ABSTRACT DATA TYPES
AND
A SEMANTICS FOR THE ANSI/SPARC ARCHITECTURE*

by

T.S.E. Maibaum

Computer Science Technical Report
CS-02-79

Computer Science Dept.
University of Waterloo
Waterloo, Ontario, Canada

January 1979

*The work outlined in this report was partially supported by a grant from the National Research Council of Canada.

ABSTRACT

The concept of abstract data type developed for modelling complex data structures is combined with an algebraic formalisation of environments as developed in mathematical semantics to provide algebraic models of data base systems. The models improve on previous models in the sense that both queries and updates are integral parts of the same model. Moreover, the algebraic method allows us to define exactly such concepts as "semantic data independence" and "implementation".

These abstract data type models of data bases then allow us to formulate an exact mathematical semantics for the proposed ANSI/SPARC data base architecture. The schemas are represented by abstract data types and the interfaces by the formalisation of the concept of "representation of".

0. Introduction

The purpose of this report is two-fold. Firstly, we intend to show that the theory of abstract data types can be used to provide models for data bases such that the models take account of both queries and updates. That this is possible is not clear from models outlined in the literature (D, T&L). These models tend to concentrate on the static aspects of data base design. Much effort has been put into studying the properties of updates (normal forms in the relational model were motivated by the need to eliminate the so-called update anomalies) but these efforts seem to be in a world quite separate from data base models in which queries are studied. For example, Codd's relational algebra has no facilities for defining updates - new sets can be constructed and tuples inserted or deleted from sets but no operation will assign a new (updated) value to the relation $\bar{R}$ in the set of relations $\bar{R}$ defining a data base. The problem can perhaps be made clear by analogy with the assignment statement in programs. Operations on integers can be used to define values (as relational operations can be used to define relations), but this would not be helpful without the ability of recording these values (by assigning the value to some variable) and changing (updating) the values assigned to variables. We hope to rectify this update "anomaly" in data base models.

Secondly, assuming a data base model including both updates and queries, we would like to provide a simple mathematical model of the ANSI/SPARC data base architecture (T&K, S, Y). The architecture was an attempt at standardisation of data base design concepts but because of its non-mathematical nature, many controversies arose as to its intended meaning. By providing a mathematical model (i.e. a semantics) for the ANSI/SPARC architecture, we hope to provide a sharper focus for these

debates. As to the model itself, the main tool is the concept of abstract data type and the representation of one abstract data type by another (ADJ(1), G, L, L&M(3)).

We adopt the algebraic method for both purposes because it seems to present a number of advantages. For example, the concept of data independence in data base design becomes in the algebraic framework the notion of abstractness (i.e., independent of any representation). This is achieved by taking as the meaning of some specification an isomorphism class of algebras. Thus the meaning (semantics) of a data type is independent of the representation used in any element of the isomorphism class. The algebraic method also allows us to proceed in the time-honoured fashion of computer science (and mathematical logic) - separation of syntax and semantics. The former is provided by a specification of the names of operations and the sorts of arguments they each take and the sorts of values they each return. The latter is provided by an axiomatisation of the behaviour of the operations (i.e., a definition of how the operations behave in combination with each other).

Nor do we take any isomorphism class of algebras satisfying some set of axioms. We must use some unique class and this uniqueness is provided by taking the least isomorphism class satisfying the equations. An element of this least class is called an initial algebra in the class satisfying the equations.

To be a little bit more precise, we provide some definitions and results on which our work is based. Let Σ be an alphabet of operation names. (Σ will in general be a many-sorted alphabet ADJ(1), L, L&N(3)). Let Σ be a set of (equational) axioms over the symbols in Σ. Let $\underline{Alg}_{\Sigma,E}$ be the class of algebras with operations named by Σ and satisfying the

equations in E. An algebra A in a class C is initial if there is a unique homomorphism from A to any B in C.

Theorem: If A and B are initial in C, then A and B are isomorphic.

