Acta Informatica 10, 27–52 (1978)

The Algebraic Specification of Abstract Data Types

J.V. Guttag*
Computer Science Department, University of Southern California, Los Angeles, CA 90007, USA

J.J. Horning**
Computer Systems Research Group, University of Toronto, Toronto, M5S 1A4, Canada

Summary. There have been many recent proposals for embedding abstract data types in programming languages. In order to reason about programs using abstract data types, it is desirable to specify their properties at an abstract level, independent of any particular implementation. This paper presents an algebraic technique for such specifications, develops some of the formal properties of the technique, and shows that these provide useful guidelines for the construction of adequate specifications.

1. Introduction

The class construct of SIMULA 67 [5] has been used as the starting point for much of the more recent work on embedding abstract types in programming languages, e.g., [15, 18, 19, and 25]. 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 behaviour 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 Integers) with the operations

NEW:	$\rightarrow$ Queue
ADD:	Queue $\times$ Integer $\rightarrow$ Queue
FRONT:	Queue $\rightarrow$ Integer
REMOVE:	Queue $\rightarrow$ Queue
EMPTY?	Queue $\rightarrow$ Boolean

* Supported in part by the National Science Foundation under grant MCS-76-06089 and the Joint Services Electronics Program monitored by the Air Force Office of the Scientific Research under contract F44620-72C-0061
** Supported in part by the National Research Council of Canada. *Current address:* Xerox Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA

0001-5903/78/0010/0027/$05.20

Unfortunately, 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 [20, 10]. 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.

In this paper we shall use the algebraic axiomatic technique developed by Guttag [7]. This approach owes much to the work of Hoare [12], and is closely related to the work of Zilles [26], Goguen, Thatcher, Wagner and Wright [6] and Spitzen and Wegbreit [22]. Its formal basis stems from the heterogeneous algebras of Birkhoff and Lipson [1].

An algebraic specification of an abstract type consists of two parts: a syntactic specification and a set of axioms. The syntactic specification provides the syntactic and type checking information that many programming languages already require: the names, domains, and ranges of the operations associated with the type. The set of axioms defines (in a sense to be discussed later) the meaning of the operations by stating their relationships to one another.

Spitzen [22] cites three attributes of this approach to the specification of data types:

> First, they are declarative and hence avoid programming details and language dependencies. Second, they are intuitively reasonable descriptions of the behaviour of various structures. Finally, they are sufficiently rigorous to permit a proof that a particular realization of the data structures is faithful to the specifications.

To these three attributes we should like to add a fourth advantage we feel applies particularly strongly to our specifications: they are easy to read and comprehend, thus facilitating informal verification of the fact that they do indeed conform to the intent of their creator.

2. A Short Example

Consider type Bag (of Integers) with the operations:

EMPTY_BAG:	$\rightarrow$ Bag
INSERT:	Bag $\times$ Integer $\rightarrow$ Bag
DELETE:	Bag $\times$ Integer $\rightarrow$ Bag
MEMBER_OF?:	Bag $\times$ Integer $\rightarrow$ Boolean

There are, of course, many ways to implement a bag. Some (e.g., a linked list representation) imply an ordering of the elements, some don't (e.g., a hash table implementation). These details are not relevant to the basic notion of a bag. A bag is nothing more than a counted set. A good axiomatic definition of type Bag must, therefore, assert that and only that characteristic. The axioms below comprise just such a definition. The axioms should prove relatively easy to read. (" = " has its standard meaning; s, i and i' are typed free variables, and ? = ? is an equality relation defined on type Integer.)

```
1) MEMBER_OF? (EMPTY_BAG, i) = false
2) MEMBER_OF? (INSERT(b, i), i') = if ? = ?(i, i')
                                   then true
                                   else MEMBER_OF?(b, i')
3) DELETE (EMPTY_BAG, i) = EMPTY_BAG
4) DELETE (INSERT(b, i), i') = if ? = ?(i, i')
                               then b
                               else INSERT(DELETE(b, i'), i)
```

As an interesting comparison, consider the following specification of type Set:

EMPTY_SET:	$\rightarrow$ Set
INSERT:	Set × Integer $\rightarrow$ Set
DELETE:	Set × Integer $\rightarrow$ Set
MEMBER_OF?:	Set × Integer $\rightarrow$ Boolean

```
5) MEMBER_OF? (EMPTY_SET, i) = false
6) MEMBER_OF? (INSERT(s, i), i') = if ? = ?(i, i')
                                   then true
                                   else MEMBER_OF?(s, i')
7) DELETE (EMPTY_SET, i) = EMPTY_SET
8) DELETE (INSERT(s, i), i') = if ? = ?(i, i')
                               then DELETE(s, i)
                               else INSERT(DELETE(s, i'), i)
```

Except for the difference in the **then** clauses of axioms 4 and 8, this specification is, for all intents and purposes, the same as that for type Bag. The specifications thus serve to point out the similarities and isolate the one crucial difference between these two types.

With some practice, one can become quite adept at reading algebraic axiomatizations. Practive also enhances one's ability to construct such specifications, but it doesn't make it trivial. The major difficulty lies in deciding how to attack the problem. Fortunately, our experience in writing algebraic specifications has led us to develop some heuristics that have proven extremely valuable in attacking this problem. A discussion of these heuristics constitutes a major portion of this paper.

Once one has constructed a specification, one must address the question of whether or not one has supplied a sufficient number of consistent axioms. The partial semantics of the type is supplied by a set of individual statements of fact. If any two of these is contradictory, the axiomatization is inconsistent. If, for example, one were to add the axiom:

```
MEMBER_OF?(DELETE(b, i), i') = if ? = ?(i, i')
                                then false
                                else MEMBER_OF?(b, i')
```

to the specification of type Bag, one would have created an inconsistent specification. There would exist values of type Bag for which it would be possible to prove both MEMBER_OF?(b, i) = true and MEMBER_OF?(b, i) = false, depending upon which axioms one chose to use.

MEMBER_OF?(DELETE(INSERT(INSERT(EMPTY_BAG, 3), 3), 3), 3)

is an example of an expression for which such a contradiction could be derived.

If the combination of axioms does not convey all of the vital information regarding the meaning of the operations of the type, the axiomatization is not sufficiently-complete. If the specification of an abstract type is not sufficiently-complete, it is not always possible to predict the behaviour of the programs that use the operations of the type. Experience with our methodology 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., DELETE(EMPTY_BAG, i) are particularly likely to be overlooked. The heuristics mentioned above have been devised to ameliorate this problem. We have also designed a system that can mechanically verify the sufficient-completeness of certain specifications. This system will be discussed later in this paper. We shall also discuss another approach to algebraic specification in which consistency seems to be the primary problem.

3. A Formal Look at Abstract Data Types and Their Algebraic Specification

Sections 1 and 2 presented informally a technique for the specification of abstract data types. This section takes a more formal look at that technique. This formalization is a necessary prelude to both a discussion of the inherent power of the specification technique and the construction of conditions sufficient to insure sufficient-completeness.

3.1. A Definition of Abstract Type

An abstract type is basically a collection of values and operations. For this reason it is quite natural to view it as an algebraic system. A type Natural Number, for example, might be defined as the values 0, 1, 2 ... and such operations as

SUCC:	Natural Number $\rightarrow$ Natural Number
PSUB:	Natural Number $\times$ Natural Number $\rightarrow$ Natural Number
EQUAL?:	Natural Number $\times$ Natural Number $\rightarrow$ Boolean
ABS:	Integer $\rightarrow$ Natural Number

Note that though we have defined the values of the type to be exactly the set $\{0,1,2\ldots\}$, both the domains and ranges of the operations of the type may include values from outside that set. For this reason, the language of conventional homogeneous algebras is not well-suited to a discussion of abstract data types. The language of heterogeneous algebras, on the other hand, is quite well-suited to the task.

