dval1 THEN rval ELSE ACCESS(arr, dval1).
```

Fig. 3. An implementation of the Stack data type with (Array, Integer) pairs.

```
representation STAK(Array[Integer, elementtype], Integer) →
Stack[elementtype],
programs
    declare arr:Array, t:Integer, elm:elementtype;
    NEWSTACK = STAK(NEWARRAY, 0),
    PUSH(STAK(arr, t), elm)
      = STAK(ASSIGN(arr, t + 1, elm), t + 1),
    POP(STAK(arr, t)) = IF t = 0 THEN STAK(arr, 0)
      ELSE STAK(arr, t − 1),
    TOP(STAK(arr, t)) = ACCESS(arr, t),
    ISNEW(STAK(arr, t)) = (t = 0),
    REPLACE(STAK(arr, t), elm)
      = IF t = 0 THEN STAK(ASSIGN(arr, 1, elm), 1)
               ELSE STAK(ASSIGN(arr, t, elm), t).
```

express axioms. Though we recognize that a richer language is usually more desirable, we have chosen to restrict ourselves here for several reasons. Most importantly, the proof procedure described in Section 4 derives much of its simplicity from the use of this restricted set of constructs. Since all conventional programming control constructs can be automatically translated into our basic set, see [15], there is no limitation in principle. We are able to avoid issues of language design and concentrate on how the basic command set can be axiomatized, used for correctness proofs, and used to synthesize implementations. In [5] and [12] the basic language for axiomatizing data types has been applied to an extensive set of examples, with explanations, so the formalism can be further studied there.

We intend that all of the operations be purely "functional" or "applicative," i.e., have no side effects. This can imply an unrealistic degree of inefficiency for implementations. In the Stack implementation of Figure 3, for example, the call of PUSH must involve copying the Array component as well as the Integer component of the Stack representation. The basic framework can be extended to permit specification of operations with side effects so that the obvious efficient implementations are possible. However, since the exposition of our proof techniques is facilitated by this restriction, we will continue in this paper to assume no side effects, and refer the reader to [6] for a discussion of the extensions required to remove this restriction.

The correctness of an implementation of a data type can be proved by showing that each axiom of the semantic specification is satisfied by the programs. As a particularly simple example of such a proof, consider the fourth Stack axiom. Assuming stk = STAK(arr, t),

```
TOP(PUSH(stk, elm)) = TOP(PUSH(STAK(arr, t), elm))
                    = TOP(STAK(ASSIGN(arr, t + 1, elm),
                        t + 1))
                    = ACCESS(ASSIGN(arr, t + 1, elm),
                        t + 1)
                    = elm.
```

The other Stack axioms can be shown to be satisfied in a similar manner, although not quite so straightforwardly. The complications that arise will be dealt with in Section 4, which discusses in detail verification of implementations.

2.2 Programs as Axioms and Axioms as Programs

In the discussion of the implementation for the Stack data type, we described STAK(arr, t) as a pair whose first component is an Array and second component is an Integer. We viewed equations such as

TOP(STAK(arr, t)) = ACCESS(arr, t)

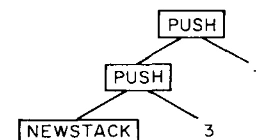
as definitions of programs for operating on the STAK pairs. Suppose, however, that we now view STAK as an *operation* whose syntactic specification is STAK(Array[Integer, elementtype], Integer) → Stack[elementtype]. Then the above equation for TOP and the other program equations can be viewed as *axioms* which comprise a semantic specification for STAK. Looking at it as an axiom, we would read the above equation as "if stk is the result of applying STAK to an Array arr and an Integer t, the value returned by TOP(stk) is ACCESS(arr, t)."

As an axiomatic specification of the Stack data type, the implementation of Figure 3 is inferior to the specification of Figure 1 in that it is not self-contained (it requires knowledge of properties of Arrays and Integers). We have called attention to the view of *programs as axioms* mainly because it suggests a *duality* between programs and axioms whose other half—*axioms as programs*—can be fruitfully exploited. We discuss this duality both here and in Section 5.2.

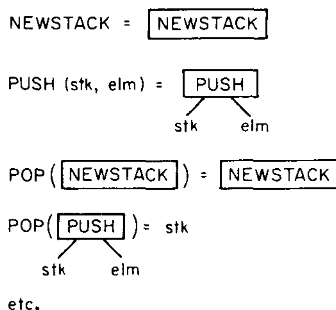
We can, in fact, view the axioms of Figure 1 as programs by regarding NEWSTACK and PUSH(stk, elm) as *trees* rather than operations. All structures built with NEWSTACK and PUSH can be pictured as trees. For example,

PUSH(PUSH(NEWSTACK, 3), 7)

can be diagrammed as



The Stack axioms can be viewed as defining operations which produce and access such tree structures:



The two equations for POP together define POP as an operation which first checks which kind of node it is given and then proceeds accordingly. This is an example of a *direct implementation.*

Direct implementations are useful from a number of standpoints. In the first place, the concept of a direct implementation can serve as an aid to constructing specifications, i.e., one can try to write the semantic axioms so that they can serve as programs operating on tree structures. If this can be done, and one has a compiler which produces running implementations of such programs, then one can experiment with the operations, testing to a limited extent whether they have the properties intended. More importantly, one can also test high-level algorithms which are programmed in terms of the data type, before fixing upon a particular implementation of the data type. Thus a true top-down implementation methodology can be achieved.

2.3 A Richer Example

The Stack data type is too simple, in a number of respects, to illustrate properly the properties and uses of algebraic axiom specifications. A richer example is provided by the symbol table data type. In this example we deal with a common but nontrivial data structuring problem: the design of a symbol table for a compiler for a block-structured language. We wish to specify and implement a set of operations for maintaining the symbol table during compilation of a program. An informal specification of the operations might be as follows:

INIT:	allocate and initialize the symbol table for the outermost scope.
ENTERBLOCK:	prepare a new local naming scope.
ADDID:	add an identifier and its attributes to the symbol table.
LEAVEBLOCK:	discard entries from the most current scope and reestablish the next outer scope. If already in the outermost scope, do nothing.
ISINBLOCK:	has a specified identifier already been declared in this scope? (Used to check for duplicate declarations.)
RETRIEVE:	return the attributes associated with the most local definition of a specified identifier.

A formal specification is given in Figure 4. As an aid to understanding these axioms, it is useful to consider a direct implementation. We let the representation be trees of INIT, ENTERBLOCK, and ADDID nodes and use the set of semantic axioms as programs. Then, for example, if this direct implementation is used by a compiler in processing the following program segment,

```
begin
  real X, Y;
    ...
  begin
    integer Y;
      ...
  end
end
```

the symbol table

SYM = ADDID(ENTERBLOCK(ADDID(ADDID(INIT, X, Real), Y, Real)), Y, Integer)

will be created within the innermost block. Diagrammed as a tree structure, this is shown below

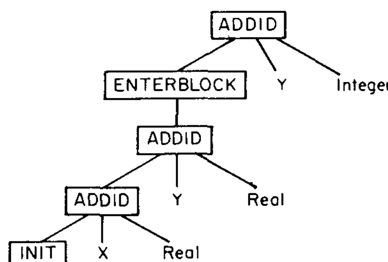


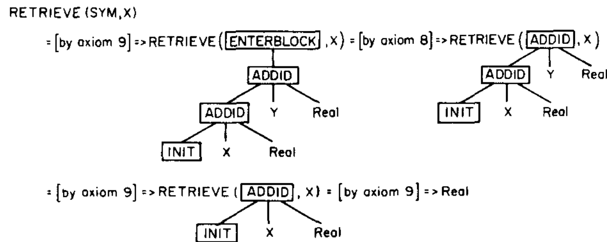
Fig. 4. The Symboltable data type.