The above theorem allows us to talk about the initial algebra in a class because initial algebras are unique up to isomorphism. Now, given any set of equations E (over Σ), we have the following result which will be of paramount importance in the sequel. (See C, ADJ(1) for further details and proofs.)

Theorem: The class $\underline{Alg}_{\Sigma,E}$ has an initial algebra which we denote $T_{\Sigma,E}$.

We will talk about $T_{\Sigma,E}$ as the initial algebra in $\underline{Alg}_{\Sigma,E}$. T_{Σ}, the algebra initial in $\underline{Alg}_{\Sigma}$ - the class of all Σ-algebras (the class defined by the empty set of equations) - can be thought of as the algebra of expressions. Given some E, $T_{\Sigma,E}$ is the algebra obtained by grouping together expressions which the equations indicate should be identified. Thus any model for data bases should be $T_{\Sigma,E}$ for some Σ and some E. Moreover, Σ is such that $\Sigma_Q \subseteq \Sigma$ is a set of query (access) operations and $\Sigma_U \subseteq \Sigma$ is a set of update operations. The equations E will indicate the inter-relationships of the query operations amongst themselves and with respect to the update operations.

The justification for this approach is two-fold. Firstly, the usefulness of abstract data types in programming methodology has been amply demonstrated (Z, L&Z(1)-(2), G, L). There is no reason why data base design should not benefit from this experience. Indeed, there has already been some

movement in this direction (M(1), M&L, T, W, EK&W). Secondly, any model which allows us to model queries and updates in the same setting is at least a start in the right direction.

In the next section, we will provide a data base model in the style of abstract data types. We begin with the concept of quotient relations (F&K) as formalised by (T) as an abstract data type. A quotient relation is a set of blocks of tuples as opposed to the usual concept of relations as a set of tuples. The blocks are defined by requiring all tuples of a given block to have certain attributes equivalent. We then modify this model by introducing update operations and the changes necessary to support these operations.

In the third section, we proceed to define an exact mathematical semantics for the ANSI/SPARC architecture by using the theory of representations (for abstract data types) as presented in (ADJ(1)) and then formalised in (L&M(3)). Since this semantics is a mathematical semantics, there can be no argument as to the meaning of certain components of the architecture as long as the semantics is accepted. There can be argument, of course, over whether the semantics is acceptable. However, we hope that the semantics presented is at least a start in the process of formalisation which we feel is necessary in this area.

1. Data Base Models

Suppose that we have a (relational) data base in which there are a number of relations $\mathcal{R}$ amongst which is the relation COURSE defined by the table below:

Figure 1

COURSE ≡

Dept.	Course #	Title
CS	140	Introduction to Mathematical Problem Solving by Computer
CS	180	Introduction to File Processing
Math	124a	Algebra and Geometry

We can query the data base in order to access the information stored therein. We could thus ask for

Title (COURSE) where Dept.(COURSE) = C&O and Course #(COURSE) = 452.

This involves simply projecting out the Title attribute value of any tuple with the other attribute values defined as above. (Thus, in this case we would get "error".) More complicated queries may involve more complicated constructions on the relations of the data base (including joins, cross products, etc.). These constructions use as operands objects built from the relations of the data base, such as COURSE, and "constant" relations (such as the empty n-ary relation for each $n \in \omega$) available to the system. Thus, at the time of this query, the universe of objects (derived relations) which can be constructed in the course of answering queries is fixed and depends only on the values of COURSE (and the other relations $\mathcal{R}$ of the data base) and the values of the "constant" relations. Thus, as far as queries are concerned, the value of COURSE (and the other relations in $\mathcal{R}$) is fixed or constant.

Now consider updating COURSE by

add to COURSE tuple (C&O, 452, Advanced Linear Programming).