A homogeneous algebra A, is a pair $[C, F]$, where C, the carrier, is a non-empty set of values, and F is a finite set of finitary operations, $Fj \cdot n$. Each $Fj \cdot n$ is a mapping:

$$Fj \cdot n\colon\ C^n \rightarrow C$$

Birkhoff and Lipson [1] generalized this to a heterogeneous algebra $[V, F]$, where V is a set of non-empty sets Vi, called phyla, and F is a finite set of finitary mappings $Fj \cdot n$:

$$Fj \cdot n\colon\ Vi1 \times Vi2 \times \cdots \times Vin \rightarrow Vk, \quad \text{where } \forall\, 1 \leqq h \leqq n[Vih \in V] \text{ and } Vk \in V.$$

This can be restricted to a *type algebra* $[V, F]$ where V is a set of phyla Vi, $1 \leqq i \leqq m$, which includes a distinguished phylum called TOI (type of interest), and F is a finite set of finitary mappings:

$$Fj \cdot n\colon\ Vi1 \times Vi2 \times \cdots \times Vin \rightarrow Vk \text{ where } \forall\, 1 \leqq h \leqq n[Vih \in V],$$
$$Vk \in V, \text{ and at least one member of the set}$$
$$\{Vi1, \ldots, Vin, Vk\} \text{ is the distinguished phylum TOI.}$$

Thus, a type Natural Number may be defined as the algebra $[V, F]$ where $V = \{\{true,\ false\},\ \{\ldots -2, -1, 0, 1, 2\ldots\},\ \{0, 1, 2\ldots\}\}$ and $F = \{\text{SUCC, PSUB, EQUAL?, ABS}\}$, where the domains and ranges of the mappings are as above. The crucial difference between a general heterogeneous algebra and a type algebra is that the latter defines one phylum (TOI) in terms of the others, whereas the former defines a set of phyla by mutual recursion. The specification technique presented in this paper presupposes the existence of only one type: Boolean. (The existence of this type, or at least of some type with two distinct values, is necessary to define the meaning of axioms that contain conditionals.) In applying the specification technique, however, it is frequently useful to presuppose the existence of other types. The specification of type Bag in Section 2, for example, assumes the existence of an independently-defined type Integer. Theoretically, it would be possible to define several types at once via mutual recursion. It seems, however, that in general a clean separation of types leads to clearer specifications.

It is hardly surprising that heterogeneous algebras are well suited to describing an algebraic system containing such common (in mathematics) domains as natural numbers, integers, and Booleans. Let us, therefore, consider a somewhat less "mathematical" example: type Queue (of Integers), with the operations:

NEW:	$\to$Queue
ADD:	Queue $\times$ Integer$\to$Queue
FRONT:	Queue$\to$Integer
REMOVE:	Queue$\to$Queue
?EMPTY?:	Queue$\to$Boolean

F, of course, is the set{NEW, ADD, FRONT, REMOVE, ?EMPTY?}. It is clear that V must contain $\{\ldots -2, -1, 0, 1, 2 \ldots\}$ and {true, false} and that $m=3$. It is not immediately clear how to represet the set TOI. This is not because of any intrinsic difference in the algebraic structure of queues and natural numbers, but is rather a function of the fact that a familiar notation for denoting values of type Queue is not available.

This is a common problem in specifying algebras, and is easily circumvented by defining the algebra in terms of a generator, rather than a carrier, set. In axiomatic set theory, for example, it is quite normal to define all sets in terms of the single generator { }, the empty set. Consider $Cl(G,F)$ the algebraic closure of F over the set G, as defined by the construction:

1) $x \in G$ implies $x \in Cl(G,F)$.

2) $Fj \cdot n \in F$ and $x1, x2, \ldots, xn \in Cl(G)$, implies that $Fj \cdot n(x1, x2, \ldots, xn) \in Cl(G,F)$.

3) These are the only members of $Cl(G,F)$.

Given homogeneous algebras $A=[C,F]$ and $A'=[Cl(G,F),F]$, if A' is the least subalgebra of A that includes G, and $A'=A$, then G is a generator set for A. The situation with respect to heterogeneous algebras is analogous. One has a family, Gi, of generators, and a family, Ci, of carriers such that:

1) $x \in Gi$ implies that $x \in Ci$

2) $Fj \cdot n$: $Vi1 \times Vi2 \times \cdots \times Vin \to Vk$ contained in F, and $x1, x2, \ldots, xn$ contained in $Ci1 \times Ci2 \times \cdots \times Cin$ implies that $Fj \cdot n(x1, x2, \ldots, xn)$ is contained in Ck.

3) These are the only members of the carriers Ci.

Now, returning to the Queue (of Integers) example, G may be defined as $\{\{\ldots -2, -1, 0, 1, 2 \ldots\}$, {true, false}, { }}, or, more succinctly, {Z, Boolean, { }}. There are two crucial things to note about this definition. First, for the phyla representing integer and Boolean values, the generator and carrier sets are identical. This is necessary because, given an interpretation that doesn't stray too far from what one might expect, the operations of the algebra will not generate new values of any phylum other than TOI. This is something we want to be true of type algebras in general, and it is what distinguishes TOI from the other phyla. The second thing to note about V is that the generator set for TOI is empty. This is a consequence of having chosen to treat the constant value NEW as a nullary "operation," rather than as "value" (a convention we shall continue to follow). Thus one of the operations serves, in effect, as the generator for the phylum. It is essential that there be at least one operator,

$$fn\colon\ Vi1 \times Vi2 \times \cdots \times Vin \rightarrow \mathrm{TOI},$$

where each Vi is contained in V-{TOI}, for in a heterogeneous algebra, each carrier phylum must be non-empty. We shall follow the convention of treating such nullary (with respect to the type of interest) functions as operations throughout this development.

3.2. The Semantics of Abstract Types

The discussion above furnishes a method for the presentation of the components that form an abstract type. To say that type Queue (of Integers) is an algebra,

$$[Cl(\{Z, \text{Boolean}, \{\ \}\}), \{\text{NEW, ADD, FRONT, REMOVE, ?EMPTY?}\}]$$

however, is not enough. Such a specification gives no more information about the type than do the kind of domain and range specifications that such languages as SIMULA 67 permit (see Section 1). The operations of the type still have to be given meanings.

This could be done by supplying an interpretation for each operation. To adopt this approach, however, is to ignore the primary reason for viewing a data type as an algebra, i.e., to provide a representation-free specification of the type. For this reason we have chosen to define the semantics of the operations of a type by supplying a set of axioms stating various properties that the algebra must possess. This is, of course, a conventional algebraic approach. The relation "$=$", for example, is most often defined merely by stating that it is reflexive, transitive, symmetric, and $a=b$ implies $f(a)=f(b)$. When defining familiar algebraic structures, e.g., the natural numbers, providing an axiomatization often presents no real problem. This is not, in general, the case for abstract data types.

The selection of appropriate axioms depends largely upon the type being defined. It is, therefore, impossible to give any general procedure for constructing axiomatizations for type algebras. It is, however, possible to characterize the sort of information that must be conveyed by any axiomatization. That is to say, it is possible to give a generalization of the form that such an axiomatization should take.

A slightly different notation for abstract types will prove convenient in this discussion. Since the generator set of TOI is by convention always empty, it need not be explicitly included in the specification of the algebra. The set of phyla, V, is thus replaced by the set $I = V - \{\text{TOI}\}$. Also, for reasons which will become clear later in this discussion, the set of operations, F, is partitioned into disjoint sets S and O such that S contains exactly those operations whose range is TOI. Intuitively, S contains the operations that can be used to generate values of the type being defined, and O the operations that map values of the type into other types. The need for operations to generate values of the type is clear, thus S will always be non-empty.

In principle, one could define a type for which O is empty. Such a type, however, would be singularly uninteresting. With no way to partition the set TOI (O empty implies no predicates) or to relate members of the set to members of other phyla, no member of TOI could be distinguished from any other member. That is to say, for all one could observe, every value of the type would be equivalent to every other value of the type. For all intents and purposes, there would be only one value of that type. The ability to distinguish among the values contained in TOI thus rests solely upon the effect that these values have when they appear in the argument lists of the operations contained in O.

A type may now be characterized as an algebra, $T=[Cl(I), S+O]$, and, in light of the above discussion, it is apparent that to define the semantics of the operations contained in O is to define the semantics of the type. Recall that all functions $Fj\cdot n$ in O are of the form

$$Fj\cdot n\colon\quad Vi1\times Vi2\times\cdots\times Vin\rightarrow Vk \text{ where for } 1\leqq h\leqq n$$
$$Vih\in I+\{\text{TOI}\} \text{ and } Vk\in I.$$

Thus, the semantics of the type may be defined as a set of mappings

$$Me\colon\quad O\times(I+\{TOI\})^p\rightarrow\{\text{error}\}+Vi\in I$$

where p is the maximum arity of the functions in O, and for all $f\in 0$ and $w\in(I+\{\text{TOI}\})^p$, $Me(f,w)=\text{error}$ (a distinguished error element) if w is not contained in the domain of f; and $Me(f,w)=y$ where $y\in Vi\in I$ if w is in the domain of f.

