The University of Warwick

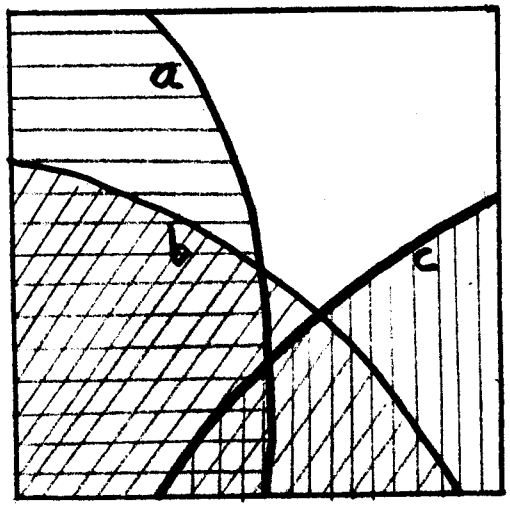
THEORY OF COMPUTATION

REPORT NO. 46

CLASSIFIED ALGEBRAS

BY

W. W. WADGE



Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND.

October 1982

Abstract

We present a new formal system for the specification and verification of abstract data types, one which allows subtypes and polymorphism.

The new system is based on a modified notion of algebra, namely that of classified algebra. A classified algebra (our terminology) is essentially a single sorted algebra together with a classification (family of subsets) of its universe. The classification, which is not necessarily a partition, is labelled by sort (type) symbols; the subset of the universe labelled by a sort symbol s are the objects of 'type' s.

An assertion in the new system is either an equation, which asserts that two expressions always have the same value, or a declaration, which asserts that the value of a particular expression is always of a particular type. Equations and declarations have equal status and the rules of inference (substitution and replacement) can be applied to both.

We show that any specification (set of assertions) has an initial model, unique up to isomorphism, which we can take to be the family of data types and operations specified. The declarations in a specification act as the 'generators' of the types, and this principle forms the basis for an induction rule of inference for proving assertions about initial models.

1. Why data types are not (just) many sorted algebras

The basic principle of the "abstract" approach to data types (see, for example, [6]) is the following: a data type is determined by the operations allowed on the data objects in question. From the mathematical point of view, the basic principle is usually taken to mean that the formal study of data types is not just the study of sets of data objects, but rather the study of algebras - an algebra being a set together with a collection of operations over the set.

For example, we could study the notion of "list" (in the sense of LISP) by studying an algebra of lists together with the operations CAR, CDR, CONS etc. One great advantage of this method is the fact that assertions about data types and operations (in particular, specifications) can be formulated in the simple equational language of universal algebra. Properties can be expressed by equations such as

$$CAR(CONS(X,Y)) == X$$

in language very close to that actually used by programmers, rather than in some abstruse metamathematical formalism intelligible only to specialists.

In most cases, however, it is not possible to define a data type in terms of operations over the single data type in question. If we are concerned with stacks of integers, for example, we must deal with operations like PUSH and TOP some of whose arguments and/or results are integers, not stacks. For this reason it is usual to use many-sorted (or "heterogenous"; see [1]) algebras (MSAs) rather than simple single sorted algebras.

The initial success of the MSA formalism made it seem for a time that the basic foundational problems had been solved, that data types "are" MSAs. Unfortunately this has not proved to be the case; a number of problems which at first appeared to be minor have emerged to cause enormous difficulties.

The first objection is really just a matter of terminology. The objection is that an MSA is not a single data type, but rather a collection (or "cluster") of data types and associated data operations. This objection

may seem trivial, but important issues are involved. In our opinion, the naive view of what is a data type ("a collection of similar data objects") is basically correct. The trouble with naive (ie set-theoretical) approaches to specification is that they ignore the data operations. In other words, data types are sets, but the study of data types cannot be just the study of sets. We must study sets together with operations on their elements, ie algebras.