We get the following table to represent COURSE (and we suppose for simplicity that the other relations are unaffected by this update):

Figure 2

COURSE ≡

Dept.	Course #	Title
CS	140	Introduction to Mathematical Problem Solving by Computer
CS	180	Introduction to File Processing
Math	124a	Algebra and Geometry
C&O	452	Advanced Linear Programming

Thus we are associating with COURSE a different value (which in this case is a set of tuples obtained from the original set of tuples of Figure 1 by the specified update).

Now queries on this new (instance of the) data base will construct objects using the same operations as before and the same constants as before but the new value of COURSE (and the other relations R of the data base). The query above will now return Advanced Linear Programming obtained by projecting out the value for the Title attribute of the tuple (C&O, 452, Advanced Linear Programming) in (the new instance of) COURSE. Thus, we have to somehow reconcile the fact that operations used to answer queries treat COURSE as a constant in the same way that the empty or universal relations are constants and the fact that COURSE is in fact not fixed over time.

Another possible analysis for this situation as outlined in M&L, M(1) is as follows. In thinking about data base systems, we make a clear logical (and mathematical) distinction between query functions and update functions. Query programs on a data base differ from normal programs on "normal machines" in the important respect that the result of exactly the same query program may be different before and after an update. Forgetting

these update capabilities for the moment, the basic query functions (i.e. the ones built into the so-called query language) together with the valid sets of data on which the query functions could be defined form a system similar to the basic machine functions on a computer (addition, multiplication, tests, etc.) together with the valid sets of data on which these basic operations are defined. That is, the query portion of a data base is essentially a special purpose machine with basic query functions as basic machine functions.

Since an update can change the meaning of a query, it must obviously modify the special purpose query machine in some way. This change is usually accomplished by changing the value of a basic query function at some point in its domain. Thus updates can be viewed as operations which manipulate basic query functions (by changing the definition of the latter in some way).

Our solution to this problem is somewhat more sophisticated than that proposed in (M(1), M&L) and is based on the work presented in (L, L&M(1), L&M(2)) on abstract data types. As in (M(1), M&L), we would like to make the algebra of relations in which an instance of R is used as a set of "constant" relations the object of an "update" algebra. This means that we have to provide some mechanism which will attach a particular value to the names (of relations) in R. Moreover, the update operations in the algebra will change the value attached to a particular relation name. Since all components of our model are abstract data types, we hope to define (axiomatically) an algebra whose operations are query and update operations and whose objects are relations.

We start, as indicated in the introduction, with an abstract data type specification of the algebra of quotient relations defined in (F&K).

This axiomatisation was presented in (T) as a practical example of data type specification and we present here a slightly altered version which we call Q. (See the appendix for a full specification.) The operators of the data type may be described as follows:

create a quotient relation, denoted R_ϕ, from a set of tuples, such that there is only one block, consisting of all the tuples (ϕ is the empty set of attributes).

display a quotient relation by listing the component tuples.

partition a quotient relation by a set of attributes B

$$R_A/B = R_{A\cup B}.$$

departition a quotient relation by a set of attributes B

$$R_A{}^*B = R_{A-B}.$$

union two quotient relations by merging blocks

$$R_A \otimes R'_A = R''_A = \{[t]_A^{R\cup R'} \mid t\in R\cup R'\}.$$

project a quotient relation partitioned on A onto a set of attributs B

$$R_A[B] = R'_A$$

where $A \subseteq B$; each tuple in R' is defined on the attribute subset B only; and every tuple in R corresponds to a tuple in R' such that on the subset B, the values of the tuples are identical.

cross two quotient relations defined on non-intersecting sets of attributes by forming the caresian product of their blocks

$$R_A \otimes R'_B = R''_{A\cup B} = \{[t]_A^R \times [t']_B^{R'} \mid t\in R \text{ and } t'\in R'\}.$$

rename the defining attribute set for the tuples in a quotient relation

$$R_A\{S\} = R'_{S(A)}$$

where S is a mapping from old attribute names to new ones and each tuple in R' corresponds to one in R but defined on the new set of names.