An example may help to clarify matters. Consider type Queue (of Integers) $=[Cl(\{Z, \text{Boolean}\}), \{\text{NEW, ADD, REMOVE}\}+\{\text{FRONT, ?EMPTY?}\}]$. To supply the semantics of this type one need only define the function

$$Me\colon \{\text{FRONT, ?EMPTY?}\}\times\{Z, \text{Boolean, Queue}\}\rightarrow\{Z, \text{Boolean}\}$$

The problem is how to present an axiomatization of such a function. The definition of Me implies one axiom that must be present in the axiomatization of any type:

$$\forall\, Fj\cdot n, x1,\ldots,xn\,[(x1,\ldots,xn) \text{ not in the domain of } Fj\cdot n \text{ implies that}$$
$$Me(Fj\cdot n, x1,\ldots,xn)=\text{error}]$$

Note that in an algebraic specification of an abstract type the domain of each $Fj\cdot n$ is formally supplied by the syntactic specification. Note also that if none of the domains of the operations of the type contains "error," and this will normally be the case, this axiom implies that

$$\forall\, Fj\cdot n, x1,\ldots,xn\,[xi=\text{error},\ 1\leqq i\leqq n, \text{ implies that}$$
$$Me(Fj\cdot n, x1,\ldots,xn)=\text{error}].$$

That is to say, the distinguished value "error" has the property that the value of any operation applied to an argument list containing "error," is "error". Since this axiom is a part of every type specification, it will be included implicitly in all subsequent specifications, and will be referred to as the *implicit axiom.* The question now is: how should such an axiomatization be completed?

3.3. *The Axiomatization of Type Algebras*

The first point to address, is what it means to "complete" an axiomatization. The notion of a complete axiom set is a familiar one to logicians. The form of definition used depends on the environment in which one is working. The statements that a complete axiom set is "one to which an independent axiom cannot be added," or "one with which every well-formed formula or its negation can be proved as a theorem," or "one for which all models are isomorphic (i.e., the axioms are categorical)," are all common. Our notion of completeness conforms to none of these statements. We have, therefore, introduced the qualifier "sufficiently" to try to differentiate our notion of completeness from these other, more common, notions. Above, sufficiently-complete was defined informally by saying that all one had to do was provide a semantics for the function *Me*. We now define it more rigorously.

The sets I and $S+O$ of the algebra for a type T may be used to define a word algebra [2]. The set of words, $L(T)$, contained in this algebra are defined inductively as follows:

1) All elements of $Vi \in I$ are contained in $L(T)$.

2) If $Fj \cdot n$ is contained in $S+O$, and $(x1, \ldots, xn)$ is contained in the domain of $Fj \cdot n$ and each xi, $1 \leqq i \leqq n$, is contained in $L(T)$, then $Fj \cdot n(x1, \ldots, xn)$ is contained in $L(T)$.

3) These are the only terms contained in $L(T)$.

Intuitively, $L(T)$ is the language defined by the abstract type. It is the union of all the carrier phyla of the type, and specifies a set of terms that may appear in the body of any program in which the type has been defined. For an axiomatization of a type to be sufficiently-complete, it must assign meaning to each of the terms in this language.

At first glance, this may seem to be a formidable task. Fortunately, however, one need explicitly consider only a subset of $L(T)$. Recall that the actual goal is to define

$$Me\colon\ O \times (I + \{\mathrm{TOI}\})^p \rightarrow Vi \in I$$

Thus, one need consider only those words contained in $L(T)$ whose outermost operation is contained in O. With this in mind, the term sufficiently-complete can now be precisely defined:

Definition. For any abstract type $T = [Cl(I), S+O]$, and any axiom set A, A is a *sufficiently-complete axiomatization* of T if and only if for every word of the form $Fj \cdot n(x1, \ldots, xn)$ contained in $L(T)$ where $Fj \cdot n \in O$, there exists a theorem derivable from A of the form $Fj \cdot n(x1, \ldots, xn) = u$, where $u \in Vi \in I$. A is *consistent* if for each $Fj \cdot n(x1, \ldots, xn)\, u$ is unique.

It can be shown (see Sect. 3.5) that the problem of establishing whether or not a set of axioms is sufficiently-complete is, in general, undecidable. Thus, one cannot be provided with necessary and sufficient guidelines to the construction of sufficiently-complete axiomatization of type algebras. If, however, one is willing to accept limitations on the kinds of algebras that may be defined and on the language used to specify the axioms, it is possible to state conditions that will be sufficient to ensure sufficient-completeness. A more complete discussion of this is contained in the next three sections.

3.4. A Schema for Presenting Axiomatizations

An axiom set, A, for specifying an abstract type, T, will consist of the implicit axiom mentioned in Section 3.2 plus a finite set of axioms, each of which is of the form:

$$\forall\, x1, \ldots, xn\, [lhs = rhs]$$

$\forall$ and $=$ have their usual meanings. $\forall$ is the universal quantifier "for all." $=$ is reflexive, transitive, symmetric, and $P \wedge x = y$ implies $P(x/y)$. ($P(x/y)$ means P with y substituted for all free occurrences of x.)

For an axiomatization of a type, T, the number of possible axioms will be constrained by limiting the possible left hand sides to a finite set, LHS. This bounding of the left hand sides is based upon the assumption (discussed in Sect. 3.2.) that the significance of the values of TOI is embedded solely in the effect that these values have when they appear in the argument lists of operations contained in O. Thus, any axiom set in which all such effects were defined would be sufficiently-complete. This train of thought led to the belief that one could limit the set LHS to those terms generated by assigning free variables to the arguments of each $s \in S$ and permuting these through the appropriate positions in the argument list of each $o \in O$. Though a set of left hand sides derived via this algorithm should, in all cases, prove sufficient, it may often also prove to be inconvenient. It is sometimes convenient to generate left hand sides in which operations contained in S occur at the outermost level (which will be discussed in Section 3.6). The level of nesting in the left hand side, however, may still be limited to two. The set LHS is defined formally as:

$$\begin{aligned} \text{LHS} = \{Fj \cdot n(v1, \ldots, vn) \mid Fj \cdot n \in S + O,\ \forall\, i \leqq n\, [&vi \text{ is a free} \\ &\text{variable or } vi = Fk \cdot m(x1, \ldots, xm) \\ &\text{and } Fk \cdot m \in S + O \text{ and } \forall\, 1 \leqq l \leqq m\, xl \text{ is} \\ &\text{a free variable}]\} \end{aligned}$$

The definition of the set RHS, in which rhs must be contained, is somewhat more complex. Informally, a right hand side may be any valid expression all of whose free variables are free in the corresponding left hand side. More formally, the set of potential right hand sides for the *i*th axiom, RHSi, is defined inductively as:

1) If xj is a constant or appears as a free variable in LHSi, then xj is contained in RHSi.

2) If $r1 \ldots rm$ are contained in RHSi and $Fj \cdot m$ is an operation (not necessarily contained in $S+O$, i.e., it may be an operation from some Vi other than the TOI) whose range is any of the phyla in V, then $Fj \cdot m(rl \ldots rm)$ is contained in RHSi.

3) If b, y, z are contained in RHSi and b is of the form $Fj \cdot m(r1, \ldots, rm)$ where the range of $Fj \cdot m$ is Boolean then **if** b **then** y **else** z is contained in RHSi.

4) These are the only members of RHSi.

The meaning of the **if then else** construct is, as one might expect, **if** b **then** y **else** $z=[y$ if b is true, z if b is false, and undefined otherwise$]$. It is interesting to note that it is not essential to include the conditional expression as a primitive. Given any two distinct values, call them TRUE and FALSE, we can define an abstract type Boolean with an IFTHENELSE operation defined by the axion:

$$\text{IFTHENELSE}(\text{TRUE}, x, y) = x$$
$$\text{IFTHENELSE}(\text{FALSE}, x, y) = y$$

Note that the mapping Me, which the axioms purport to define, does not appear explicitly in the axiomatization of a type. Nevertheless, the axioms can be used to define Me. Consider, as an example, a type Posint (positive integer):

$$T = [Cl(\{\text{Boolean}\}), \{\text{ONE, SUCC, ADD}\} + \{\text{?ONE?}\}]$$

where the domains and ranges of the operations are:

ONE:	$\rightarrow$ Posint
SUCC:	Posint $\rightarrow$ Posint
ADD:	Posint $\times$ Posint $\rightarrow$ Posint
?ONE?:	Posint $\rightarrow$ Boolean

A suitable axiomatization might include the implicit axiom plus:

1) $\forall x1 \in \text{Posint}, x2 \in \text{Posint} [\text{ADD}(x1, x2) = \text{ADD}(x2, x1)]$
2) $\forall x \in \text{Posint} [\text{ADD}(\text{ONE}, x) = \text{SUCC}(x)$
3) $\forall x1 \in \text{Posint}, x2 \in \text{Posint} [\text{ADD}(\text{SUCC}(x1), x2) = \text{SUCC}(\text{ADD}(x1, x2))]$
4) $\text{?ONE?}(\text{ONE}) = \text{true}$
5) $\forall x \in \text{Posint} [\text{?ONE?}(\text{SUCC}(x)) = \text{false}]$

For this type, $O = \{\text{?ONE?}\}$ and $I = \text{Boolean}$, thus the domain of Me is $[\{\text{?ONE?}\} \times \text{Posint}]$ and the range Boolean. A simple induction on the length of words can be used to show that for all $x \in \text{Posint}$, axioms 1–3 can be used to reduce x to ONE or SUCC* (ONE). (Here, the $*$ is the "Kleene star.") Thus, for any $x \in$Posint, ?ONE?(x) can be reduced to either true or false. Thus the mapping

$$Me\colon \{?\mathrm{ONE}?\} \times \mathrm{Postint} \rightarrow \mathrm{Boolean}$$

is fully defined.