The second problem, this time a serious one, is that of 'error' values resulting from nonsensical combinations. It arises in all but the simplest data types. In specifying stacks, for example, we quickly discover that we must include an equation which gives us a value for the expression POP(NIL) (which, according to the type system, must be a stack). In general there are two ways of dealing with the problem: by assigning healthy 'default' values to the problem combinations, or by introducing special error objects. Neither is very satisfactory. The first tries to ignore the problem of errors and makes it very difficult even to formulate the idea of a safe program, eg one which does not attempt to pop empty stacks. The second greatly complicates the specifications because it is necessary to qualify some equations with preconditions to the effect that the variables involved denote non-error values.

A second serious problem with MSAs is the lack of polymorphism. In real programming languages it is common practice to use the same symbol for operations with different argument and result types (the symbol "+" is the usual example). If we want to remain within the MSA formalism,however, we must employ a different symbol for each separate use. For example, if we are dealing with lists of different kinds of objects (eg lists of numbers and lists of characters) we need distinct versions of the list operations - an operation NCAR for taking the head of a numeric list, an operation CCAR for taking the head of character list, NCONS for constructing numeric lists, and so on. Worst of all, we need a whole family of IF-THEN-ELSE clones, one for each type of data which might be involved in a choice. Furthermore each of these operations needs its own copy of the 'generic' specifying equations.

A third serious problem, related to the previous two, is that of subtypes. The MSA formalism requires that the various carriers be disjoint sets. In practice, however, there often arise situations in which one type (as a set of values) is naturally considered as a subtype of another. The classical example is given by the types integer and real. Before FORTRAN, everyone agreed that integers were real numbers of a special kind. Now many languages force us to distinguish between the integer 3 and its close cousin, the real number 3.0. Some of these languages have gone even further and enlarged the family of three-ish objects to include 3.0D00 (double precision three), (3.0,0.0) (complex three), (3.0D00,0.0D00) (double precision complex three), (3,0) (integer complex three) etc. etc. In the same way, we must distinguish between an empty list of integers, an empty list of characters, and an empty list of booleans. We are forbidden from talking about a general type list and cannot even contemplate 'mixed' lists of (say) booleans and characters alternating.

Another shortcoming of the MSA system is the lack of what we might call 'parametrised' types. Suppose, for example, that we wished to axiomatise the type 'height balanced binary tree'. The problem is that we cannot combine two arbitrary height balanced trees (by making them the two immediate subtrees of a new tree) and expect the result to be height balanced. It is necessary that the two trees be of almost the same height. What we need is a formalism which allows sorts with parameters, for example hbt(n). The closest we can come to this idea in the MSA system is to have an infinite collection hbt0, hbt1, hbt2, ... of sorts, one for each possible height. This approach, however, requires infinite signatures and infinite specifications, and so is far from satisfactory.

The final complaint against the MSA formalism, perhaps the most serious, is that its treatment of types is not really algebraic. The basic principle of the algebraic approach is that it is not the objects which are important, but rather the operations on them. If we apply this principle to the study of

data types, we are forced to conclude that the important things to study are the operations on data types. It is very easy to imagine useful operations on data types: for example, the union of two types (all objects of either type), or the cross product (all ordered pairs of objects with the first component of the first type, the second component of the second type). Even so-called "types" like stack are really (unary) operations on types; if t is a type, stack(t) is the collection of stacks whose components are of type t. However, the MSA formalism simply does not allow operations on types (sorts) in any form, and no amount of metamathematical manipulations (eg studying families of MSAs) can compensate for this fundamental deficiency.

The shortcomings of the MSA formalism, in particular the lack of subtypes and polymorphism are well known. Such problems arise in any language or system based on a simple-minded 'pigeonholing' approach to data types. Anyone who has tried to implement lists in PASCAL, for example, will understand the need for polymorphism. Subtypes and polymorphism, as well as parametrised types an operations on types, were already discussed by the author and A. Shamir in [5]. In that work, however, we did not take up the problem of formal specification.

2. Generalizations of the notion of signature

The most natural way to solve these problems is to generalise the notion of a signature and of the type of an operation. In an MSA the the sorts form an unstructured set, and each operation has only one type, consisting of a sequence of sorts - one for each argument place and one for the result. If we want polymorphism, for example, we could allow an operation symbol to have more than one type; and if we want subtypes we allow a partial order on the sort symbols. Goguen has described an "order-sorted algebra" approach in [2] which incorporates these two generalisations (his system developed out of earlier efforts to solve the error problem).