restrict — a quotient relation to contain only blocks satisfying some property expressed by a set comparison operation (e.g. set containment, set equality) involving the values in one (ordered) sequence of attributes of the relation and either another such sequence of attributes or a set of constant tuples. Thus

$$R_A[\underline{X}\theta\underline{Y}] = \{[t]_A^R \mid [t]_A^R \in R_A \text{ and } [t]_A^R[\underline{X}]\theta[t]_A^R[\underline{Y}]\}$$

where $\underline{Y}$ is a sequence of attribute names, or

$$R_A[\underline{X}\theta C] = \{[t]_A^R \mid [t]_A^R \in R_A \text{ and } [t]_A^R[\underline{X}]\theta C\}$$

where C is a set of constant tuples.

As is common with axiomatisations of abstract data types, we take advantage of the existence of canonical forms (unique expressions representing congruence classes of expressions) to simplify our axiomatisation. In (T), a function R: set[attribute] × set[tuple] × set[attribute] → relation is used to define the canonical forms of expressions. (That is, given a partitioned relation with defining set of attributes a, defining set of tuples τ, and partitioning set of attributes a', R(a,τ, a') is the canonical form of the congruence class to which this relation belongs.)

As indicated previously, this model cannot take into account the updates made available in a data base. First of all, the relations R in our particular data base are not even objects of the algebra (although their values as sets of tuples could be constructed). We begin by including each relation name $\bar{R}$ in R as a name of a constant relation (with the same attribute set as $\bar{R}$) in the sort relation. The operations defined above now are defined

on arguments which include these relation names. However, the axioms do not tell us anything about the value of expressions using names in $\mathcal{R}$ and so we cannot evaluate such expressions. We could perform this evaluation if we knew which instantiation of $\mathcal{R}$ we were using. Thus we must somehow encode the instantiation in our algebra.

The way that this is done is suggested by the concept of environment in the mathematical semantics of programs. An environment is a binding of (program) variables to values (plus perhaps some other information). It is this binding of variables to values which was cast in an algebraic setting in (L, L&M(2), M(2)). Let σ be an environment (or data base instance (dbi) as it might more properly be called in this context) and let $\bar{R} \epsilon \mathcal{R}$. We write $\bar{R}(\sigma)$ to denote the value of $\bar{R}$ in the dbi σ. Thus the syntax of $\bar{R}$ is now

$$\bar{R}: \underline{dbi} \rightarrow \underline{relation}$$

where dbi is the set of possible data base instances. We add to the sorts of Q the sort dbi and add the constant dbi Φ with syntax

$$\Phi: \rightarrow \underline{dbi}.$$

This dbi Φ will be used to denote the initial dbi in which the value of $\bar{R} \epsilon \mathcal{R}$ is $\bar{R}(\Phi) = R(a_R, \phi, a'_R)$= (the empty relation with attribute set a and partitioning set a'). In fact we write axioms of this form for each $\bar{R} \epsilon \mathcal{R}$. We must as a consequence add some new operations (corresponding to some of the operations in Q) which take the dbi into account. For example,

$$\text{EDISPLAY: } \underline{relation} \times \underline{dbi} \rightarrow \underline{set}[\underline{tuple}]$$

(as opposed to the original operation

$$\text{DISPLAY: } \underline{relation} \rightarrow \underline{set}[\underline{tuple}]).$$

This is because the relation which is the argument of EDISPLAY may be defined in terms of the names in R and so evaluation is possible only with the knowledge of the values attached to the names in R in the dbi in question. In fact, any of the operations which have arguments of sort relation will now have a corresponding operation taking the extra argument of sort dbi. Figure 3 gives the new syntax for operations which are introduced in this way.

Figure 3

EDISPLAY: relation × dbi → set[tuple]
EPARTITION: relation × set[attribute] × dbi → relation
EDEPARTITION: relation × set[attribute] × dbi → relation
EUNION: relation × relation × dbi → relation
EPROJECT: relation × set[attribute] × dbi → relation
ECROSS: relation × relation × dbi → relation
ERENAME: relation × substitution × dbi → relation
ERESTRICT: relation × sequence[attribute] × comparator × restrictor × dbi → relation

(We have not included the new syntax for the hidden (auxiliary) functions corresponding to the hidden functions in Q.)