```
type Symboltable

syntax
        INIT → Symboltable,
        ENTERBLOCK(Symboltable) → Symboltable,
        ADDID(Symboltable, Identifier, Attributelist) →
          Symboltable,
        LEAVEBLOCK(Symboltable) → Symboltable,
        ISINBLOCK(Symboltable, Identifier) → Boolean,
        RETRIEVE(Symboltable, Identifier) →
          Attributelist ∪ {UNDEFINED}.

semantics
  declare symtab:Symboltable, id, id1:Identifier, attrlist:Attributelist;
    1) LEAVEBLOCK(INIT) = INIT,
    2) LEAVEBLOCK(ENTERBLOCK(symtab))
         = symtab,
    3) LEAVEBLOCK(ADDID(symtab, id, attrlist))
         = LEAVEBLOCK(symtab),
    4) ISINBLOCK(INIT, id) = FALSE,
    5) ISINBLOCK(ENTERBLOCK(symtab), id)
         = FALSE,
    6) ISINBLOCK(ADDID(symtab, id, attrlist), id1)
         = IF id = id1
           THEN TRUE
           ELSE ISINBLOCK(symtab, id1),
    7) RETRIEVE(INIT, id) = UNDEFINED,
    8) RETRIEVE(ENTERBLOCK(symtab), id)
         = RETRIEVE(symtab, id),
    9) RETRIEVE(ADDID(symtab, id, attrlist), id1)
         = IF id = id1
           THEN attrlist
           ELSE RETRIEVE(symtab, id1).
```

Suppose now that we apply RETRIEVE to SYM and X. Simulating the RETRIEVE operation using the direct implementation, we have



If tree structure operations are implemented with reasonable efficiency, then this direct implementation could be used in a compiler to test the Symboltable specification more extensively and to test other components of the compiler. However, because this implementation requires potentially long searches for the RETRIEVE, ISINBLOCK, and LEAVEBLOCK operations, it is not very efficient. In the next section we turn to the problem of designing a more efficient implementation.

3. Algebraic Axioms as an Aid to Top-Down Development of Implementations

The key to successful top-down design is the ability to construct, at each level of refinement, abstractions which suppress all irrelevant detail while clearly exposing the relevant concepts and structure. By deferring detail, one reduces the number of decisions that must be made at any one time. Verifying the correctness of each refinement as it is developed is crucial. Therefore, the specification of a refinement, though possibly quite abstract, must be complete and unambiguous. All symbols that appear in it must be well-defined.

Though systems have occasionally been designed in a top-down fashion, they have for the most part been tested from the bottom up. This was necessary because the upper levels could not be easily tested in the absence of an implementation of lower levels. By eliminating (via direct implementations) the necessity of supplying such implementations, one eliminates the need to delay testing while awaiting the implementation of other modules. More importantly, if one executes with specifications rather than implementations of abstract operations, the possible sources of a known error are far more limited.

The ability to use specifications for testing is closely related to the policy of restricted information flow [19]. If a programmer is supplied with algebraic definitions of the abstract operations available to him and is forced to write and test his module with only that information, he is denied the opportunity to rely intentionally or accidentally upon information that should not be relied upon. This serves not only 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, limit the danger of choosing a poor representation and becoming inextricably locked into it.

In this section we carry out a design of a hierarchically structured implementation of the Symboltable data type, using algebraic specifications of the data types employed at each level of the implementation. In an earlier presentation of this design [7] the lowest level of the implementation was expressed as a set of PL/1-like programs. We differ here in that we continue to use the restricted set of language features described in Section 2.

Let us now consider how we might proceed to design a reasonably efficient implementation of the Symboltable data type. First, note that if we ignore the complication introduced by block structure, a symbol table can be viewed abstractly as providing a mapping from identifiers to attribute lists. One way to handle block structure, especially suitable in a one-pass compiler, is to have a stack of mappings, each mapping from identifiers to attribute lists, with the top mapping on the stack corresponding to the current innermost block being processed. This is the method we have chosen in the implementation given in Figure 5.

This implementation uses the operations of the Stack data type schema of Figure 1 and the Mapping data type schema of Figure 6. Note that we have bound the parameters of the Stack and Mapping types. The Mapping data type is the same as the Array data type of Figure 2, except for the addition of an ISDEFINED operation.

Before continuing to refine these operations, i.e. before supplying implementations for types Stack and Mapping, we should consider the problem of ascertaining whether or not the above implementation of the Symboltable data type is correct. This can be approached formally by using the proof techniques to be described in Section 4, or less formally by using testing techniques based on direct implementations of types Stack and Mapping. Had we, for example, tested portions of the compiler by using a direct implementation of type Symboltable (Figure 4), we might now run the same tests using the current implementation of type Symboltable and direct implementations of types Stack and Mapping. We shall not discuss such testing here, but in Section 4 we shall carry out parts of the formal proof of correctness.

Earlier we noted that the development process could be halted when one arrived at a program all of whose operations were either primitive or efficiently directly implementable. We have not yet reached that point. While the use of the Stack yields a significant improvement in the time required for the LEAVEBLOCK operation (over the time required by the direct implementation of the Symboltable type), the time required for the RETRIEVE operation has not been improved at all. A solution to this problem, in the form of a hash-table implementation of the Mapping data type, is given in the Appendix.

Fig. 5. An implementation of the Symboltable data type with a stack of mappings.

representation

```
SYMT(Stack[Mapping[Identifier, Attributelist]]) → Symboltable.
```

programs

```
declare stk:Stack, id:Identifier, attrlist:Attributelist;
INIT = SYMT(PUSH(NEWSTACK, NEWMAP)),
ENTERBLOCK(SYMT(stk)) = SYMT(PUSH(stk, NEWMAP)),
ADDID(SYMT(stk), id, attrlist)
  = SYMT(REPLACE(stk, DEFMAP(TOP(stk), id, attrlist))),
LEAVEBLOCK(SYMT(stk))
  = IF ISNEW(POP(stk))
    THEN SYMT(REPLACE(stk, NEWMAP))
    ELSE SYMT(POP(stk)),
ISINBLOCK(SYMT(stk), id) = ISDEFINED(TOP(stk), id),
RETRIEVE(SYMT(stk), id)
  = IF ISNEW(stk)
    THEN UNDEFINED
    ELSE IF ISDEFINED(TOP(stk), id)
      THEN EVMAP(TOP(stk), id)
      ELSE RETRIEVE(SYMT(POP(stk)), id).
```

Fig. 6. Mapping data type.

type Mapping[domaintype:Type, rangetype:Type]

syntax

```
NEWMAP → Mapping,
DEFMAP(Mapping, domaintype, rangetype) → Mapping,
EVMAP(Mapping, domaintype) →
  rangetype ∪ {UNDEFINED},
ISDEFINED(Mapping, domaintype) → Boolean.
```

semantics

```
declare map:Mapping, dval, dval1:domaintype,
  rval:rangetype;
EVMAP(NEWMAP, dval) = UNDEFINED,
EVMAP(DEFMAP(map, dval, rval), dval1)
  = IF dval = dval1 THEN rval
    ELSE EVMAP(map, dval1),
ISDEFINED(NEWMAP, dval1) = FALSE,
ISDEFINED(DEFMAP(map, dval, rval), dval1)
  = IF dval = dval1 THEN TRUE
    ELSE ISDEFINED(map, dval1).
```

4. Proving Correctness

In this section we turn to the problem of proving correctness of implementations of data types. We shall continue to center the discussion upon the example of the Symboltable data type, showing parts of the proof of correctness of the example implementation given in Section 3.

One of the most important aspects of the proof techniques used to prove correctness of algebraically specified data types, is that the proofs are factored into levels corresponding to the implementation levels. To prove the correctness of a data type implemented in terms of other data types, we need to rely only on the axiomatic specifications of the other data types, *not* on their implementations. For example, as we will show in Section 4.2, to verify the top level of the implementation of the Symboltable data type, we use the semantic axioms for the Stack and Mapping data types and ignore their implementations. In fact, we could (and, in general, would) have proceeded with the proof of the top level before implementing the Stack or Mapping data types.

Another highly significant aspect of the use of axioms for other data types in the proof of an implementation, is the computational nature of the proof steps: the axioms are used as rewrite rules and proofs proceed via a series of reductions. Therefore, to a large extent, the proof process can be automated. In Section 5 we shall discuss some of the details of an interactive system we are developing to verify implementations of data types.

4.1 Formal Deduction Using The Boolean Data Type

Although we will not be completely formal in the example proofs in the following sections, it is useful to discuss at this point the basis we have chosen for automating such proofs, since it fits well with the overall approach of using algebraically specified data types. In fact, many of the formal deductions will be based on the algebraic specification of the Boolean data type given in Figure 7.

In this specification, each of the usual logical operators is related by an axiom to the IF-THEN-ELSE operator which is axiomatized by relating it to TRUE and FALSE in the first two axioms. The use of ∧, ∨, etc. in infix rather than prefix form is purely for convenience. Although the above specification defines IF-THEN-ELSE only for Boolean operands, we assume that every data type T1 can use IF-THEN-ELSE with the syntactic specification