As a notational convenience, because all free varibles are universally quantified over the correct type, the quantifiers that start each axiom may be dropped. Hence the notation used in Section 2.

3.5. The Power of the Schema and the Decidability of Sufficiently-Complete

Having restricted the manner in which type definitions may be supplied, it becomes necessary to address the question of whether or not the schema provided is sufficiently "powerful" to specify any type that one might wish to define. That is to say, does the class of algebras that can be defined via the provided mechanisms include all of those algebras that might prove useful to a programmer? The answer is "almost." While we cannot always specify exactly the intended algebra, we can specify a containing algebra. That is to say, we can always specify an algebra that, by the forgetting of some operations, can be restricted to the intended algebra. The necessity of allowing forgetting, i.e., the use of hidden functions, is discussed in [16] and proved in [24].

Theorem 1. Any algebraic system $[Cl(I), S+O]$ such that all the phyla of I are recursively enumerable sets and all operations contained in $S+O$ are partial computable functions, is contained in an algebra that may be axiomatically defined using only the primitives (as defined above) $\forall$ (implicitly), $=$, and **if then else**.

Proof. The proof of this theorem follows directly from Kleene's proof that every (general) recursive function can be expressed equationally [14]. See [11] for a comprehensive discussion of this and closely related issues.

Thus, the schema provided for type specifications will, in all instances, prove to be sufficiently powerful – probably too powerful. While it is true that on rare occasions one may wish to write a non-terminating program (e.g., an operating system), in most cases non-termination is indicative of an error. In particular, it is hard to imagine a useful type, $T = [Cl(I), S+O]$, where O contains a potentially non-terminating operation. It therefore seems useful to devise restrictions on the schema for building axioms that will ensure that only total functions are specified as members of O. This is not to say that any input to an operation must be deemed acceptable, e.g., NEW as an argument to FRONT of type Queue, but merely that the operation should never fail to terminate. In the case of an unacceptable input it could, for example, return the distinguished value "error." It is, of course, impossible to derive a set of necessary and sufficient restrictions. To do so would be to solve the halting problem. So we investigate sufficient conditions.

Theorem 2. Any conditions that are sufficient to ensure sufficient-completeness will ensure that all operations contained in O are total.

Proof of Theorem 2.

1) If $(x1,\dots,xn)$ is not contained in the domain of Fn, then, by the implicit axiom, $Fn(x1,\dots,xn)=\text{error}$. Thus $\forall Fn \in S+O$, $x1,\dots,xn$ $[Fn(x1,\dots,xn)$ contained in $L(T)$ implies that $Fn(x1,\dots,xn)$ converges to a value$]$.

2) If $Fn(x1,\dots,xn) \in L(T)$ and $Fn \in O$, then, by the definition of sufficiently-complete, there exists a $u \in ((Vi \in I)+\{\text{error}\})$ such that $F(x1,\dots,xn)=u$. Thus for all $Fn(x1,\dots,xn) \in L(T)$. $Fn \in O$ implies that $Fn(x1,\dots,xn)$ has a definite value. q.e.d.

It is important that sufficiently-complete implies that all operations contained in O are total. From the user's point of view, it means that if his axiomatization is sufficiently-complete, he not only learns something about the axiomatization, but he also leans (or is reassured about) something about the operations that have been axiomatized. More importantly, Theorem 2 also serves to highlight the difficulty involved in checking the sufficient-completeness of an axiomatization.

Ideally, one would like to be able to construct a total function

$$R\colon \text{ Axiom set} \times \text{Syntactic specifications} \rightarrow \text{Boolean}$$

such that $R(A, ss)=\text{true}$ if A is a sufficiently-complete axiomatization of ss, and false otherwise. However, since, as implied above, such a function could be used to solve the halting problem, it is clear that no such function exists. Thus, one is led to consider a slightly less informative function, and a somewhat different notion of completeness.

Definition. Given a total function R such that $R(A, ss)=\text{true}$ implies that A is a sufficiently-complete axiomatization of ss, A is *recognizably sufficiently-complete* with respect to R (r.s.c. (R)), if and only if $R(A, ss)=\text{true}$.

Unfortunately, this problem cannot be solved completely satisfactorily either.

Theorem 3. There does not exist a function R for determining sufficient-completeness such that for all types $T=[Cl(I), S+O]$, where $F \in O$ implies that F is total, there exist r.s.c. (R) axiomatizations.

About the Proof. It is shown that if the theorem were not true, the set of all total functions would be recursively enumerable. Since this is known to not be the case, the theorem must be true.

Proof of Theorem 3.

1) Assume that Theorem 3 is not true. I.e., that there exists a function R1 such that for all $T=[Cl(I), S+0]$, if all $F \in O$ are total, then there exists a sufficiently-complete axiomatization, A, of T such that $R1(A, ss(T))=\text{true}$, and further for all axiomatizations A', A' not sufficiently-complete implies that $R1(A', ss(T))=\text{false}$.

2) Theorem 2 has an obvious corollary: that for any function R, any type $T = [Cl(I), S+O]$, and any axiom set A; if A is a r.s.c.(R) axiomatization of the syntactic specification of T $(ss(T))$, then all operations contained in O are total.

3) The set of all function names is known to be recursively enumerable (r.e).

4) Therefore, the set of all finite sets of function names is r.e.

5) Thus the set of all syntactic specifications is r.e.

6) For any syntactic specification, the set of axiomatizations (according to the schema we have presented) is r.e.

7) From 5 and 6 the set of all pairs $(A, ss(T))$ is r.e.

8) From 7 and 1, $B = \{T = [Cl(I), S+O] |$ there exists an r.s.c.$(R1)$ axiomatization of $T\}$ is r.e.

9) Thus the set $BO = \{o |$there exists $T = [Cl(I), S+O]$ and $T \in B$ and $o \in O\}$ is r.e.

10) Thus the set $FO = \{F | F \in BO\}$ is r.e.

11) From 2, $F \in FO$ implies that F is total.

12) From 1, all total functions are contained in FO.

13) 11 and 12 imply that FO is the set of all total functions, 10 that this set is r.e.

14) But this is known not to be the case. Hence the assumption of 1 must be false. q.e.d.

The ramifications of this are not so unfortunate as they may at first appear to be. In programming, the occasions when one has need of a non-primitive recursive function seem to be very rare indeed. It thus seems that if one were to construct a decision procedure, $R2$, such that for all types where all operations are primitive recursive there exist r.s.c. $(R2)$ axiomatizations, one would have a procedure that would be useful for a wide range of applications. A set of restrictions on the schema for supplying axiomatizations of types around which such a decision procedure can be built is presented in the next section.

3.6. Sufficient Conditions for Establishing Sufficient-Completeness

In the introduction we argued that one of the main difficulties in supplying axiomatic specifications of abstract data types is knowing when one has "fully specified" the type. We also suggested that this problem could be ameliorated by the use of a system that could check the completeness of an axiomatization and suggest ways of completing an incomplete specification. Section 3.3, however, demonstrated that in the general case the sufficient-completeness problem is unsolvable, i.e., there cannot exist a decision procedure for recognizing sufficient-completeness. Section 3.5 proved an even stronger result: that there does not exist a semi-decision procedure, R, such that for all type algebras there exists an axiomatization that is recognizably sufficiently-complete with respect to R. Thus, the scope of any system such as that suggested in the introduction must necessarily be somewhat limited.

In this section, we present a set of conditions that are sufficient to guarantee that an axiom set is sufficiently-complete. We also prove that for any type

$[Cl(I), S+O]$ if all members of $S+O$ are primitive recursive, there exists an axiomatization of that type that fulfills these conditions. These conditions can thus serve as the basis of a semi-decision procedure that can be used to show the sufficient-completeness of a large number of axiomatizations.