Now that we have changed the syntax of the data type, we must reflect this change in the equations which define the behaviour of the type. First of all, we need to define the value of an expression whose result is a relation when the expression is evaluated in a particular dbi σ. If the expression is $\bar{R} \epsilon R$, then we know that its value is $\bar{R}(\sigma)$. Assume also that $\bar{R}(\sigma)$ is in canonical form. That is, $\bar{R}(\sigma) = R(a,t,a')$ for some set of tuples t, attribute set a, and partitioning set a'. If we have some other relation $R(a,t,a')$, then its value in σ is just $R(a,t,a')$. Thus we have the axioms for

EDISPLAY:

$$EDISPLAY(R(a,t,a'),\sigma) = DISPLAY(R(a,t,a'))$$

$$EDISPLAY(\bar{R},\sigma) = DISPLAY(\bar{R}(\sigma)) \text{ for each } \bar{R} \text{ in } \mathcal{R}.$$

Similarly for EPARTITION we have:

$$EPARTITION(R(a,t,a'),a'',\sigma) = PARTITION(R(a,t,a'),a'')$$

$$EPARTITION(\bar{R}(a'',\sigma) = PARTITION(\bar{R}(\sigma),a'') \text{ for each } \bar{R} \text{ in } \mathcal{R}.$$

Thus the strategy for axiomatising "E-operations" is to evaluate $\bar{R}$ in σ and then invoke the corresponding operation on the result of this evaluation. We will not give any more of the new axioms but the interested reader may find them in the appendix.

Now we must explain how new values are attached to the names in $\mathcal{R}$. To do this, we will define some update operations which alter the value attached to $\mathcal{R}$ by changing the dbi σ to a new one σ'. We define three operations whose syntax is presented below. We need a new sort in our algebra called name which has as constants the names in $\mathcal{R}$. Thus we have

$$\bar{R}: \rightarrow \underline{name}$$

for each $\bar{R}$ in $\mathcal{R}$. The update operations are:

$$UPDATE: \underline{name} \times \underline{tuple} \times \underline{tuple} \times \underline{dbi} \rightarrow \underline{dbi}$$

$$INSERT: \underline{name} \times \underline{tuple} \times \underline{dbi} \rightarrow \underline{dbi}$$

$$DELETE: \underline{name} \times \underline{tuple} \times \underline{dbi} \rightarrow \underline{dbi}$$

UPDATE is meant to change the value of some tuple in some $\bar{R} \in \mathcal{R}$ to another value (thus producing a new value of $\bar{R}$). INSERT and DELETE have the obvious meanings. We must now axiomatise these operations and we generally do so by defining the behaviour of queries on the new dbi. Thus we have

$$\bar{R}(UPDATE(\bar{R}',t,t',\sigma)) = \underline{if}\ \bar{R} = = \bar{R}'$$

$$\underline{then}(\underline{if}\ \bar{R}(\sigma) = R(a,\tau,a')\ \underline{then}\ R(a,(\tau-\{t\}) \leftarrow t',a'))$$

$$\underline{else}\ \bar{R}(\sigma).$$

That is, we first of all check to see if $\bar{R}$ is syntactically the same as $\bar{R}'$ (i.e. that $\bar{R}$ and $\bar{R}'$ are the same name). If they are, then the updated relation is being queried and the value of the relation in the new dbi is the result of adding t' to the value of $\bar{R}$ in σ less the tuple t. (We have not worried about the error condition in which $t \notin \bar{R}(\sigma)$. This can be done with a more complicated axiomatisation.) If $\bar{R}$ is not $\bar{R}'$, then the value of $\bar{R}$ in the updated dbi is just the value of $\bar{R}$ in σ. We have assumed that updating a relation named by $\bar{R}'$ does not affect any of the other relations. If this were not so, we would need a more complicated axiom to reflect the change to other relations. The axioms for the other update operations are:

$$\bar{R}(\text{INSERT}(\bar{R}',t,\sigma)) = \underline{\text{if}}\ \bar{R} = = \bar{R}'\ \underline{\text{then}}\ (\underline{\text{if}}\ \bar{R}(\sigma) = R(a,\tau,a')\ \underline{\text{then}}\ R(a,\tau \leftarrow t,a'))\ \underline{\text{else}}\ \bar{R}(\sigma).$$

$$\bar{R}(\text{DELETE}(\bar{R}',t,\sigma)) = \underline{\text{if}}\ \bar{R} = = \bar{R}'\ \underline{\text{then}}\ (\underline{\text{if}}\ \bar{R}(\sigma) = R(a,\tau,a')\ \underline{\text{then}}\ R(a,\tau-\{t\},a'))\ \underline{\text{else}}\ \bar{R}(\sigma).$$

(Again, we have not worried about error conditions and we have assumed that neither operation affects any relation other than the relation being updated.)

So far we have not at all worried about the use of functional dependencies in the definition of relations such as those in $\bar{R}$. We can in fact add axioms to govern the behaviour of updates so that functional dependencies are preserved. Let $t \epsilon \tau$ be a logical function on tuples and sets of tuples (testing for membership of t in τ) and use PIECE(t,a) to

indicate the tuple obtained from t by only taking values of attribures in a. Let $b \rightarrow b'$ be a functional dependency defined on the relation named by $\bar{R}$ in $\mathcal{R}$. Then

$$\begin{aligned}
&\text{UPDATE}(\bar{R},t,t',\sigma) = \\
&\quad \underline{\text{if}}\ \text{PIECE}(t',b) = \text{PIECE}(t'',b) \\
&\qquad \wedge\ t'' \in \text{EDISPLAY}(\bar{R},\sigma) - \{t\} \\
&\qquad \wedge\ \neg\ (\text{PIECE}(t',b') = \text{PIECE}(t'',b')) \\
&\quad \underline{\text{then}}\ \text{error}.
\end{aligned}$$

That is, if $\bar{R}$ already contains a tuple t" (other than t) whose b components are the same as those of t', then the update is allowed only if the b'-attributes of t' and t" are the same. Otherwise there is an error.
For INSERT we have:

$$\begin{aligned}
&\text{INSERT}(\bar{R},t,\sigma) = \\
&\quad \underline{\text{if}}\ \text{PIECE}(t,b) = \text{PIECE}(t',b) \\
&\qquad \wedge\ t' \in \text{EDISPLAY}(\bar{R},\sigma) \\
&\qquad \wedge\ \neg\ (\text{PIECE}(t,b') = \text{PIECE}(t',b')) \\
&\quad \underline{\text{then}}\ \underline{\text{error}}.
\end{aligned}$$

DELETE does not present a problem as far as functional dependencies are concerned.

Thus we can take functional dependencies into account and many of the concepts related to functional dependencies become encoded in the algebraic framework. For example, the closure of a set of functional dependencies becomes in the algebraic framework the set of equations of the above form derivable from the above equations and the usual (algebraic) rules of reasoning. Moreover, if we start with the initial dbi Φ, then the above axioms guarantee that any dbi σ obtained by use of the three update operations will satisfy the functional dependencies.

The example we have (incompletely) presented above was based on the relational model. However, the idea of environments encoding data base instances is independent of the model with which we begin. There is, unfortunately, not enough space here to present the underlying theory in all its generality. This will have to be presented elsewhere. We have also neglected to a large extent the problem of errors in a model and the problems introduced by "don't care" values in tuples. The former can be handled in a straight-forward manner using techniques outlined in (ADJ(1), G). "Don't care" values can be handled by the introduction of special "don't care" constants and appropriate axiomatisation of the behaviour of tuples and relations containing these values.