Goguen's suggestion is certainly valuable, and order sorted algebras do in fact solve some of the problems raised here. However once there are subtypes a new problem appears, one which did not arise with the old 'pigeon-holing' MSA approach. The OSA (Order Sorted Algebra) formalism is limited by the fact that the type of an expression is still determined by the types of the subexpressions. As a result, it is not really possible to reason about the type of the value of an expression, ie about special values an expression may have because of special properties of the algebra.

Suppose, for example, that we have a signature with types integer and real with the former a subtype of the latter. The equation

$$X_{real} - X_{real} == 0$$

(here X_{real} is a variable of type real) is well typed (0 is of type integer and therefore also of type real) but the type system prevents us from substituting some occurrences of the right hand side by the left hand side. We cannot deduce the equation

$$Y_{integer} + (X_{real} - X_{real}) == Y_{integer}$$

from the equation

$$Y_{integer} + 0 == Y_{integer}$$

because the former is simply not well-typed.

As a second example, suppose that we have a signature with types integer and stack (of integer). We all 'know' that the expression

$$IF\ EMPTY(T_{stack})\ THEN\ 5\ ELSE\ TOP(T_{stack})$$

is safe and yields an integer value. The type system, however, is unaware of this fact because it ignores the nature of the test and abstracts only its type (boolean). The best that we could conclude using an OSA (order- sorted algebra) system is that the above expression is of a supertype of integer which includes error objects as well.

Goguen himself recognises this problem, an extends his simple OSA by allowing new syntactic objects called declarations. A declaration is a expression together with a sort, for example (in Goguen's notation)

$$int:\ X_{real} - X_{real}$$

which is interpreted as asserting that value of the expression in question is always of the indicated type, for all correctly typed assignments of values to variables. Goguen's declarations have an unusual status, however, and seem to be considered more as part of the signature than as part of the specification. They complicate the system even further without decisively resolving the problem of 'special case' types.

3. Classified Algebras

The conventional MSA system and the new OSA system (even the extended version with declarations) all share a fundamental assumption (with many other languages and systems as well) which we feel is at the root of some of the most serious problems in the field of data types. Both the MSA and OSA systems are based on a concept of type which is primarily syntactic: a type system is seen as being above all a classification of syntactic objects, ie of expressions. The classifications permitted by the OSA system are more sophisticated than those allowed by the MSA one, but the principle remains the same. The OSA formalism can (and should) be generalised even further, say to AASs (Algebra Sorted Algebras) to handle operations on types. Yet the blind spot (the inability to talk of the type of the value of an expression) will remain. With more elaborate signatures, the problem will get worse. For example, it should be quite normal practice to write a program that produces a height balanced tree, but not using a method so simple that a syntactic type checker can verify the fact unaided. We propose to solve this problem by developing a notion of algebra based on a semantic concept of type: we see type systems as being primarily classifications of semantic objects, ie of data objects. We call these new kinds of algebras classified algebras. The CA (classified algebra) system is more general than the OSA system, and is at the same time (mercifully) notationally much simpler. We do not pretend that CAs solve all the problems present in the algebraic approach; our intention is rather to present them as a simple example of a semantically based system.

The new system can be thought of as the result of carrying Goguen's enrichment of an OSA system to its logical conclusion. We promote his declarations to the status of full fledged assertions, sharing the same rights and privileges as equations. Since declarations can be used to formulate all the type information about operations 'coded up' in an OSA signature, we can rely on declarations alone and drop any a priori syntactic concept of type. A signature in the new system is simply a collection of sort symbols and operation symbols (we assume they can be distinguished) which includes the special sort symbol anything. We even drop any syntactic notion of rank, although rankings could be retained for those uncomfortable in their absence. By an operation over a set X we mean a function from the set of finite sequences of elements of X to X. Then given any signature Sg, a Sg-classified algebra A is a function with domain Sg such that

(i) A(anything) is a nonempty set, called the universe of A;

(ii) A applied to any sort symbol in Sg yields a subset of the universe of A;

(iii) A applied to any operation symbol in A yields an operation over the universe of A. A classified algebra is therefore just a conventional (apart from being rank- less) single sorted algebra together with a labelled classification of the elements of its universe. The sets {A(s) : s a sort symbol other than anything} correspond to the carriers in a conventional MSA. In general, there will be elements of the universe of a CA A which do not appear in any of its 'carriers'. These are best thought of as error objects or even 'junk' which is the result of nonsensical combinations.