```
(IF Boolean THEN T1 ELSE T1) → T1
```

and the same axioms as the first two in the above specification.

We shall make frequent use of the following rewrite rules for IF-THEN-ELSE, which are theorems provable from the Boolean axioms:

```
1. [Repeated result rule]
     (IF p THEN q ELSE q) = q
2. [Redundant IF rule]
     (IF p THEN TRUE ELSE FALSE) = p
3. [IF-distribution rule]
     (IF(IF p THEN q ELSE r) THEN a ELSE b)
       = IF p THEN (IF q THEN a ELSE b)
         ELSE (IF r THEN a ELSE b)
4. [Logical substitution rules]
     (IF p THEN q[p] ELSE r)
       = IF p THEN q[TRUE for p] ELSE r)
     (IF p THEN q ELSE r[p])
       = (IF p THEN q ELSE r[FALSE for p]).
```

In the left-hand side of a rule, "q[p]" means q must be an expression in which p occurs as a subexpression (possibly p = q). In the right-hand side, "q[TRUE *for* p]" is the result of substituting TRUE for all occurrences of p in q. We require that p occur as a subexpression of q to limit applicability of the rule to those places where

Fig. 7. Boolean data type.

type Boolean

syntax

TRUE → Boolean,
FALSE → Boolean,
(IF Boolean THEN Boolean ELSE Boolean) → Boolean,
(Boolean ∧ Boolean) → Boolean,
(Boolean ∨ Boolean) → Boolean,
¬Boolean → Boolean,
(Boolean ⊃ Boolean) → Boolean,
(Boolean ≡ Boolean) → Boolean.

semantics

declare p, q, r: Boolean;
(IF TRUE THEN q ELSE r) = q,
(IF FALSE THEN q ELSE r) = r,
(p ∧ q) = IF p THEN q ELSE FALSE,
(p ∨ q) = IF p THEN TRUE ELSE q,
¬p = IF p THEN FALSE ELSE TRUE,
(p ⊃ q) = IF p THEN q ELSE TRUE,
(p ≡ q) = IF p THEN q ELSE ¬q,

it will effect a change. Rules 1–4 are also theorems for IF-THEN-ELSE in other data types.

As noted in [16] and [1], the Boolean axioms and rules 1–4 combine to yield a system for simplifying expressions of the propositional calculus. It is easily shown that the Boolean axioms and rules 1–4 are complete with respect to the propositional calculus in that any valid expression in propositional calculus (i.e. provable by truth table) is reducible to TRUE by use of these rewrite rules. These rules form the basis for an automatic simplifier for logical expressions which is described in Section 5.

It is, of course, necessary to go beyond propositional calculus and include deductive rules for equality and other operators. We will not go into the details of the formal rules here except to mention the following important rule:

5. [Case analysis rule]
   $f(a_1, \dots, (\text{IF } p \text{ THEN } x \text{ ELSE } y), \dots, a_n)$
   $= \text{IF } p \text{ THEN } f(a_1, \dots, x, \dots, a_n)$
   $\text{ELSE } f(a_1, \dots, y, \dots, a_n)$
   when f ≠ IF-THEN-ELSE.

This is a "second-order" rewrite rule and the rule applies to IF-THEN-ELSE expressions in any operand position. The case in which f is IF-THEN-ELSE is already covered in part by the IF-distribution rule. Note that if f were permitted to be IF-THEN-ELSE, then the expression IF a_1 THEN (IF p THEN x ELSE y) ELSE a_3 could be transformed to IF p THEN (IF a_1 THEN x ELSE a_3) ELSE (IF a_1 THEN y ELSE a_3) and the rule would apply again, leading to an infinite sequence of applications. An important application of this case analysis rule occurs when f is "=," e.g.

((IF p THEN x ELSE y) = z) == [by case analysis rule] ⇒
(IF p THEN (x = z) ELSE (y = z)).

4.2 Verification of One of the Symboltable Axioms

The basic proof technique for verifying an implementation of a data type is to show that each of the axiomatic specifications for the data type is satisfied when the programs for the operations of the data type are substituted into the axioms. As our first example, consider the ninth axiom for the Symboltable data type:

```
RETRIEVE(ADDID(symtab, id, attrlist), id1)
  = IF id = id1 THEN attrlist
             ELSE RETRIEVE(symtab, id1).          (1)
```

To start, we assume there exists a stack, stk, such that symtab = SYMT(stk). (We will show in Section 4.3 how to verify this assumption.) Substituting, we obtain the verification condition

```
RETRIEVE(ADDID(SYMT(stk), id, attrlist), id1)
  = IF id = id1 THEN attrlist
             ELSE RETRIEVE(SYMT(stk), id1).       (2)
```

The goal now is to show that this equation is true using the programs for RETRIEVE and ADDID (Figure 5) and the axioms for the Stack and Mapping data types (Figures 1 and 6) as rewrite rules. We must also use the axioms and some theorems for the Boolean data type, as discussed in Section 4.1.

Working first on the left-hand side of (2), we make the following reductions:

```
LHS == [by ADDID program] ⇒
  RETRIEVE(SYMT(stk1), id1)
    where stk1 = REPLACE(stk, DEFMAP(TOP(stk), id, attrlist))

== [by RETRIEVE program] ⇒
  IF ISNEW(stk1)
    THEN UNDEFINED
    ELSE IF ISDEFINED(TOP(stk1), id1)
           THEN EVMAP(TOP(stk1), id1)
           ELSE RETRIEVE(SYMT(POP(stk1)), id1)

== [by REPLACE, ISNEW, POP and TOP axioms (and definition of
   stk1)] ⇒
  IF ISDEFINED(DEFMAP(TOP(stk), id, attrlist), id1)
    THEN EVMAP(DEFMAP(TOP(stk), id, attrlist), id1)
    ELSE RETRIEVE(SYMT(POP(stk)), id1)

== [by ISDEFINED axiom] ⇒
  IF (IF id = id1
    THEN TRUE ELSE ISDEFINED(TOP(stk), id1))
      THEN EVMAP(DEFMAP(TOP(stk), id, attrlist), id1)
      ELSE RETRIEVE(SYMT(POP(stk)), id1)

== [by IF-distribution Rule] ⇒
  IF id = id1
    THEN IF TRUE
           THEN EVMAP(DEFMAP(TOP(stk), id, attrlist), id1)
           ELSE RETRIEVE(SYMT(POP(stk)), id1)
    ELSE IF ISDEFINED(TOP(stk), id1)
           THEN EVMAP(DEFMAP(TOP(stk), id, attrlist), id1)
           ELSE RETRIEVE(SYMT(POP(stk)), id1)

== [by IF axiom] ⇒
  IF id = id1
    THEN EVMAP(DEFMAP(TOP(stk), id, attrlist), id1)
    ELSE IF ISDEFINED(TOP(stk), id1)
           THEN EVMAP(DEFMAP(TOP(stk), id, attrlist), id1)
           ELSE RETRIEVE(SYMT(POP(stk)), id1)
```

```
== [by EVMAP axiom] ⇒
  IF id = id1
    THEN (IF id = id1 THEN attrlist
      ELSE EVMAP(TOP(stk), id1))
    ELSE IF ISDEFINED(TOP(stk), id1)
            THEN (IF id = id1 THEN attrlist
              ELSE EVMAP(TOP(stk), id1))
            ELSE RETRIEVE(SYMT(POP(stk)), id1)

== [by Logical substitution rule] ⇒
  IF id = id1
    THEN (IF TRUE THEN attrlist
      ELSE EVMAP(TOP(stk), id1))
    ELSE IF ISDEFINED(TOP(stk), id1)
            THEN (IF FALSE THEN attrlist
              ELSE EVMAP(TOP(stk), id1))
            ELSE RETRIEVE(SYMT(POP(stk)), id1)

== [by IF axioms] ⇒
  IF id = id1
    THEN attrlist
    ELSE IF ISDEFINED(TOP(stk), id1)
            THEN EVMAP(TOP(stk), id1)
            ELSE RETRIEVE(SYMT(POP(stk)), id1)
```

Having already substituted once for RETRIEVE, we do not substitute again for the recursive call. We need now only show that the right-hand side of (2) reduces to the same expression. We begin by substituting in the implementation of RETRIEVE.