Convention. For notational convenience, it will be assumed that for all operations contained in $S+O$, all arguments contained in TOI precede all arguments contained in other phyla in the operation's domain specification. This implies that each operation is of the form $f(y^*)$ or $f(x, y^*)$ (the y^* indicates a list, possibly empty, of arguments) where x is contained in TOI.

Definition. Consider an axiomatization, A, of a type $T=[Cl(I), S+O]$ and an arbitrary predicate P. A term of the form $f(x^*)$, where x^* is free, is *P-safe* if for all legal assignments to x^* there exists a theorem derivable from A of the form $f(x^*)=z$, and $P(z)$.

Definition. An axiomatization, A, is a *P-safe axiomatization* of a type $T=[CL(I), S+O]$ if $\forall f \in S+O$, $f(x^*)$ is P-safe.

Now, consider the predicate ATOMIC(y) which is true if y contains no operations contained in $S+O$ or y is of the form $s(y^*)$ where $s \in S$. It should be clear that the notation of *atomically-safe* is closely related to that of sufficiently-complete. It implies that all terms of the form $o(x^*)$, $o \in O$, can be reduced to terms that we assume are already well-defined. The atomic-safety problem, like the sufficient-completeness problem, is unsolvable in the general case. It is, however, possible to develop conditions that are sufficient to guarantee the atomic safety of all terms of the form $o(x, y^*)$.

Consider a simplified schema for the axiomatization of abstract data types: one in which the **if then else** has been eliminated. (It will be reintroduced later in this development.) Given this restriction, we derive some lemmas about our axiom systems in general.

Definition. For any computable measuring function M, M: free terms $\rightarrow$ Natural numbers, an equation lhs = rhs is *monotone in M* ($m(M)$) if and only if $M(\text{lhs}) > M(\text{rhs})$.

Lemma A. Consider any measuring function M, and any predicate P such that $(M(f(x^*))=0)$ implies $P(f(x^*))$, and any set of axioms, A. If for all terms of the form $f(x^*), f \in F$ we can derive a monotone theorem of the form $f(x^*)=z$, then A is P-safe. That is to say for all assignments to x^* there exists a theorem derivable from A of the form $f(x^*)=z$ where $P(z)=\text{true}$.

Lemma A establishes a monotone induction principle that we shall invoke several times in the course of this paper. Its most obvious (and first) application will be to the atomic safety problem, where the predicate P is represented by the predicate IS_ATOMICALLY_SAFE. Before proceeding to apply this principle, however, let us prove its soundness.

About the Proof. The lemma is shown to be true by induction on $M(f(x^*))$.

Proof of Lemma A.

1) From the statement of the lemma, for any ground term $f(x^*)$, $M(f(x^*))=0$ implies that there exists a theorem of the form $f(x^*)=z$ (i.e., $f(x^*)=f(x^*)$), such that $P(z)$. This forms the basis of our induction.

2) Assume that for any ground term (i.e., one with no variables of the TOI) $f(x^*)$ such that $M(f(x^*))\leqq n$ it is possible to derive a theorem of the form $f(x^*)=z$ where $P(z)$.

3) Consider any ground term $f(x^*)$ such that $M(f(x^*))=n+1$. The statement of the lemma assures us that it will be possible to derive a theorem of the form $f(x^*)=w$ where $M(w)<M(f(x^*))$. That is to say $M(w)\leqq n$. By the induction hypotheses of step 2, it is possible to derive a theorem of the form $w=z$ such that $P(z)$. q.e.d.

Now, let us consider a particular measuring function.

Definition. NEST(x) is equal to the greatest depth to which operations contained in $S+O$ are nested in the term x. NEST$(o(s(j)))$ where $o\in O$, $s\in S$, and $j\in Vi\in I$, for example, equals two.

Lemma B. A is an atomically safe axiomatization of $T=[Cl(I), S+O]$ if $\forall\, o\in O$ and $\forall\, s\in S$ there exists an axiom of the form $o(s(x,y^*),w^*)=z$, where either

I) the axiom is m(NEST)

II) there exist axioms $z=z1$, $z1=z2,\ldots,zn=zn+1$ and the theorem $z=zn+1$ is m(NEST).

Proof of Lemma B

1) Clearly, NEST$(x)=0$ implies that x is atomic, i.e., NEST$(x)=0$ implies ATOMIC(x). Therefore, NEST is a measuring function as per Lemma A. q.e.d.

Recall that the above discussion deals with axioms in which the right hand side contains no conditionals. This deficiency is remedied by Lemma C.

Definition. An axiom that meets either condition *I* or condition II as stated in Lemma B is said to be *Bsafe.*

Lemma C. A is an atomically safe axiomatization of $T=[Cl(I), S+O]$ if $\forall\, o\in O$ and $\forall\, s\in S$ there exists an axiom of the form $O(s(x, y^*), w^*)=z$ where either

I) z contains no conditionals and the axiom is *Bsafe.*

II) z is of the form **if** b **then** $z1$ **else** $z2$ where b is Boolean, $o(s(x, y^*), w^*)=z1$ and $o(s(x, y^*), w^*)=z2$ are *Bsafe*, and NEST$(b)<$NEST$(o(s(x, y^*), w^*))$ or the range of o is Boolean and $o(s(x, y^*), w^*)=b$ is *Bsafe.*

Proof of Lemma C.

1) The conditions placed on b guarantee that for any ground instance of b it is possible to derive a theorem of the form $b=$true or a theorem of the form $b=$false.

2) Thus, for any ground instance of $o(s(x, y^*), w^*)$ it will be possible to derive either $o(s(x, y^*), w^*)=z1$ or $o(s(x, y^*), w^*)=z2$.

3) The last stipulation in II above and Lemma B guarantee that these will be reducible to atomic terms. q.e.d.

Note that Lemma C still does not provide for nested conditionals. Theorem 4, which concludes our development of sufficient conditions to ensure safety, remedies this.

Definition. An axiom that meets either condition I or condition II of Lemma C is said to be *Csafe.*

Definition. An axiom that meets either condition I or condition II of Theorem 4 (below) is said to be *4safe.*

Theorem 4. A is a safe axiomatization of $T=[Cl(I), S+O]$ if $\forall o \in O$ and $\forall s \in S$ there exists an axiom of the form $o(s(x, y^*), w^*)=z$ where either

I) the axiom is *Csafe*, or

II) z is of the form **if** b **then** $z1$ **else** $z2$ where b is Boolean, $o(s(x, y^*), w^*) = z1$ and $o(s(x, y^*), w^*)=z2$ are *4safe*, and $\mathrm{NEST}(b) < \mathrm{NEST}(o(s(x, y^*), w^*))$ or the range of o is Boolean and $o(s(x, y^*), w^*)=b$ is *4safe.*

Proof of Theorem 4.

1) Consider an axiom $o(s(x, y^*), w^*)=z$. If there is no nesting of conditionals in z, then, by Lemma C, the theorem holds. This forms the basis of an induction.

2) Assume that the theorem holds if the level of nesting of conditionals in z is $\leqq n$.

3) Consider the case where conditionals are nested to a depth of $n+1$ in z.

4) Consider the outermost conditional. It must be of the form **if** b **then** $z1$ **else** $z2$. The conditions placed on b guarantee that for any ground instance of b it is possible to derive a theorem of the form $b=$true or $b=$false.

5) Thus, for any ground instance of $o(s(x, y^*), w^*)$ it will be possible to derive either $o(s(x, y^*), w^*)=z1$ or $o(s(x, y^*), w^*)=z2$.

6) The depth to which conditionals are nested in $z1$ or $z2$ must be $\leqq n$. Thus, by the induction hypothesis, the theorem holds. q.e.d.

As suggested earlier, this theorem leads directly to a set of conditions, $R2$, that are sufficient to ensure sufficient-completeness. To wit, if an axiomatization of an abstract data type meets the conditions outlined in Theorem 4, then the axiomatization is sufficiently-complete. At first glance, this may not seem to be a significant improvement over earlier formulations of sufficient-completeness. It does, however, have several advantages. Both the number of terms that have to be checked and the size of the axiom set A are finite. Thus, there is a reasonable bound on the time required to algorithmically determine whether or not a set of axioms is an r.s.c. ($R2$) axiomatization of any given type. The algorithm is a simple one:

1) Generate the set $\mathrm{STERMS}=\{s(x^*) \mid s \in S$ and all members of x^* are free variables$\}$.