The syntactic side of the CA system is equally simple. Expressions (terms) are built up from variables by combining operation symbols with with operand expressions - the choice and number being arbitrary. An equation is an ordered pair <E,E'> of terms; we will also use the notation

$$E == E'.$$

A declaration is an ordered pair $\langle E,s \rangle$ with E a term and s a sort symbol; we will also use the notation

$$E{:}s.$$

Variables are typed (or sorted), as in the MSA system, and so can be thought of as ordered pairs consisting of a token together with a sort symbol.

A declaration $E{:}s$ is true in an algebra A iff the value of the expression E is of type s (ie is in $A(s)$) for all properly typed assignments of values to variables. Thus the declaration

$$\mathrm{PRODUCT}(V_{\mathrm{vector}},\mathrm{TRANSPOSE}(V_{\mathrm{vector}})){:}\mathrm{posreal}$$

asserts that the operation $A(\mathrm{PRODUCT})$ applied to an element of $A(\mathrm{vector})$ and to $A(\mathrm{TRANSPOSE})$ applied to that element is in $A(\mathrm{posreal})$.

An assertion is either an equation or a declaration. An assertion P is true in an algebra A iff it is true as an equation or as a declaration, as above.

Declarations are intended to allow explicit formulation of the type properties of expressions which in other systems are formulated implicitly by the signature (which assigns operation types to operation symbols). The availability of declarations relieves us from the necessity of encoding type information in this syntactic form (in the signature) and at the same time is much more general as well.

For example, the fact that the operation symbol PUSH is of type

$$\langle \mathrm{stack},\mathrm{int},\mathrm{stack} \rangle$$

can be expressed by the declaration

$$\mathrm{PUSH}(T_{\mathrm{stack}},I_{\mathrm{int}}){:}\mathrm{stack}$$

We can, of course, write more than one declaration concerning the same operation; for example,

$$X_{\mathrm{int}} + Y_{\mathrm{int}} : \mathrm{int}$$

$$X_{\mathrm{real}} + Y_{\mathrm{real}} : \mathrm{real}$$

$$X_{\mathrm{pos}} + Y_{\mathrm{pos}} : \mathrm{pos}$$

so that we can formalise the notion of an operator being "polymorphic". The ordering between types can also be formulated with declarations: the

declaration

$$X_{int} : real$$

asserts that type int is a subtype of type real. This declaration is true in an algebra A iff A(int) is a subset of A(real).

Finally, declarations can be used to give type information about 'special case' expressions which cannot be deduced just from type information concerning the operation symbols used in the expression. Here are three examples:

$$X_{real} - X_{real} : int$$

$$IF\ T\ THEN\ X_{int}\ ELSE\ Y_{real} : int$$

$$IF\ X_{real} > 0\ THEN\ SQRT(X_{real})\ ELSE\ SQRT(-X_{real}) : real$$

Here is a specification of the natural numbers with the operations zero (0), successor ('), addition (+) and multiplication (*).

$$0 : nat$$

$$X_{nat}' : nat$$

$$X_{nat} + Y_{nat} : nat$$

$$X_{nat} * Y_{nat} : nat$$

$$X_{nat} + 0 == X_{nat}$$

$$X_{nat} + Y_{nat}' == (X_{nat} + Y_{nat})'$$

$$X_{nat} * 0 == 0$$

$$X_{nat} * Y_{nat}' == (X_{nat} * Y_{nat}) + X_{nat}$$

and here are additional assertions which specify stacks of natural numbers with POP, TOP and PUSH. The extra sort symbol nstack stands for non-empty stacks.

$$NIL : stack$$

$$W_{nstack} : stack$$

$$PUSH(V_{stack}, X_{nat}) : nstack$$

$$POP(W_{nstack}) : stack$$

$$TOP(W_{nstack}) : nat$$

$$TOP(PUSH(V_{stack}, X_{nat})) == X_{nat}$$

$$POP(PUSH(V_{stack}, X_{nat})) == V_{stack}$$

4. Theories and initial models

Classified algebras were developed primarily to facilitate the specification of abstract data types. A specification is a collection of assertions which somehow 'axiomatises' the algebra of the intended data types and operations. In general, a specification has many different models; that is,there are many different algebras in which all the assertions in the specification are true. There is, however, one particular collection of algebras associated with the specification which have an excellent claim to be the algebras 'intended'. These are the initial models of the specification.