```
== [by RETRIEVE program] ⇒
  IF id = id1 THEN attrlist
    ELSE IF ISNEW(stk)
            THEN UNDEFINED
            ELSE IF ISDEFINED(TOP(stk), id1)
                    THEN EVMAP(TOP(stk), id1)
                    ELSE RETRIEVE(SYMT(POP(stk), id1).
```

At this point we observe that no further reductions are possible, yet we do not have the expression to which the left-hand side was reduced. To reduce this expression to the desired form, we need to prove the lemma: ISNEW(stk) = FALSE. How such a lemma is proved is discussed in Section 4.3. At this juncture, let us merely assume the lemma and use it immediately to reduce the above expression to the expression derived for the left-hand side. Equation (2) has thus been shown to be true, and axiom (1) verified for the implementation.

We have shown this proof in detail to illustrate its largely straightforward nature. Up to the use of the lemma ISNEW(stk) = FALSE, each step was merely the application of a rewrite rule. The rules could have been applied in a different order, but this would not have altered the final result since, if the axioms are complete as rewrite rules, all possible reduction sequences will terminate with the same result. It should be noted that to ensure rewrite rule completeness, we restrict the form that the axioms may take [17]. We must also take care in the unfolding of recursive implementations. This is not done automatically, but only under user control.

Verification of the axioms in the manner of the above proof establishes only *partial correctness* of the implementation, i.e. that if the programs terminate they give results satisfying the axioms. Proof of termination must be done separately. However, implementations of data types are often simple enough that termination is obvious, and we shall not deal with the issue of formal proofs of termination in this paper.

4.3 Data Type Induction

In the proof of axiom (2) in the previous section, we made use of the unproved lemma:

ISNEW(stk) = FALSE. (3)

To prove (3), we recall that stk is not just an arbitrarily chosen stack, but one assumed to be generated as a representation of a symbol table. If we examine the syntactic specification of the Symboltable data type, we see that the only operations which produce symbol tables as their output are INIT, ENTERBLOCK, ADDID, and LEAVEBLOCK. Examining the program for each of these operations, we see that INIT generates an initial stack, stk, for which (3) is true, and that if (3) is true of the stack representing the symbol table argument of any of the other operations, then it is true of the stack produced in the result. Therefore, (3) must be true of all stacks produced as representations of symbol tables by operations of the Symboltable data type.

The general principle being used in the above proof is that of *data type induction* (called "generator induction" in [20] and [23]). Paraphrasing the discussion in [20, p. 141], we suppose that a data type T has, according to its syntactic specification, exactly the operations F_1, ..., F_t whose range is the set of values of T. Let P(x) be a property of values of type T. Then if the truth of P for arguments of type T of each F_i implies the truth of P for the results of calls of F_i allowed by the syntactic specification of T, then it follows that P is true of all values of the data type. If strong type-checking is assumed, the validity of this rule follows by induction on the number of computation steps involving values of type T. As Spitzen and Wegbreit point out, the data type induction principle "is analog to the principle of complete induction over the integers. As with complete induction, one of the results which must be established is the base step, that P is true of the results of those primitives F with no arguments of type T." In the case of symbol tables, INIT is the only such primitive.

Let us examine more carefully the proof of (3) by data type induction. We can regard any property, P_1, that we wish to prove about symbol tables by data type induction as an operation with the syntactic specification:

P_1(Symboltable) → Boolean,

and the semantic specifications:

P_1(INIT)
P_1(symtab) ⊃ P_1(ENTERBLOCK(symtab))
P_1(symtab) ⊃ P_1(ADDID(symtab, id, attrlist))
P_1(symtab) ⊃ P_1(LEAVEBLOCK(symtab)). (4)

These specifications can be generated automatically from the syntactic specification of the Symboltable data type.

To prove (3), we let the interpretation of P_1 in terms of the implementation values be:

$$P_1(\text{SYMT(stk)}) = (\text{ISNEW(stk)} = \text{FALSE}). \quad (5)$$

We then prove each of the conditions in (4) using this interpretation of P_1 and the implementation programs of the Symboltable operations. For example, the fourth condition becomes:

```
P1(SYMT(stk)) ⊃ P1(LEAVEBLOCK(SYMT(stk)))

== [by (5) and LEAVEBLOCK program] ⇒
  (ISNEW(stk) = FALSE)
    ⊃ P1(IF ISNEW(POP(stk))
      THEN SYMT(REPLACE(stk, NEWMAP))
        ELSE SYMT(POP(stk)))

== [by case analysis rule] ⇒
  (ISNEW(stk) = FALSE)
    ⊃ (IF ISNEW(POP(stk))
      THEN P1(SYMT(REPLACE(stk, NEWMAP)))
        ELSE P1(SYMT(POP(stk)))

== [by (5)] ⇒
  (ISNEW(stk) = FALSE)
    ⊃ (IF ISNEW(POP(stk))
      THEN ISNEW(REPLACE(stk, NEWMAP)) = FALSE
        ELSE ISNEW(POP(stk)) = FALSE)

== [by ISNEW axiom and logical substitution rule] ⇒
  (ISNEW(stk) = FALSE)
    ⊃ (IF ISNEW(POP(stk))
      THEN FALSE = FALSE
        ELSE FALSE = FALSE)
```

which reduces to TRUE with the application of the reflexive property of Boolean equality, the repeated result rule, and the ⊃ axiom.

If the proof of a property P of a data type requires interpretation of P in terms of the implementation, then it is called an *implementation invariant.* Thus [with the establishment of the other three conditions in (4)], we have almost completed showing that (3) is an implementation invariant of the Symboltable data type. What remains to be shown is that

$$P(\text{symtab}) = (\exists\ \text{stk} \in \text{Stack such that symtab} = \text{SYMT(stk)}) \quad (6)$$

holds always. Again, this can easily be verified using data type induction. The lemma (6) is an example of a *representation invariant,* which we define to be that implementation invariant which describes how the abstract values are represented. The representation invariant can be constructed automatically from the *representation* part of the implementation.

The *normal form lemma* for a data type is the representation invariant for its direct implementation. Consider, for example,

```
P(symtab) = (symtab = INIT)
  ∨ (∃ symtab1 such that symtab = ENTERBLOCK(symtab1))
  ∨ (∃ symtab1, id, attrlist such that
      symtab = ADDID(symtab1, id, attrlist)).          (7)
```

This can be shown to satisfy each of the conditions in (4) by using the syntactic specification and axioms for the Symboltable data type to demonstrate that LEAVEBLOCK is an *extension,* i.e. it does not allow us to generate any values of type Symboltable that cannot be generated using only INIT, ENTERBLOCK, and ADDID. For a complete discussion of this see [8]. Normal form lemmas are useful in proofs which make use of the data type, in that they can be used to reduce the number of cases that must be considered in a proof by case analysis. In the proof of (5), for example, (7) tells us that we need not consider the fourth formula in (4).

4.4 Interpretation of the Equality Operator

Another important consideration in proving that an implementation satisfies an axiom is the interpretation of the equal sign in the axiom in terms of the implementation. To illustrate this, consider the axiom for stacks, POP(PUSH(stk, elm)) = stk, and the implementation of Figure 2. We assume there exists an array, arr, and an integer, t, such that stk = STAK(arr, t). Substituting this into the axiom, we obtain

```
POP(PUSH(STAK(arr, t), elm)) = STAK(arr, t)

== [by PUSH program] ⇒
  POP(STAK(ASSIGN(arr, t + 1, elm), t + 1)) = STAK(arr, t)

== [by POP program and integer theorem ((t + 1) − 1 = t)] ⇒
  (IF t + 1 = 0 THEN STAK(ASSIGN(arr, t + 1, elm), 0)
    ELSE STAK(ASSIGN(arr, t + 1, elm), t)) = STAK(arr, t).
```

Assuming $t \geq 0$, which can be proved as an implementation invariant by the methods of Section 4.3, we can make a further reduction so that the equation becomes

$$\text{STAK(ASSIGN(arr, t + 1, elm), t)} = \text{STAK(arr, t)}. \quad (8)$$

What we now need to establish the validity of (8), is a proof that STAK(ASSIGN(arr, t + 1, elm), t) and STAK(arr, t) are indistinguishable by any sequence of operations mapping us out of type Stack. To do this we introduce an *equality interpretation* for this representation of type Stack. To the implementation of Figure 3 we thus add