2) Generate the set $\mathrm{CTERMS}=\{o(x^*) \mid o \in O;\ x1, \ldots, xm \in \mathrm{STERMS};\ xm+1, \ldots, xn$ are free variables; and o is an n-ary function whose first m arguments are of type TOI$\}$

```
3) Do for each $x \in$CTERMS
       notsafe: = true
     Do for each $a \in A$ while (notsafe)
       If (lhs of $a = x$) & ($a$ meets condition I or
            $a$ meets condition II of Theorem 4)
         then notsafe: = false
     end
     If notsafe
       then Return (not r.s.c. ($R2$))
   end
   Return (is r.s.c. ($R2$))
```

In addition to providing us with a procedure for checking sufficient-completeness, $R2$ provides us with some heuristics for constructing sufficiently-complete axiomatizations. Unfortunately, the obvious approach (i.e., beginning with the set CTERMS as the set of left hand sides for which right hand sides must be constructed) is often inconvenient. If some of the operations contained in O have several arguments of type TOI, CTERMS can be quite large. Consider, for example, type Natural Number $= [Cl(\{\text{Boolean}\})$. {ZERO, SUCC, PRED, ADD, MULTIPLY} + {?ZERO?, ?EQUAL?}]. There would be 30 left hand sides generated. By providing axioms that relate values in TOI to one another, e.g., ADD(SUCC(x), ZERO) = SUCC(x), however, the number of terms to be considered can be substantially reduced. Hence the notion of convertibility. Informally, if the values "generated" by an operation f, can always be expressed in terms of other operations, then f is said to be convertible. In type Natural Number, for example, all terms of the form PRED(x) are convertible to either ZERO, error, or SUCC(y): where y does not contain PRED. Convertible operations are essentially "functional extensions" which do not extend the set of values of TOI. More formally:

Definition. Given an axiom set A and a type $[Cl(I), S+O+\{f\}]$ where f: TOI $\times Vi1 \times \cdots \times Vin \to$ TOI is not contained in S, if for each ground term of the form $f(u, v^*)$ there exists a theorem derivable from A of the form $f(u, v^*) = z$ where z contains no instances of f, then f is said to be *convertible to S.*

Now, consider a type $T = [Cl(I), S+O]$ with a sufficiently-complete axiomatization A. Lemma D presents conditions that are sufficient to ensure that given an axiomatization $A + A'$ of type $T' = [Cl(I), S+O+\{f\}]$, f is convertible to S.

Definition. For any term x and any operation f, $Mf(x)$ is equal to the number of instances of f that appear in x.

Lemma D. Consider types $T = [Cl(I), S+O]$ with axiomatization A, and $T' = [Cl(I), S+O+\{f\}]$ with axiomatization $A+A'$. f is convertible to S if for each term of the form $f(s(x^*), y^*)$ where $s \in S$ and all members of x^* and y^* are free variables, there exists in A' an axiom of the form $f(s(x^*), y^*) = z$ where either

I) $Mf(z) = 0$, or

II) s is not a constant (i.e., it is not nullary with respect to TOI), and for every subterm, w, of z (including z itself), if w is of the form $f(u, v^*)$, then u is a free variable contained in x^* and all members of v^* are free variables contained in y^*.

About the Proof. It is shown by induction on depth of nesting (in steps 2–5) that for any ground term of the form $f(s(a^*), b^*)$, where $s \in S$, it is possible to derive a theorem of the form $f(s(a^*), b^*) = z$, where $Mf(f(s(a^*), b^*)) > Mf(z)$. Step 1 is then invoked to show that this is a sufficient condition to guarantee the convertibiltiy of f.

Proof of Lemma D.

1) It follows directly from Lemma A that if for each ground term of the form $f(s(a^*), b^*)$, where $s \in S$, it is possible to derive a theorem of the form $f(s(a^*), b^*) = z$, where $Mf(f(s(a^*), b^*)) > Mf(z)$, then f is convertible to S.

2) Consider any ground term $f(s(a^*), b^*)$ such that $\mathrm{NEST}(s(a^*)) = 1$. s must be a constant, thus case II of Lemma D is not applicable. Hence, there must exist an applicable axiom of the form $f(s(x^*), y^*) = z$ where $Mf(z) = 0$. Thus, we may directly derive a theorem of the form $f(s(a^*), b^*) = z1$ where $Mf(f(s(a^*), b^*)) > Mf(z1)$.

3) Assume that if $\mathrm{NEST}(s(a^*)) \leqq n$ then for any ground term of the form $f(s(a^*), b^*)$ it is possible to derive a theorem of the form $f(s(a^*), b^*) = z1$ where $Mf(f(s(a^*), b^*)) > Mf(z1)$.

4) Consider any ground term $f(s(a^*), b^*)$ such that $\mathrm{NEST}(s(a^*)) = n+1$. The lemma asserts that there must be an applicable axiom $f(s(x^*), y^*) = z$ such that either

a) $Mf(z) = 0$. In this case it is possible to directly derive a theorem of the form $f(s(a^*), b^*) = z1$ where $Mf(f(s(a^*), b^*)) > Mf(z)$.

b) It is possible to directly derive a theorem of the form $f(s(a^*), b^*) = z1$ where for all subterms of $z1$ of the form $f(u, v^*)$, u must be a member of a^*. $\mathrm{NEST}(u)$ must, therefore, be less than $\mathrm{NEST}(s(a^*))$, i.e., $\mathrm{NEST}(u) \leqq n$. Therefore, by the assumption of step 4, it is possible to derive a theorem of the form $f(u, v^*) = z1$ where $Mf(f(u, v^*)) > Mf(z1)$.

5) By induction, it is therefore possible to derive, for any ground term of the form $f(s(a^*), b^*)$, a theorem of the form $f(s(a^*), b^*) = z$ where $Mf(f(s(a^*), b^*)) > Mf(z)$.

6) Therefore, by step 1, f is convertible to S. q.e.d.

Like Lemma A, Lemma D makes no provision for conditionals. This is rectified in Theorem 5.

Definition. An axiom that meets either condition I or condition II of Lemma D is said to be *Dok*.

Theorem 5. All free terms of the form $f(x, y^*)$ are convertible to S if $\forall s \in S$ there exists a *5-ok* axiom, i.e., an axiom of the form $f(s(x^*), y^*) = z$ where either

I) z contains no conditionals and the axiom is *Dok*.

II) z is of the form **if** b **then** $z1$ **else** $z2$ where b is Boolean and atomic, or Boolean and *4safe*, and $f(s(x^*), y^*) = z1$ and $f(s(x^*), y^*) = z2$ are 5-*ok*.

Proof of Theorem 5.

1) The conditions placed on b guarantee that for any ground instance of $f(s(x^*), y^*)$ it is possible to derive a theorem of the form $b = \text{true}$ or one of the form $b = \text{false}$.

2) Thus, for any ground instance of $f(s(x^*), y^*)$ it will be possible to (eventually) derive a Dok theorem of the form $f(s(x^*), y^*) = z$.

3) Lemma D asserts that this is a sufficient condition to guarantee that all terms of the form $f(s(x^*), y^*)$ are convertible. q.e.d.

The concept of convertibility combined with that of safety leads to a second set of conditions, $R3$, that is sufficient to guarantee sufficient-completeness. Informally, if every term of the form $o(x, y^*)$, where $o \in O$, is either safe or convertible to something that is safe, then the axiomatization is sufficiently-complete. More formally:

Theorem 6. An axiom set, A, for a type, $T = [Cl(I), S + O]$, is sufficiently-complete if there exists a partitioning of S into disjoint sets C (constructors) and E (extensions) such that all constants are contained in C and

1) For all $c \in C$ and all $o \in O$, the term $o(c(x^*), y^*)$, where the members of x^* and y^* are free, is *4safe*, and

2) There exists an ordering, $e1, e2, \ldots, em$, of the functions in E, such that $e1$ is convertible to C, $e2$ is convertible to $C + \{e1\}$, etc.

About the Proof. It is first shown (steps 1–3) that it is sufficient to prove that for all ground terms of the form $o(c(x^*), y^*)$, $o \in O$ and $c \in C$, there exists a theorem of the form $o(c(x^*), y^*) = z$, where z is atomic or safe. That this is true follows immediately from the definition of safe and the initial conditions specified in the theorem.

Proof of Theorem 6.

1) Recall that an axiom set, A, is sufficiently-complete if and only if for each $w = Fj \cdot n(x1, \ldots, xn) \in L(T)$ where $Fj \cdot n \in O$, there exists a theorem derivable from A such that the theorem is of the form $w = u$ where u is contained in $Vi \in I$. Recall also that $u \in Vi \in I$ implies that u is atomic with respect to TOI.

2) $Fj \cdot n \in O$ implies that at least one of $x1, \ldots, xn$ is contained in TOI, thus (by our convention) $x1 \in \text{TOI}$. $x1 \in \text{TOI}$ implies that $x1$ is either a constant or of the form $s(u, v^*)$, $s \in S$.