An algebra A which is an element of a class K of algebras is said to be initial (in K) iff given any other algebra B in K there is a unique homomorphism from A to B. A model of a set of assertions is an initial model iff it is initial in the class of all models of the set. It has been shown that in the MSA and OSA systems a specification always has an initial model, and that this initial model is unique up to isomorphism. A specification can therefore be considered as defining a particular isomorphism class of algebras - this is the justification of the initial algebra approach to data type specification (see, for example, [3]).

The motivation behind the formal definition is hard to guess; but fortunately there is also a suggestive informal characterisation. The initial model is the one which

(i) includes only those data objects which are required by the specification to exist;

(ii) identifies only those objects which the specification requires to be identified. The initial algebra is therefore in a certain sense the 'minimal' solution to the specification.

If we want to use the initiality principle in conjunction with CAs, we must obviously prove that every CA specification has an initial model unique up to isomorphism. This is not particularly difficult, and we present the proof here in outline only.

The crucial concept is that of homomorphism. By a homomorphism from a CA A to a CA B we mean a function h from the universe of A to that of B such that

(i) for any operation symbol f and any elements $a_0, a_1, \ldots a_{n-1}$ in the universe of A, the result of applying B(f) to $h(a_0)$, $h(a_1), \ldots h(a_{n-1})$ is the image (under h) of the result of applying A(f) to $a_0, a_1, \ldots, a_{n-1}$;

(ii) for any sort symbol s and any element a of the universe of A, if a is in A(s) then h(a) is in B(s).

Now suppose that we have a specification (set of assertions) S and a signature Sg. We will construct a particular algebra A which is an initial model of S.

The universe of A consists of equivalence classes of Sg-expressions, two expressions E and E´ being equivalent off the equation E==E´ is a logical consequence of the assertions in S (ie iff the equation is true in all models of S). Given any sort symbol s in Sg, A(s) is the set of all equivalence classes of expressions E for which E:s is a logical consequence of S. Finally, if f is an operation symbol in Sg and $[E_0], [E_1], \ldots, [E_{n-1}]$ are in the universe of A, then the result of applying A(f) to $[E_0], [E_1], \ldots, [E_{n-1}]$ is $[f\langle E_0, E_1, \ldots E_{n-1}\rangle]$, ie the equivalence class of the term formed from f with $E_0, E_1, \ldots E_{n-1}$ as operands.

In order to check that A is well defined we must verify that

(i) if E:s follows from S, and E´ is equivalent to E, then E´:s follows from S;

(ii) if E_i is equivalent to E_i´ (i<n) then $f\langle E_0, E_1, \ldots, E_{n-1}\rangle$ is equivalent to $f\langle E_0', E_1', \ldots, E_{n-1}'\rangle$.

(s a sort symbol and f an operation symbol). This is straightforward.

Next, we must show that A itself is a model of S. This follows from the more general but easily established result (important in its own right) that an assertion (which may involve variables) is true in A iff each of its variable - free substitution instances is a logical consequence of S. (The substitutions allowed are those which involve replacing a variable of type s

by an expression E for which E:s follows from S). The result is hardly surprising, considering that the elements of the universe of A are (equivalence classes of) variable - free expressions. Since all the substitution instances of assertions in S are obviously consequences of S, it follows that A is a model of S.

Finally, it is easy to see that A is an initial model. Given any other model B of S, we define the function h from the universe of A to that of B by setting h([E]) to be the value of E in B (recall that E has no variables). The fact that B is a model of S ensures that h is well defined and is a homomorphism.