```
equality interpretation
  (STAK(arr, t) = STAK(arr1, t1))
    = (t = t1) ∧ ∀k(1 ≤ k ≤ t ⊃ ACCESS(arr, k)
    = ACCESS(arr1, k))                                  (9)
```

Using (9), (8) becomes:

```
(t = t) ∧ ∀k(1 ≤ k ≤ t ⊃ ACCESS(ASSIGN(arr, t + 1, elm), k)
  = ACCESS(arr, k))
==[by equality axiom for integers and ACCESS axiom for arrays] ⇒
  ∀k(1 ≤ k ≤ t ⊃ (IF t + 1 = k THEN elm ELSE ACCESS(arr, k))
    = ACCESS(arr, k))
== [by inequality property of integers] ⇒
  ∀k(1 ≤ k ≤ t ⊃ (IF FALSE THEN elm ELSE ACCESS(arr, k))
    = ACCESS(arr, k))
```

which is reduced to TRUE by use of the IF and ⊃ axioms, the repeated result rule, and the property ∀k(TRUE) = TRUE.

As part of the verification of the implementation, it is necessary to prove that the chosen interpretation of equality of stacks has the properties of an equality operator. These properties are

1. (Reflexivity) $x = x$
2. (Symmetry) $x = y \supset y = x$
3. (Transitivity) $x = y \wedge y = z \supset x = z$
4. (Substitution) $x = y \supset p = p[y\ \textit{for}\ x]$ where p is any expression.

Properties 1–3 are easily proved for the equality interpretation (9) for stacks, since they reduce to the corresponding properties for equality in type elementtype. To prove the substitution property it suffices to show that the implementation satisfies

4a. stk = stk1 ⊃ PUSH(stk, elm) = PUSH(stk1, elm)
4b. stk = stk1 ⊃ POP(stk) = POP(stk1)
4c. stk = stk1 ⊃ TOP(stk) = TOP(stk1)
4d. stk = stk1 ⊃ ISNEW(stk) = ISNEW(stk1)
4e. stk = stk1 ⊃ REPLACE(stk, elm) = REPLACE(stk1, elm)

since, by the syntactic specification, Stack values can appear in expressions only as the arguments of PUSH, POP, TOP, ISNEW, or REPLACE operations. The proof of 4a is, letting stk = STAK(a, t), stk1 = STAK(b, u),

STAK(a, t) = STAK(b, u) ⊃ PUSH(STK(a, t), elm) = PUSH(STAK(b, u), elm)

== [by PUSH program and Stack equality interpretation] ⇒
  $[(t = u) \wedge \forall k(1 \le k \le t \supset \text{ACCESS}(a, k) = \text{ACCESS}(b, k))]$
    $\supset [(t + 1 = u + 1) \wedge \forall k(1 \le k \le t + 1$
      $\supset \text{ACCESS}(\text{ASSIGN}(a, t + 1, elm), k)$
        $= \text{ACCESS}(\text{ASSIGN}(b, u + 1, elm), k))]$
== [by substitution of u for t] ⇒
  $[(t = u) \wedge \forall k(1 \le k \le u \supset \text{ACCESS}(a, k) = \text{ACCESS}(b, k))]$
    $\supset \forall k(1 \le k \le u + 1 \supset \text{ACCESS}(\text{ASSIGN}(a, u + 1, elm), k)$
      $= \text{ACCESS}(\text{ASSIGN}(b, u + 1, elm), k)).$

The equation relating ACCESS expressions reduces to IF u + 1 = k THEN elm = elm ELSE ACCESS(a, k) = ACCESS(b, k), and thus the conclusion is reduced to the second hypothesis. We omit the details of the proofs of 4b–4e, but note that they have been proved automatically by the programs described in Section 5.

In the general case of a data type T with abstract operations $F_1, \ldots, F_n$, the verification conditions for the substitution property are

$$x = y \supset F_i(\ldots, x, \ldots) = F_i(\ldots, y, \ldots), \quad i = 1, \ldots, n$$

where the other arguments are held fixed.

Our use of equality interpretations is a generalization of an earlier method using *abstraction functions* [9]. An abstraction function is a function A(x) which maps representation values x onto the abstract values which they represent. As an example, an abstraction function for the implementation of the Stack data type by Array/Integer pairs can be defined by

A(arr, t) = IF t = 0 THEN NEWSTACK
    ELSE PUSH(A(arr, t − 1), ACCESS(arr, t)).

Given an abstraction function, an equality interpretation can be defined in terms of it, e.g.

(STAK(a, t) = STAK(b, u)) = (A(a, t) = A(b, u));

but the opposite is not true.

4.5 Summary and Comparison to Another Proof Technique

Let us now summarize the main steps of the proof procedure for verifying data type implementations using algebraic specifications. We suppose that we have an algebraic specification of a data type and an implementation expressed in terms of other data types for which we also have algebraic specifications. Then the main steps are as follows:

1. State the *representation invariant* of the implementation and prove it using data type induction (as explained at the end of Section 4.3).
2. State the *equality interpretation* of the implementation as discussed in Section 4.4.
3. Using the representation invariant, substitute the representation into the axioms of the data type and into the equality axioms (reflexive, symmetric, transitive, substitution), obtaining a set of verification conditions (see Section 4.2).
4. Prove each of the verification conditions, using as rewrite rules the programs of the implementation, the equality interpretation, and the axioms of the data types used in the programs (including the Boolean, Integer, etc. data types). In some cases, completion of a proof will require one or more assumptions to be made about the representation of the data types used in the implementation (see Section 4.2).
5. Prove that the assumptions made in step 4, or a stronger set of assumptions, are valid, using data type induction (Section 4.3).

In Section 5 we will discuss an interactive program which guides the user through steps 1, 3, 4, and 5, accomplishing many of the tasks automatically.

We conclude this section with a brief discussion of a technical difference between our proof technique and that of Zilles [25] and Goguen [3]. Regarding equality of the values of an abstract data type, we recall our basic assumption that the properties of the values can be derived solely from the relations determined by the axioms (Section 2.1). On the basis of this assumption, we conclude that two values *may* be assumed to be the same unless provably different. Zilles [25] and Goguen [3] make the opposite assumption, i.e. that values should be considered to be different unless they are demonstrably equal.

This difference in viewpoint is formally expressed in terms of the congruence relations defined by the axioms of the type. "The congruence relations used are the smallest congruence relations which contain all of the defining relations [axioms]. This means that two expressions are equivalent if and only if there is a consequence of expressions such that the first and last expressions are the expressions in question and every adjacent pair of expressions can be shown to be equivalent using some defining relation." [25] We, on the other hand, permit any congruence relation (including the smallest) consistent with the axioms. An example may help to clarify this distinction.

Consider the abstract type Symboltable as defined in Figure 4 and examine two possible Symboltable values:

S = ADDID(ADDID(INIT, Y, Integer), X, Real)
T = ADDID(ADDID(INIT, X, Real), Y, Integer).

Using the axioms, it cannot be shown that S = T; hence, under the smallest congruence viewpoint these values would be considered unequal. We prefer to regard S and T as equal, ignoring the difference in the order in which the identifiers are entered.

The practical ramifications of this difference in viewpoint are important. If one interprets the axioms as defining the smallest congruence relations, then the algebra defined by the axioms is unique up to isomorphism. All acceptable implementations of the abstract type must, therefore, yield algebras that are isomorphic to one another and to the original algebra. This observation is the basis of the technique used by both [25] and [3] to show the correctness of implementations of abstract data types.

This is a somewhat restrictive notion of correctness. It requires that abstract values that are not provably equal be mapped onto distinct concrete values. That this is a significant restriction may be easily shown through reference to our example. One efficient implementation of the Symboltable data type was achieved by using a hashing function (Figure 8). However, this implementation does not preserve the information necessary to distinguish between the expressions S and T; it is therefore not a correct implementation with respect to the smallest congruence definition of correctness.

It should be pointed out that this restrictive notion of correctness does have certain advantages. The requirement that no information be lost increases the likelihood of being able to add new operations to the type without having to make extensive changes to existing implementations. Implementations which conform to our notion of correctness do not, in general, exhibit this property.

It should also be pointed out that by simply adding the axiom