3) Because all terms of the form $e(x, y^*)$ where $e \in E$, are convertible, we need only consider those terms w where $x1$ is a constant or of the form $c(u, v^*)$ and $c \in C$.

4) Thus we need only show that for any assignment to the free variables of all terms of the form $o(c(x^*), y^*)$ where $o \in O$ and $c \in C$, there exists a theorem of the form $o(c(x^*), y^*) = z$, where z is atomic with respect to TOI.

5) That is to say all free terms of the form $o(c(x^*), y^*)$ must be safe. This is guaranteed by the first condition of the theorem. q.e.d.

Thus, the notions of safety and convertibility lead to a set of purely syntactic conditions that can be used to verify sufficient-completeness. They can, therefore, be used to build a procedure, $R3$,

$R3$: Axiom set $\times$ Syntactic specification $\rightarrow$ Boolean

such that $R3(A, ss(T))=$true only if A is a sufficiently-complete axiomatization of T. Note, however, that Theorem 6 deals only with the soundness of such an $R3$. To know that $R3$ is sound is not enough. $R4=[F(A, ss(T))=$false$]$ is, after all, a perfectly sound procedure in that it never returns true if A is not a sufficiently-complete axiomatization of T. A useful procedure, R, for verifying sufficient-completeness must not only be guaranteed against returning spurious values, but also must be capable of recognizing a large class of sufficiently-complete axiomatizations. Almost all of the algebraic specifications of abstract types that we have constructed were either "naturally" r.s.c. ($R3$) or not sufficiently-complete. (In the latter cases the application of $R3$ often served to indicate how the axiomatization could be successfully completed.) This "experiment" to determine the utility of $R3$ has been supplemented by the following more formal statement of its applicability:

Theorem 7. For any type $T=[Cl(I), S+O]$ all of whose operations are primitive recursive, there exists an axiomatization, A, of a containing algebra (see Theorem 1), such that A is r.s.c. ($R3$). I.e., there exists a partitioning of S into disjoint sets C and E such that

I) For all $c \in C$ and all $o \in O$, the term $o(c(x^*), y^*)$ where x^* and y^* are free is safe, and

II) There exists an ordering of the functions in E, $e1, \ldots, em$, such that $e1$ is convertible to C, $e2$ is convertible to $C+\{e1\}$, etc.

About the Proof. It is first shown that it is possible to construct an r.s.c. ($R3$) specification for the set of basic primitive recursive functions. Call the type with these basic functions and the single output function ?EQUAL?, $Prf=[Cl(\text{Boolean}), S+\{\text{?EQUAL?}\}]$. This will form the basis of an induction. We know, by definition, that any primitive recursive function can be generated from the members of S by a finite number of applications of composition and primitive recursion. We will assume that for any $fm \in Fm$ where Fm is the set of primitive recursive functions that can be constructed using m applications of composition and primitive recursion, there exists a set of axioms such that Fm is convertible to $S+F1+\cdots+Fm-1$. Finally, using this assumption, it is shown that any $fm+1 \in FM+1$ is convertible to $S+F1+\cdots+Fm-1+Fm$.

Proof of Theorem 7.

1) The basic primitive recursive functions consist of one nullary function (call it ZERO), one unary function (call it SUCC), and an infinite number of projection (or "pick out") functions. Since every primitive recursive function can be constructed using only a finite subset of these projection functions, however, we need only show that any (rather than all) projection function can be specified. This is indeed the case, since any axiom of the form $f(x1, \ldots, xn)=xm$, $1 \leq m \leq n$,

is safe if $f \in O$, or obeys constraint I of Lemma D if $f \in S$. The basic type, *Prf*, for the primitive recursive functions, with ?EQUAL? as the output function, may be r.s.c. ($R3$) specified as follows:

Operations:

ZERO: $\rightarrow Prf$
SUCC: $Prf \rightarrow Prf$
?EQUAL?: $Prf \times Prf \rightarrow$ Boolean

Axioms:

?EQUAL? (ZERO, ZERO) = true
?EQUAL? (ZERO, SUCC(x)) = false
?EQUAL? (SUCC(x), ZERO) = false
?EQUAL? (SUCC(x), SUCC(y)) = ?EQUAL?(x, y)

2) Now, assume that any $fm - FM$ is convertible to $S + F1 + \cdots + Fm - 1$.
3) Consider $fm+1 \in Fm+1$. $fm+1$ must be defined by either

a) $fm+1(x0, x1, \ldots, xn) = h(g0(x0, \ldots, xn), \ldots, gk(x0, \ldots, xn))$, where $h, g0, \ldots, gk$ are all contained in $S + F1 + \cdots + Fm$, or

b) $fm+1(\text{ZERO}, x1, \ldots, xn) = g(x1, \ldots, xn)$

$fm+1(\text{SUCC}(x), x1, \ldots, xn) = h(fm+1(x, x1, \ldots, xn), x, x1, \ldots, xn)$

where g and h are contained in $S + F1 + \cdots + Fm$.

Note that axioms that conform to either of these forms obey the restrictions outlined in Lemma D. Thus $fm+1$ must be convertible to $S + F1 + \cdots + Fm$. q.e.d.

There are, therefore, a large number of abstract types for which there exist axiomatizations that are r.s.c. ($R3$). The fact that the conditions outlined above are purely syntactic means that $R3$ can be quite simple. It also means that there are a large number of sufficiently-complete axiomatizations that will not be recognized as such. In practice, one would almost certainly wish to add some primitive rules of inference to any procedure designed to check for sufficient-completeness. One would, for example, like to be able to infer the safety of the term ?EQUAL?(ZERO, SUCC(x)) from the axioms ?EQUAL?(x, y) = ?EQUAL?(y, x) and ?EQUAL?(SUCC(x), ZERO) = false. A detailed discussion of inference techniques is, however, outside the scope of this paper.

An examination of the conditions comprising $R3$ leads to an informal heuristic "procedure" that has proven to be useful in constructing sufficiently-complete axiomatizations of abstract data types:

1) Partition the operations of the type, $T = [I, S + O]$ into the sets O, C, and E as defined above. (Note that there may be several such partitionings.)

2) Build the set CTERMS $= \{c(x1, \ldots, xn) \mid c$ is an n-ary function $\in C$ and $\forall\, 1 \leq i \leq n\ [xi$ is a free variable$]\}$.

3) Build the set OTERMS $= \{o(x1, \ldots, xn) \mid o$ is an n-ary function $\in O$ and $\forall\, 1 \leq i \leq n$ [if the ith argument in the domain of o is $\in I$ then xi is a free variable, otherwise $xi \in$ CTERMS$]\}$.

4) Build the set ETERMS $=\{e(x1, \ldots, xn) \mid e$ is an n-ary function contained in E and $\forall 1 \leqq i \leqq n$ [if the ith argument in the domain of o is $\in I$ then xi is a free variable, otherwise $xi \in$CTERMS]$\}$.

5) Consider a type $T'=[I, C+O]$. The set OTERMS may be used as a set of left hand sides in constructing a safe axiomatization of T'. Construct, using the conditions of Theorem 4 for guidance, such an axiomatization.

6) Complete the axiomatization of T by adding axioms that demonstrate the convertibility of all members of E. The members of ETERMS form a sufficient set of left hand sides. Theorem 5 may be used for guidance in supplying the right hand sides.

Programmers tend to regard this procedure as strange and somewhat intimidating when it is first presented; however, we have found that they can usually master it after working a few examples, and move on with little difficulty to data types that appear in their applications. The most common types of problems are the following:

1) Although the partitioning of the operations into the sets S and O can be done purely on the basic of their ranges, the partitioning of S into C and E requires some consideration of their semantics, and is not always done correctly. It is not always easy to apply the informal rule that constructors produce new values of the type, while extensions merely produce values that could have been produced in other ways. In practice, the sympton of an incorrect partitioning is generally a left hand side for which a "simpler" right hand side cannot be supplied. E.g., within type Stack, if POP were classified as a constructor, rather than as an extension, OTERMS would contain TOP(POP(s)); if PUSH were classified as an extension, rather than a constructor, ETERMS would contain PUSH(NEW, i).

2) There is a strong temptation to put too much in a single type. As a rule of thumb, we have found that types with two to four constructors are quite manageable, but that types with more than four constructors are generally more easily (and more clearly) specified by decomposing them into simpler types. (An exception to this rule of thumb is a type with a large number of constants, e.g., type Character.)