Clearly, the proof of the existence of initial models for CA specifications differs little from analogous proofs in the other systems.

5. Reasoning about Classified Algebras

One of the great attractions of equational algebra as a logical system is the extreme simplicity of the rules of inference. The two rules are, of course, the rule of substitution (of expressions for variables) and the rule of replacement (of equals for equals). The CA system uses these same rules, with appropriate modifications. The essential difference is that the CA rules can be used to derive new declarations as well as new equations.

The rule of substitution allows us to infer, given an assertion P and a declaration E:s, any assertion P′ formed from P by substituting E for all occurrences in P of any one variable of type s. Thus from the assertion (in this case, a declaration)

$$X_{pos} + Y_{pos} : pos$$

and the declaration

$$Z_{int} * Z_{int} : pos$$

we can infer the declaration

$$X_{pos} + (Z_{int} * Z_{int}) : pos.$$

Substitution can, as indicated, be performed on equations as well.

The rule of replacement allows us to infer, given an assertion P and an equation E==E', any assertion P' formed from P by replacing any occurrence of E in P by E'. Thus from the assertion (again, a declaration)

$$X_{even} + (2 * Y_{int}) : even$$

and the equation

$$2 * Y_{int} == Y_{int} * 2$$

we can infer the declaration

$$X_{even} + (Y_{int} * 2) : even.$$

Of course replacement can, like substitution, be applied to equations as well.

It can be shown (though we will not do it here) that these rules (together with the obvious axioms) are complete: an assertion P can be proved from a set S of assertions iff P is a logical consequence of S, ie iff P is true in all models of S.

Now suppose that we have a set S of assertions which we want to consider as a specification, and that we are interested in proving certain properties of the data types specified. The initial model of a specification is a model; thus any assertions which can be derived using the rules of inference just given are true in the initial model. Assertions derived in this way, however, are those which are true in all models of the specification. In general there are assertions true in the initial algebra which are not true in other models of the specification and so cannot be derived using the ordinary rules of inferences. The commutative law of addition, for example, is not a consequence (in the ordinary sense) of the specification of the natural numbers given earlier.

Obviously, we require stronger rules of inference which which take into account the special features of the initial model. We saw earlier that the universe of the initial model contains only those data objects required to exist by the specification. In the CA system we can restate these properties in a more suggestive form : the elements of the initial algebra are exactly those which are generated by the declarations in the specification. This formulation points, of course, towards a rule allowing assertions to be proved

by induction on the complexity of the structure of the data objects of a given type. We have in mind some form of "generator induction" (see, for example, [5]) which extends the classical rule of mathematical induction over the natural numbers.

An induction rule of this type is easily found. Suppose that Sg is a signature, that S is a specification, that s is a sort symbol in S and that P(w) is an assertion involving the variable w of type s (and possibly others, possibly of other types as well). Let E_0:s, E_1:s,...,E_{n-1}:s be all the declarations for the type s in S. We form a new sequence E_0', E_1',...,E_{n-1}' of terms in which each E_i' is the result of replacing all variables of type s in E_i by new operation symbols (which will be used as nullaries) not already in Sg. Let a_0, a_1,..., a_{m-1} be the new symbols. Then we are required to prove, using the ordinary rules of inference, the assertions $P(E_i')$ (i<n). In doing so we use S, the declarations c_j:s (j<m), and the induction hypotheses $P(c_j)$ (j<m). Having done so, we can conclude that P(w) is true in the initial model of S.

The rule is simple enough, but in attempting to apply it to the example specifications given earlier we run into an unexpected difficulty. In proving assertions about the natural numbers, for example, we would expect only two 'cases' : proving P(0) and proving P(a') assuming P(a). According to the rule, however, we must also prove two other cases: $P(a_0+a_1)$ from $P(a_0)$, and $P(a_0*a_1)$ from $P(a_0)$ and $P(a_1)$. The extra steps are unnecessary and often impossible, so that our induction rule proves to be of limited usefulness.

Of course we all 'know' that the declarations 0:int and X_{int}':int by themselves are enough to generate the integers. In general, however, it would be very difficult to say which of a set of declarations are redundant and should therefore not require separate induction steps.