2. A Semantics for the ANSI/SPARC Architecture

We intend in this section to present a mathematical model to make precise the meaning of the ANSI/SPARC architecture for data base systems. We will focus our attention on the main elements of the architecture as we lack the space to provide a more detailed treatment.

The main components of the ANSI/SPARC architecture are the three schemas (conceptual, internal, and external) and the interfaces between these schemas. As to the details of the schemas and the interfaces, we assume the reader is familiar with their description as in (S, T&L, Y). We have already indicated the mathematical meaning we ascribe to the conceptual schema - it is an abstract data type DB representing both the query and update capabilities of a data base. The fact that DB is an abstract data type (ie., an isomorphism class of algebras) guarantees that our schema is independent of the representation which will be used to support an implementation of the data base system - i.e., we have semantic data independence. This is a great advantage in the design stage of the construction of a data base system for the same reason that the use of abstract data types in programming is useful - decisions about details which do not affect the logical structure of the data base system are postponed to the implementation stage. If the model is truly representation independent, future implementation changes in the system can be accomplished without changing the conceptual model. The advantages of using an axiomatic (and algebraic) specification of the model will become apparent when we discuss the other schemas.

The internal schema DB(I) is an implementation of the conceptual schema DB. Since DB is an abstract data type, we can make precise the meaning of the word implementation. In general terms, we proceed as follows.

Let $T_{\Sigma,E}$ be an abstract data type defined on the symbols Σ and equations E. A representation of $T_{\Sigma,E}$ is a Σ-algebra A_Σ such that A_Σ and $T_{\Sigma,E}$ are isomorphic. (Thus A_Σ is initial in $\underline{Alg}_{\Sigma,E}$.) So $T_{\Sigma,E}$ and A_Σ differ only in the representation of the objects and the concomittant changes in the meanings of the symbols in Σ. But how do we obtain such an algebra A_Σ? Generally, we have available already implemented abstract data types and we define A_Σ in terms of these. For example, if $T_{\Sigma,E}$ arose from the specification of stacks of integers and we had available implementations of integer arrays B_Ω and integers C_Δ, then we could define A_Σ as follows. The objects would be pairs of arrays and integers. The stack operations would be specified as procedures in terms of array and integer operations. To verify that the algebra A_Σ constructed in this way was truly a representation of $T_{\Sigma,E}$ we would have to show that this A_Σ and $T_{\Sigma,E}$ were isomorphic. (For techniques to provide such proofs see (ADJ(1), G, GH&M).) Thus the internal schema DB(I) is an algebra isomorphic to DB and it is constructed from some already defined data types. For our example, we could choose a hierarchical model of data bases (making the kinds of extensions to this model outlined in the previous section to provide for updates) and then provide an implementation of the conceptual schema (defined in terms of partitioned relations) in the obvious way. If this were not suitable, then we could base our implementation on a network model or any other abstract data type(s) suitable for the purpose.

The interface between the conceptual and internal schemas is provided by the formalisation of the representation map assigning to each object in DB its representation in DB(I). In fact the concept of representing one abstract data type in terms of other types can itself be formalised as an abstract data type (L&M(3)) and it is in fact this "representation

data type" which would be used to define the conceptual/internal schema interface.