ADDID(ADDID(symtab, id, attrs), id1, attrs1)
    = ADDID(ADDID(symtab, id1, attrs1), id, attrs),

our hash table implementation becomes acceptable under the smallest congruence viewpoint. Unfortunately, commutative axioms such as the above lead to rewrite rules that once applied can always be applied to their result. This will not lead to a problem in verifying the correctness of an implementation of the type being defined (Symboltable in this example). Nor will it lead to a problem when verifications of programs that use the operations of the type are done by hand. In performing semi-automated verifications, however, there is a danger that the system will go into an infinite loop. To prevent this, it is necessary to recognize this possibility in advance, and mark this (or similar axioms) for special treatment. This special treatment basically involves restricting application of the rewrite rule derived from the axiom to occasions specifically requested by the user of the system.

5. Automatic Tools

The previous sections have presented a methodology for the specification of data types, and discussed the use of such specifications in design, implementation, testing, and verification. Much of the work involved in applying the methodology can be profitably automated with tools whose use will speed up the application of the methods and avoid the errors which would inevitably occur amidst the tedium of applying such straightforward steps by hand. While we would argue that the results of using the methodology are valuable enough that it should be used even if it must be done by hand, we hope to show in this section that useful automatic tools can be achieved with a modest investment of software development.

We first discuss in Section 5.1 the concepts of *direct implementations, expression data types,* and *reduction systems.* Direct implementations are useful for testing and in some cases can serve as actual implementations. Reduction systems are useful for carrying out proofs of correctness of applications of abstract data types. In Section 5.2 we discuss the realization of reduction systems and direct implementations with a simple *pattern-match compiler* based mainly on ideas from [13].

This pattern-match compiler is one of the main components of a "Data Type Verification System" (DTVS) which we have implemented for testing and verifying abstract data types. Another important component is a simple deductive system called CEVAL ("Conditional Evaluator"), which is used to carry out proofs of data type verification conditions, mainly as a series of reduction steps. The main part of the implementation of CEVAL is a Boolean Expression Reduction System obtained from the Boolean data type specification of Figure 7 and extended to include the basic Boolean theorems 1–4 as reduction rules. (This implementation was obtained by hand coding, but, except for some hand optimization, it could have been obtained by applying the pattern-match compiler of the system to the axioms and theorems of Section 4.1.) In addition to this basic knowledge of propositional calculus, CEVAL also incorporates the Case Analysis Rule of Section 4.1, and some basic rules for equality substitution, ordering properties, and quantifier and lambda-variable reduction.

5.1 Direct Implementations, Expression Data Types, and Reduction Systems

An important point about direct implementations is that they do not require any additions to the simple language for specifications used in Section 2. In fact, we define a *direct implementation* of a data type T to be an implementation whose representation part is a subset of the syntactic specification of T and whose program part is a subset of the semantic specification of T. The *maxi-*

mal direct implementation of a data type T is the implementation whose representation part is the entire syntactic specification of T and whose program part is the entire semantic specification for T.

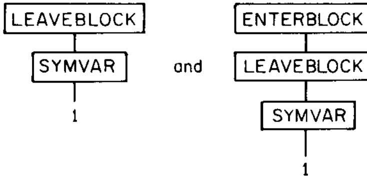
Direct implementations need not deal with terms containing free variables. In performing proofs of correctness, on the other hand, we must deal with variables whose values are not known. We begin by supplementing the syntactic specification of the Symboltable data type (Figure 4) with

SYMVAR(Integer) → Symboltable

but add no axioms relating the new operation, SYMVAR, to the other operations. By doing this, we obtain a different data type which can be regarded as a *Symboltable Expression data type.* This is because SYMVAR(1), SYMVAR(2), etc., can be regarded as *variables* of type Symboltable, and because the lack of axioms relating SYMVAR to other operations makes it necessary to include expressions such as

LEAVEBLOCK(SYMVAR(1)),
ENTERBLOCK(LEAVEBLOCK(SYMVAR(1)))

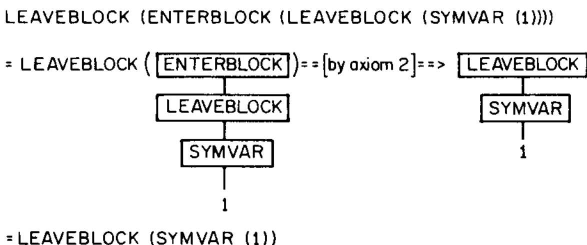
among the values of the data type. In terms of the tree structures introduced in Section 2 to explain direct implementations, the values of a direct implementation should include trees such as



as well as trees built from INIT, ENTERBLOCK, and ADDID nodes.

In general, for a given data type T and operation $F(D_1, \ldots, D_n) \rightarrow T$, we define the *T(F)-expression data type* to be the data type obtained by adding $F(D_1, \ldots, D_n) \rightarrow T$ to the syntactic specification of T. The *T(F)-expression reduction system* is defined to be the maximal direct implementation of the T(F)-expression data type.

The example discussed above is the Symboltable(SYMVAR) Expression data type. The Symboltable(SYMVAR) Expression Reduction System not only provides an implementation of the Symboltable data type, but also is capable of making reductions such as



Consequently, this reduction system can be used to carry out steps in proofs about applications of the Symboltable operations in, for example, proofs of properties of other parts of a compiler.

5.2 Compilation of Reduction Systems

We now turn to a more detailed discussion of how reduction systems (and therefore direct implementations) are realized by a pattern-match-compilation process (which we shall call PMC). The key to this process is the full exploitation of the duality between programs and axioms discussed in Section 2.2.

We shall confine ourselves in this paper to an informal description in terms of the Symboltable example. From the syntactic specification S of the Symboltable data type, PMC first produces a set of *node constructor operations* INITNODE, ENTERBLOCKNODE, ..., RETRIEVENODE, and *projection* (or *selector*) *operations* ENTERBLOCKA, ADDID_A, ADDID_B, ADDID_C, etc. The constructor operations have the same syntactic specification as the corresponding operations in S, while projection operations are from the constructor range type to one of the argument types of the constructor, e.g.

ADDID_NODE(Symboltable, Identifier, Attributelist) →
  Symboltable,
ADDID_A(Symboltable) → Symboltable,
ADDID_B(Symboltable) → Identifier,
ADDID_C(Symboltable) → Attributelist.

The projection operations satisfy the semantic axioms implied by their name, e.g.

ADDID_A(ADDID_NODE(symtab, id, attrlist)) = symtab,
ADDID_B(ADDID_NODE(symtab, id, attrlist)) = id,
ADDID_C(ADDID_NODE(symtab, id, attrlist)) = attrlist

PMC also constructs a *node discriminator operation*,

NODEKIND(Symboltable) →
  {INITOP, ENTERBLOCKOP, ADDID_OP, LEAVEBLOCKOP}

such that

NODEKIND(INITNODE) = INITOP
NODEKIND(ENTERBLOCKNODE(symtab)) = ENTERBLOCKOP
NODEKIND(ADDID_NODE(symtab, id, attrlist)) = ADDID_OP
NODEKIND(LEAVEBLOCKNODE(symtab)) = LEAVEBLOCKOP.

The actual implementation of these constructor, projection, and discriminator operations could of course be any of a variety of implementations which satisfy these axioms, e.g. with Lisp operations:

ADDID_NODE(symtab, id, attrlist)
  = LIST(ADDID_OP, symtab, id, attrlist),
ADDID_A(symtab) = CADR(symtab),
ADDID_B(symtab) = CADDR(symtab),
ADDID_C(symtab) = CADDDR(symtab),
NODEKIND(symtab) = CAR(symtab).

Essentially this implementation is used in our Data Type

Verification System. Alternatively, each of these operations could be implemented by machine language code for allocating memory blocks of one or more words and accessing fields within these blocks, as discussed in [11].

PMC then proceeds to translate the semantic axioms into programs operating on the nodes. The translation is done incrementally (axiom by axiom). Each operation is first given an initial program definition, in terms of only its corresponding node constructor:

```
INIT = INITNODE
ENTERBLOCK(symtab) = ENTERBLOCKNODE(symtab)
etc.
```

Then, as each axiom is processed, it is used to modify the existing program of one of the operations. The first axiom, for example, LEAVEBLOCK(INIT) = INIT, is used to modify the program for LEAVEBLOCK to become

```
LEAVEBLOCK(symtab)
  = IF NODEKIND(symtab) = INITOP
    THEN INIT
    ELSE LEAVEBLOCKNODE(symtab).
```

From the next axiom, LEAVEBLOCK(ENTERBLOCK(symtab)) = symtab, PMC produces

```
LEAVEBLOCK(symtab)
  = IF NODEKIND(symtab) = ENTERBLOCKOP
  THEN ENTERBLOCKA(symtab)
    ELSE IF NODEKIND(symtab) = INITOP
      THEN INIT
      ELSE LEAVEBLOCKNODE(symtab).
```

The third axiom, LEAVEBLOCK(ADDID(symtab, id, attrlist)) = LEAVEBLOCK(symtab), produces

```
LEAVEBLOCK(symtab)
  = IF NODEKIND(symtab) = ADDID_OP
    THEN LEAVEBLOCK(ADDID_A(symtab))
    ELSE IF NODEKIND(symtab) = ENTERBLOCKOP
      THEN LEAVEBLOCKA(symtab)
      ELSE IF NODEKIND(symtab) = INITOP
        THEN INIT
        ELSE LEAVEBLOCKNODE(symtab).
```

Note the recursive call in this program. The next three axioms lead to a similar program for ISINBLOCK, and the final three determine the RETRIEVE program. (Although the axioms are grouped according to the main operator of the left-hand side, this is not necessary.) In general, given an axiom LHS = RHS, PMC modifies the program for F, where F is the main operator of LHS, from its existing program body B to a new body of the form

```
IF P THEN RHS' ELSE B,
```

where P is a test for the argument pattern of LHS, and RHS′ is obtained from RHS by substituting node projection operations for composite parameters in LHS. If no composite argument occurs in LHS (as in the case of the Stack axiom REPLACE(stk, elm) = PUSH(POP(stk), elm)), the new program body is simply RHS.

The code output by this straightforward approach is often not very efficient because it may contain redundancy in the pattern tests. CEVAL can then be used to remove these redundant tests. After this has been done, the IF-THEN-ELSE expressions that remain can then be replaced by *case* selection constructs where appropriate. We are currently experimenting with these optimizations in our Data Type Verification System.

5.3 A Data Type Verification System

The Data Type Verification System under development at the USC Information Sciences Institute accepts specifications and implementations of data types and performs three classes of operations on them: (1) maintaining a "database for data types," i.e. storage and retrieval for display or manipulation; (2) compiling direct implementations and expression reduction systems from the specifications; (3) carrying out proofs about data types, particularly proofs that given implementations of a data type satisfy its specification. Input to the system is first processed by a command interpreter, which permits interactive, incremental use of the system. (A file read command also permits commands to be batched on a file.) Specification and implementations are accepted in a syntax very close to that used in this paper. All expressions in user input are subjected to strong type checking using declarations of variables and interface specifications of operations. During type checking, operators are renamed to internal names. This device permits generic operators and avoids conflict at later stages with names of Interlisp functions. For equations in the user input (those expressing axioms and programs), the resulting internal form is then fed into a simple pattern-match compiler based on the ideas of Section 5.2. Thus a set of Interlisp functions is obtained which may then be compiled by the Interlisp compiler into machine code. Execution of this code or of the original functions (via the EVAL interpreter) has the effect of application of the original equations as rewrite rules. Presently, we are building a large database of data type specifications and implementations, both in the form of Interlisp files containing the EVAL and compiled versions of the functions output by the pattern-match compiler and runable core images containing many different types.

The facilities for testing using direct implementations and proving using expression reduction systems have been used for testing and carrying out proofs about a number of data types. To verify our implementation of Symboltable in terms of a stack of mappings, for example, the user inputs the Symboltable, Stack, and Mapping specifications and the Symboltable implementation. He then directs the system to generate the verification conditions for the implementation. These would consist of the Symboltable axioms and the equality axioms for the Symboltable equality operator (see Section 4.4), all interpreted in terms of the representation.

The user then attempts to prove each of the verification conditions using CEVAL. In these proofs the rewrite rules from the Symboltable programs and Stack

and Mapping axioms are used automatically, without further direction from the user. In some cases, as noted in Section 4, completion of a proof will require one or more assumptions to be made about the representation or about the Stack or Mapping data types. If this is the case, the system will stop with a reduced form of the original verification condition. Examination of this output will often lead the user to the necessary assumptions which are input by the user and used as needed without justification. To complete the verification of the implementation, it is necessary to prove these assumptions, or a stronger set of assumptions, as theorems (of the Symboltable data type implementation or of the Stack or Mapping data types). The proof system keeps track of the status of the proof of each of the original verification conditions and the assumptions made, plus enough other information that the current stage of the proof can be recreated automatically by a "recheck" command. By making a transcript tracing the operation of the proof system during a recheck of a completed proof, one obtains detailed documentation of the proof, unobscured by proof steps that did not contribute to the final result.

6. Summary

Elsewhere it has been argued that abstract data types can be effectively employed as a "thought tool" in the structured development of programs ([7], [21], [24], [14]). In this paper we have attempted to show that the use of algebraic axioms as a means for describing data abstractions is also valuable, when properly used, for both formal and informal program validation.

In Sections 2 and 3 we discussed the axiom language and some techniques for constructing algebraic axiomatizations. A complete implementation of a moderately complex symbol table was developed to show how these specifications might be used in practice. In Section 4 we demonstrated how properly written axioms can be used in formal program verification. Abstract data types provide a mechanism for factoring proofs into manageable sections. At one level of abstraction, the axiomatic specification provides us with theorems that may be applied in the verification of programs that use abstract data types. Writing the axioms in a certain style allows them to be used as reduction rules so that the proofs become largely symbol manipulation exercises. We have illustrated how such axioms may be used to verify the correctness of implementations of higher-level abstract data types in terms of lower-level ones.

In Section 5 we have described and given the formal basis for the symbol manipulation processes needed to prove correctness and to provide direct implementations. This system can carry out large parts of the proofs without a theorem prover; therefore, the cost of execution and the software sophistication normally required for such proofs is substantially reduced. Furthermore, this system is able to exploit the duality between axioms and programs as described in Section 3. A direct implementation of a data type is achieved by using the axioms. This allows for the interpretive execution of programs that make use of algebraically axiomatized data types. The coupling of early testing with proofs of correctness within the same automated framework is a valuable tool in the programming process.

Appendix. Hash-Table Implementation of Mapping Data Type

The Mapping data type has many implementations that would be reasonably efficient for the Symboltable application, e.g. using balanced tree structures or hash tables. Figure 8 depicts an implementation using a hash table. The hierarchical relationship of the types used in this implementation is reflected in Figure 9.

Note that the parameters domaintype and rangetype have been bound to Identifier and Attributelist. A hash table can be viewed abstractly as a composition of mappings, of which one is implementable by "random access" techniques to provide efficiency in searching. The actual implementation of the Mapping data type is given in Figure 10. In this implementation we have assumed that Arrays with integer domains have random access characteristics. We have left the other mappings abstract; i.e. Mapping1 is a data type identical to Mapping, except for renaming. For convenience we have assumed an operation for initializing an Array:

```
INITIALIZE(Array[Integer, Attributelist], Integerrange,
  Mapping1[Identifier, Attributelist]) →
  Array[Integer, Mapping1[Identifier, Attributelist]]
```

with the axiom

```
declare arr: Array, irange: Integerrange, int: Integer,
                              map1: Mapping1[Identifier, Attributelist];
ACCESS(INITIALIZE(arr, irange, map1), int)
  = IF ISIN(int, irange)
    THEN map1
    ELSE ACCESS(arr, int)
```

Thus INITIALIZE is a generalization of the ASSIGN operator. In most programming languages, no INITIALIZE operator for arrays is provided, but, of course, it is easily implemented with a loop over the elements of irange.

Regarding the HASH operation, we assume only that its syntactic specification is