On the other hand, if the declarations

$$X_{int} + Y_{int} : int$$
$$X_{int} * Y_{int} : int$$

really are 'redundant' in a certain sense, we ought to ask ourselves why they

were included. The answer, of course, is that they represent supposedly vital type information about the operations + and *, information given by the signature in conventional algebraic systems. In an OSA or MSA system, this information must be included because a signature must assign types to operation symbols. In the CA system, however, there is no a priori requirement that we include the corresponding declarations in a specification. The extra declarations are, in fact, redundant. The following assertions

$$0:\text{int}$$

$$X_{int}{}':\text{int}$$

$$X_{int} + 0 == X_{int}$$

$$X_{int} + Y_{int}{}' == (X_{int}+Y_{int})'$$

$$X_{int} * 0 == 0$$

$$X_{int} * Y_{int}{}' == X_{int}*Y_{int} + X_{int}$$

specify exactly the same initial algebra. The extra declarations are already true in the initial models of the above specification and can be proved using the induction rule previously stated. To prove, for example, that the sum of two integers is an integer, we must prove

$$X_{int} + 0 : \text{int}$$

and

$$X_{int} + A' : \text{int}$$

from the specifications, the declaration A:int and the induction hypothesis

$$X_{int} + A : \text{int}$$

The proof is very simple. In the same way, we can use the induction rule to prove the associative, commutative, and distributive laws and other important assertions. Several steps are required : assertions proved by one application of induction are added to the specification and used in subsequent induction proofs.

One of the great advantages of this induction rule is that it allows proofs to be formulated entirely within the object language, ie essentially that used by programmers. It does not require knowledge of, or reference to, metamathematical notions such as that of homomorphism.

Incidentally, the other specification also contains redundant assertions. The specification of stack really requires only the five assertions

$$NIL : stack$$

$$T_{nstack} : stack$$

$$PUSH(T_{stack}, I_{int}) : nstack$$

$$TOP(PUSH(T_{stack}, I_{int})) == I_{int}$$

$$POP(PUSH(T_{stack}, I_{int})) == T_{stack}.$$

Of course we could certainly criticise a specification for making no provision at all for errors. But extra error assertions could be added to those above without forcing the creation of any new objects of type stack. The above assertions can be thought of as specifying the bare minimum of what everyone agrees about stacks. In fact the type nstack itself is not really necessary; we could remove the second assertion and replace "nstack" by "stack" in the declaration following. The resulting set of four assertions has an even better claim to represent the 'hard core' of axioms about stacks.

6. Signatures and type checking

It might seem that the adoption of a semantically based system like the one presented here necessitates the abandonment of syntactic type checking. After all, one could reason, in the CA system there are no operation types and anything can be applied to anything.

Any specification (set of assertions) S already determines a classification of the set of all expressions. For each sort symbol s we have the corresponding collection of all expressions E for which E:s is true in the initial model of S. This classification may not, however, be decidable, so that we cannot in general expect to have an algorithm which will check the types of expressions in this sense. For some sets of assertions, however, the corresponding syntactic classification is decidable; in particular, if the assertions are all declarations. Of course, it is very unlikely that a specification would consist only of declarations. But given a specification S

we can form another set T of assertions such that

(i) each element of T is true in the initial model of S (ie is true of the types specified);

(ii) the syntactic classification induced by T is (easily) decidable.

In such a situation the type checking with respect to T is 'partially correct'. If our type checker concludes that a given expression is of a given type, then this will be the case. The only problem is that there may exist complicated expressions that fool the type checker, eg whose value is always an integer even though the best that the type checker can do is classify it as of type real. In many applications, however, partial type information is enough.

Those who approve of the discipline enforced in a strongly typed language may feel very uneasy about the way in which the formal system allows expressions to be built up in an arbitrary fashion allowing eg stacks to be added and integers to be POPed. There is no reason, however, to allow the programmer the full freedom which is available in principle in the formal system. An implementer could always select some set T (as described above) and require that expressions in a program be 'classifiable' according to T. The strength of resulting type discipline depends inversely on that of T, but there seems to be no a priori limit on what should or should not be allowed to appear in programs. Our formal system therefore takes no stand on the issue and leaves the decision where it belongs, in the hands of the language designer.