3) Sometimes the operations of a type are most naturally specified by introducing additional "private" functions that are both defined and used within the specification, but that never explicitly appear in programs that use the type. Frequently these private functions will be output functions that indicate the "history" of a value: the sympton indicating the need for them is the inability to construct a right hand side without more information about the structure of the left hand side than is provided by our restricted definition of LHS. Note that if "private" functions are flagged as such, thay need not be included in implementations of the type.

4) When constructing right hand sides, it is easy to overlook an acceptable one (in terms of Theorem 7) because it doesn't seem any "simpler" than the left hand side. E.g., in defining a queue, the essential observation is that if q is non-empty, REMOVE(ADD(q, i)) can be reduced to ADD(REMOVE(q), i).

Despite these problems, the discipline imposed by our restricted form of axioms seems to simplify the problem of producing axioms for any well-understood type. We will discuss both these guidelines and their application in greater detail in a future paper.

3.7. Some Remarks on Consistency

The consistency of an arbitrary set of axioms is, like completeness, an unsolvable problem. There are, however, many techniques that are sufficient to guarantee consistency. Two seem particularly germane in the current context.

The construction of a model is perhaps the most widely used technique for establishing the consistency of axiom sets. To show that an axiomatization of an abstract type is consistent, it is sufficient to construct a provably "correct" implementation of the type. (What it means and how to go about proving the "correctness" of implementations of abstract types is discussed in separate papers [7, 8].) From a practical point of view, this is often the best way to demonstrate consistency. If the abstract type is to be used, an implementation must eventually be provided. Hopefully, a proof of the "correctness" of this implementation will also be provided as a matter of course. If this is the case, the proof of consistency comes "free" as a side-effect. The danger in adopting this approach to showing the consistency of a specification is that if the specification is inconsistent, it is possible to waste considerable time trying to build a model that cannot be built. This problem can be avoided by proving the consistency of a specification before trying to implement it. One way to do this is to demonstrate that the axiomatization has the Church-Rosser property [4].

The conditions for establishing sufficient-completeness presented in Section 3.6 were based upon the monotonicity of the axioms, i.e., "$=$" was not treated as a symmetric relation. What was actually shown was that the conditions presented are sufficient to guarantee that for any term $o(x, y^*)$, $o \in O$, there exists a series of reductions,

$$o(x, y^*) \to z1 \to z3 \to \cdots \to z, \quad \text{where} \quad z \in Vi \in I.$$

Any r.s.c. ($R3$) axiomatization may, therefore, be viewed as a set of replacement rules. Thus, for any such axiomatization, consistency may be demonstrated by proving that the set of replacement rules exhibits the Church-Rosser property. Informally, a set of replacement rules is Church-Rosser if whenever one applies a replacement rule to reduce a term, and then a rule to reduce the resulting term, etc., until there is no longer an applicable rule, the final result does not depend upon the order in which the rules were applied. More formally:

Definition. If a term, t, can be reduced to another term, t', by a single application of a replacement rule, then $t \to t'$.

Definition. $\to *$ is the reflexive transitive closure of $\to$.

Definition. A set of replacement rules, S, is *Church-Rosser* if and only if

$\forall t1, t2, t3 \in L(T)[(t1 \to * t2 \ \& \ t1 \to * t3)$
implies that there exists $t \in L(T)$ such that $(t2 \to * t \ \& \ t3 \to * t)]$.

There are many ways in which replacement systems may be shown to be Church-Rosser: Rosen [21] provides a useful survey.

4. Related Work

The work which most closely resembles that described in this paper is that of Zilles [26] and Goguen et al. [6]. We will discuss one fundamental distinction between our approach and the other two: there are also definite differences in emphasis and numerous technical differences which we will not pursue here.

Recall our assertion that the significance of values contained in TOI rests solely in the effect that those values have when they occur as arguments to functions contained in O. This assumption led us to the conclusion that values contained in TOI may be assumed to be the same unless provably different. [26] and [6] make the opposite assumption, i.e., that values must be considered different unless they are demonstrably equal.

This difference in viewpoint is formally expressed in terms of the congruence relations defined over TOI by the axioms of the type. Zilles says that "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 sequence 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" [26]. We, on the other hand, permit any congruence relation that satisfies the axioms. An example may help to clarify this distinction.

Consider the abstract type Set defined in Section 2. Consider also two values of type Set, x and y, such that x is defined by the expression INSERT(INSERT(EMPTY_SET, 2), 3) and y by the expression INSERT(INSERT(EMPTY_SET, 3), 2). It can be readily seen that the theorem $x = y$ cannot be derived from the axioms for type Set. It is also clear that one cannot derive the theorem $\neg$(MEMBER_OF? (DELETE (x, z), $z1$) = MEMBER_OF? (DELETE (y, z), $z1$)) for any $z, z1 \in$ Integer. Therefore, while the smallest congruence viewpoint leads one to place x and y in different equivalence classes, our viewpoint allows us to place them in the same equivalence class.

Acknowledgments. We would like to thank John Lipson for his help with the research described in this paper and Jim Thatcher, Hans-Josef Jeanrond and the referee for suggestions that led to material improvements over the version of this paper first submitted in 1976. We would also like to thank Wlad Turski for his helpful advice and encouragement. Members of IFIP Working Group 2.3 (Programming Methodology) provided useful criticism.

References

1. Birkhoff, G., Lipson, J.D.: Heterogeneous algebras. J. Combinatorial Theory **8**, 115–133 (1970)
2. Boon, W.: The word problem. Ann. of Math. 207–265 (1959)
3. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. D.A.I. Research Report No. 19, Department of Artificial Intelligence, University of Edinburgh, March 1976
4. Church, A., Rosser, J.: Some properties of conversion. Trans. Amer. Math. Soc. **39**, 472–482 (1936)
5. Dahl, O.-J.: The SIMULA 67 common base language. Norwegian Computing Center, Oslo, Norway, 1968
6. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.G.: Abstract data-types as initial algebras and correctness of data representations. Proceedings Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975
7. Guttag, J.V.: The specification and application to programming of abstract data types. Department of Computer Science, University of Toronto, Toronto (Canada), Ph.D. Thesis, 1975. Available as Computer Systems Research Group Technical Report CSRG-59
8. Guttag, J.V.: Abstract data types and the development of data structures. Comm. ACM **20**, 396–404 (1977)
9. Guttag, J.V., Horowitz, E., Musser, D.R.: Abstract data types and software validation. USC Information Sciences Institute Technical Report ISI/RR-76-48, 1976
10. Henderson, P., Snowden, R.: An experiment in structured programming. Nordisk Tidskr. Informationsbehandling (BIT) **12**, 38–53 (1972)
11. Hermes, H.: Enumerability, decidability and computability, New York: Academic Press 1965
12. Hoare, C.A.R.: Proofs of correctness of data representation. Acta Informat. **1**, 271–281 (1972)
13. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. Acta Informat. **2**, 335–355 (1973)
14. Kleene, S.: General recursive functions of natural numbers. Math. Ann. **112**, 729–745 (1936)
15. Liskov, B.H., Zilles, S.N.: Programming with abstract data types. Proceedings of ACM SIGPLAN Symposium on Very High Level Languages. SIGPLAN Notices **9**, 50–59 (1974)
16. Majster, M.E.: Limits of the algebraic specification. SIGPLAN Notices **12**, 10, 37–41 (1977)
17. Minsky, M.: Computation: Finite and infinite machines, New York: Prentice-Hall 1967
18. Morris, J.H.: Types are not sets. ACM Symposium on the Principles of Programming Languages, pp. 120–124, October 1973
19. Palme, J.: Protected program modules in SIMULA 67. FOAP Report C8372–M3 (E5), Research Institute of National Defense, Stockholm, Sweden, 1973
20. Parnas, D.L.: A technique for the specification of software modules with examples. Comm. ACM **15**, 330–336 (1972)
21. Rosen, B.: Tree manipulating systems and Church-Rosser theorems. J. Assoc. Comput. Mach. **20**, 160–187 (1973)
22. Spitzen, J., Wegbreit, B.: The verification and synthesis of data structures. Acta Informat. **4**, 127–144 (1975)
23. Standish, T.A.: Data structures – an axiomatic approach. BBN Report No. 2639, August 1973
24. Thatcher, J.W., Wagner, E.G., Wright, J.B.: Data type specification: Parameterization and the power of specification techniques. Private communication, 1977
25. Wulf, W.A., London, R.L., Shaw, M.: Abstraction and verification in Alphard: introduction to language and methodology. Carnegie-Mellon University and USC Information Sciences Institute Technical Reports, 1976
26. Zilles, S.N.: Data algebra: a specification technique for data structures. Massachusetts Institute of Technology, Cambridge (Mass.), Ph.D. Thesis, 1978 (forthcoming)

Received June 21, 1976