```
HASH(Identifier) → Hashrange
```

for some particular range Hashrange of Integer values. This is sufficient to show the correctness of the implementation of Figure 10; the distribution of identifiers over the range is of concern only as a matter of efficiency.

We will use a direct implementation of the Mapping1

Fig. 8. Symboltable implementation via hash tables.

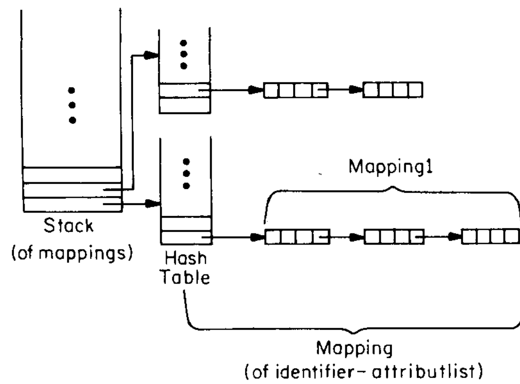


Fig. 9. Hierarchy of types in Symboltable implementation.

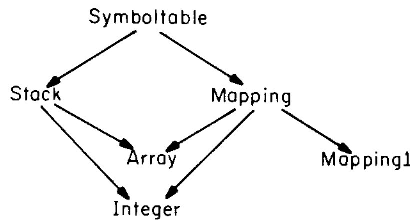


Fig. 10. Implementation of Mapping data type with a hash table.

representation

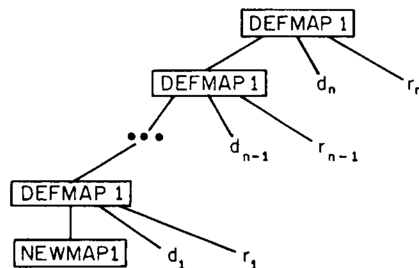
MAPP(Array[Integer, Mapping1[Identifier, Attributelist]]) → Mapping[Identifier, Attributelist]

programs

```
declare arr:Array, id:Identifier, attrlist:Attributelist;

NEWMAP = MAPP(INITIALIZE(NEWARRAY,
  Hashrange, NEWMAP1)),
DEFMAP(MAPP(arr), id, attrlist)
  =MAPP(ASSIGN(arr, HASH(id),
DEFMAP1(ACCESS(arr, HASH(id)), id, attrlist))),
EVMAP(MAPP(arr), id)
  = EVMAP1(ACCESS(arr, HASH(id)), id),
ISDEFINED(MAPP(arr), id)
  = ISDEFINED1(ACCESS(arr, HASH(id)), id).
```

data type, representing a Mapping1 value with a tree structure of the form



where the d_i are domain values and the r_i are range values.

For the implementation of the Stack data type we could use the implementation with Array/Integer records, as already given in Figure 2. Another possibility is the direct implementation using tree structures composed of PUSH and NEWSTACK nodes, as discussed in Section 2.2.

In a complete implementation the Identifier and Attributelist data types also must be dealt with, but we shall ignore them here, as we do not regard them as being part of the Symboltable data type. (The Symboltable implementation as we have given it requires very little interaction with the Identifier data type—only the HASH and Identifier equality operations interact—and essentially no interaction with the Attributelist data type.) Thus under the assumptions discussed at the beginning of Section 3, the design of the implementation is complete.

Acknowledgments. We would like to thank Dick Jenks, Ralph London, Nancy Lynch, Mark Moriconi, Jernej Polajnar, Dave Wile, Marty Yonke, and the referees for their careful reading of earlier drafts of this paper.

Received July 1976; revised May 1978

References

1. Boyer, R.S., and Moore, J.S. Proving theorems about LISP functions. *J. ACM 22*, 1 (January 1975), 129–144.
2. Dahl, O.-J. The SIMULA 67 common base language. Norwegian Comput. Ctr., Oslo, 1968.
3. Goguen, J.A., Thatcher, J.W., Wagner, E.G., and Wright, J.B. Abstract data-types as initial algebras and correctness of data representations. Proc. Conf. on Comptr. Graphics, Pattern Recognition and Data Structure, May 1975.
4. Good, D.I., London, R.L., and Bledsoe, W.W. An interactive program verification system. *IEEE Trans. Software Eng. SE-1*, 1 (March 1975), 56–67.
5. Guttag, J.V., Horowitz, E., and Musser, D. The design of data type specifications. In *Current Trends in Programming Methodology*, R.T. Yeh, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1978, pp. 60–79.
6. Guttag, J.V., Horowitz, E., and Musser, D.R. Some extensions to algebraic specifications. Proc. Language Design for Reliable Software, March 1977.
7. Guttag, J.V. Abstract data types and the development of data structures. *Comm. ACM 20*, 6 (June 1977), 396–404.
8. Guttag, J.V., and Horning, J.J. The algebraic specification of abstract data types. *Acta Informatica 10*, 1 (1978), 27–52.
9. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica* 4 (1972), 271–281.
10. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica 2* (1973), 335–355.
11. Hoare, C.A.R. Recursive data structures. *Int. J. Comptr. and Inform. Sci. 4*, 2 (June 1975), 105–132.
12. Horowitz, E., and Sahni, S. *Fundamentals of Data Structures.* Computer Science Press, June 1976.
13. Jenks, R.D. The SCRATCHPAD language. Proc. of ACM SIGPLAN Symp. on Very High Level Languages. *SIGPLAN Notices* 9, 4 (April 1974), 101–111.
14. Liskov, B.H., Snyder, A., Atkinson, R., and Shaffert, C. Abstraction mechanisms in CLU. *Comm. ACM 20*, 8 (Aug. 1977), 564–576.
15. Manna, Z. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.
16. McCarthy, J. Basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirchberg, Eds., North-Holland Publ. Co., Amsterdam, 1963, pp. 33–70.
17. Musser, D.R. A data type verification system based on rewrite

rules. Proc. of Sixth Texas Conf. of Comput. Syst., Austin, Tex., Nov. 1977.
18. Palme, J. Protected program modules in SIMULA 67. FOAP Rep. No. C8372-M3(E5), Research Inst. of National Defense, Stockholm, 1973.
19. Parnas, D. L. Information distribution aspects of design methodology. *Information Processing 71*, 1 (1972), North-Holland Pub. Co., Amsterdam, 339–344.
20. Spitzen, J., and Wegbreit, B. The verification and synthesis of data structures. *Acta Informatica 4* (1975), 127–144.
21. Standish, T.A. Data structures: an axiomatic approach. BBN Rep. No. 2639, Bolt Beranek and Newmann, Cambridge, Mass., 1973.
22. Suzuki, N. Automatic verification of programs with complex data structures. Ph.D. Th., Comptr. Sci. Dept., Stanford, U., Rep. No. STAN-CS-76-552, Feb. 1976.
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. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Eng. SE-2*, 4 (December 1976), 253–265.
25. Zilles, S. N. Abstract specifications for data types. IBM Res. Lab., San Jose, Calif., 1975.

Programming Languages — J.J. Horning, Editor

An Example of Hierarchical Design and Proof

Jay M. Spitzen, Karl N. Levitt, and Lawrence Robinson
SRI International

Hierarchical programming is being increasingly recognized as helpful in the construction of large programs. Users of hierarchical techniques claim or predict substantial increases in productivity and in the reliability of the programs produced. In this paper we describe a formal method for hierarchical program specification, implementation, and proof. We apply this method to a significant list processing problem and also discuss a number of extensions to current programming languages that ease hierarchical program design and proof.

Key Words and Phrases: program verification, specification, data abstraction, software modules, hierarchical structures
CR Categories: 4.0, 4.6, 5.21, 5.24

1. Introduction

The use of structuring techniques in programming—for example, programming by successive refinement [5] (also called hierarchical programming)—has been recognized as increasingly helpful in the design and management of large system efforts. A number of such design techniques are now promoted for routine use in com-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
Research supported by the Office of Naval Research (Contract N00014-75-C-0816), the National Science Foundation (Grant DCR74-18661), and the Air Force Office of Scientific Research (Contract F44620-73-C-0068).
Authors' present addresses: J.M. Spitzen, Advanced Systems Department, Xerox Corporation, 3333 Coyote Hill Road, Palo Alto, CA 94304; K.W. Levitt and L. Robinson, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.
© 1978 ACM 0001-0782/78/1200-1064 $00.75.