The approach to type checking presented here is in many respects similar to (and inspired by) that used by Milner in his ML language (as described in [4]).

There is, however, one vital a priori reason for making at least some restriction on the form of programs. The problem is the vast amount of untyped 'junk' floating around even (or especially) in the initial model of a

specification. It is totally unreasonable to expect that the implementor of a specification should have to worry about the junk and provide representations for data objects that are essentially mathematical garbage (such as the result of adding 5 to a truth value, or of using successor as a binary operator). Ideally, the implementor should have to worry about sensible objects only, ie those with a type (other than anything). At the same time, the programmer would be restricted to programs that produce healthy output, ie that produce values with a type.

Unfortunately, it is possible to produce specifications in which junk can appear as intermediate results in computations that produce sensible values. On the other hand, with most 'normal' specifications this cannot happen. Clearly what is required is a formal definition of normality, a simple if not exhaustive criterion for normality of a given specification, and a proof that the junk in the initial model of a normal specification can be safely ignored. This remains to be done.

7. Extensions and Conclusions

The basic innovation in our CA approach is to allow explicit reasoning about the types of expressions by means of declarations. At the same time we have greatly reduced the explicit, syntactic way in which this information encoded in conventional systems. It should be apparent then, this process can be carried even further than we have done already.

For one thing, we could eliminate the syntactic typing of variables by allowing variable type declarations to be used as preconditions for assertions. In this way we would write

$$X{:}int,\ Y{:}int \rightarrow X+Y{:}int$$

instead of

$$X_{int} + Y_{int} : int$$

Of course once we allow preconditions we might as well allow arbitrary declarations as preconditions or even equations as well. This would allow us

to write, for example,

A:posmatrix, A = TRANSPOSE(A) -> A:hermitian

Another direction in which we could extend the system is to allow parametrised types and type operations, the need for which we have already described. The simplest type operations are probably union and intersection, with axioms like

X:A, X:B -> X:A∧B

We could also have a cross product operation on types associated with a pairing function and axioms like

X:A, Y:B -> MKPAIR(X,Y): AxB

Extending the concept even further, we could allow type variables and user-specified operations on types like Stack. The specification of Stack might include assertions like

X:t, H:Stack(t) -> PUSH(H,X):Stack(t)

As for parametrised types, we might specify Vector (with Vector(I) a vector of length I) with assertions like

I:nat, X:Vector(I), Y:Vector(I) -> X+Y:Vector(I)

Finally, we could even raise the order of the system by allowing metatypes which represent classifications of types. An example could be Numeric, with axioms like

t:Numeric, X:t, Y:t -> X+Y:t

In all of these suggested systems, it is necessary that all (or at least most) sets of assertions have initial models, and it is desirable that there be simple induction rules for proving assertions about initial models.

The classified algebra approach outlined in this paper is in a sense, the simplest of all the systems suggested here. Nevertheless we feel that even the CA system is rich enough to illustrate the power and potential of an algebraic system freed from a built-in commitment to some particular syntactic notion of type.

8. References

[1] Birkhoff, G. and Lipson, D., "Heterogenous Algebras," J. Combinatorial Theory 8,1970, pp. 115-133.

[2] Goguen, J. A., "Order Sorted Algebras: exceptions and error sorts, coercions and overloaded operators," Semantics and Theory of Computation report no. 14,Computer Science Department, UCLA, December 1978.

[3] Goguen, J. A., Thatcher, J.W. and Wagner, E. G., "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," in Current Trends in Programming Methodology, Vol. 4, Data Structuring (ed. by R. Yeh), Prentice-Hall, 1978, pp. 80-144.

[4] Milner, R., "A Theory of Type Polymorphism in Programming," JCSS 17 (1978), pp. 348-375.

[5] Shamir, A. and Wadge, W., "Data Types as Objects," Springer Lecture Notes in Computer Science no. 52 (ed. by G. Goos and J. Hartmannis), Springer Verlag, 1977, pp. 465-479.

[6] Spitzen J. and Wegbreit B., "The Verification and Synthesis of Data Structures," Acta Informatica 4 (1975), pp. 127-144.