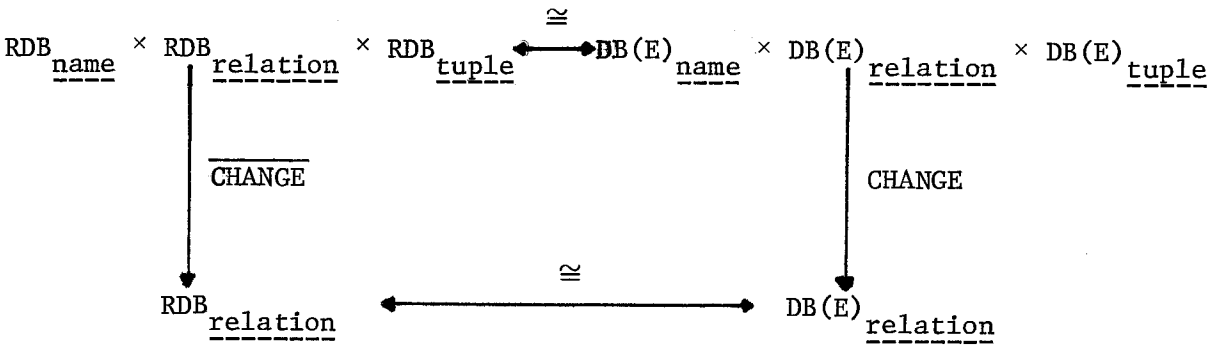
An external schema is the "view" presented to a class of users and in keeping with our philosophy it must be an abstract data type. Whether this type is presented to the user as an axiomatic specification or as some intuitively appealing (set theoretic) model isomorphic to it is immaterial. (In fact, the set theoretic model would provide the more reasonable method.) For example, we could define an external schema DB(E) in the style of Codd (C) (i.e., using unpartitioned relations). We then have to relate this external schema DB(E) to the conceptual schema DB. Clearly we have to define a representation for DB(E) in terms of the type DB. The representation of a relation in DB(E) will of course be a partitioned relation in DB. As to the usual operations of the relational model we have the following implementations (F&K, T) using the operations on partitioned relations defined in the previous section.

Figure 4

Relational Operations	
projection: R[A]	$R_{\phi}(A)$
restriction: R[$\underline{A}\theta\underline{B}$]	$R_{\alpha(R)}[\underline{A}\theta\underline{B}]*\alpha(R)$
join: R[$\underline{A}\theta\underline{B}$]S	$(R_{\alpha(R)} \otimes S_{\alpha(S)})[\underline{A}\theta\underline{B}]*\alpha(R)*\alpha(S)$
division: R[$\underline{A}\div\underline{B}$]S	$((R_{\alpha(R)}*A)\otimes S_{\phi}[B])[\underline{A}\supseteq\underline{B}][\alpha(R)-A]*(\alpha(R)-A)$
cartesian product: R×S	$R_{\phi}\otimes S_{\phi}$
union: R∪S	$R_{\phi}\oplus S_{\phi}$
intersection: R∩S	$(R_{\alpha(R)}\otimes S_{\alpha(S)})[\alpha(R)=\alpha(S)][\alpha(R)]*\alpha(R)$
difference: R-S	$(R_{\alpha(R)}\otimes S_{\phi})[\alpha(R)\neq\alpha(S)][\alpha(R)]*\alpha(R)$

($\alpha(R)$ is the set of defining attributes of the relation R.)

If update operations were provided in DB(E), then they would also have to be modelled in terms of the update operations available in DB. Suppose that update operation CHANGE (taking as arguments a relation name, a relation and a tuple and producing a new value for the named relation) was part of the specification of DB(E). Then the fact that we had defined a representation RDB for DB(E) would guarantee that any update defined by CHANGE in DB(E) can be simulated by a derived update operation in DB (the implementation of CHANGE in RDB in terms of the update operations in DB). Consider the following diagram:



RDB_s (or $DB(E)_s$) is the set of objects of sort s in RDB (or DB(E)). Thus $DB(E)_{relation}$ is the set of (unpartitioned) relations and $RDB_{relation}$ is the representation of these relations using DB. $\cong$ is the isomorphism between RDB and DB(E) and $\overline{CHANGE}$ is the representation in RDB of the operation CHANGE in DB(E). (So $\overline{CHANGE}$ is a derived operation of DB.) We claim that this diagram commutes. That is, if an update in DB(E) caused by CHANGE results in a value in $DB(E)_{relation}$ and this value is then mapped by $\cong$ into $RDB_{relation}$ then we should get the same result by mapping the original arguments of CHANGE into the corresponding values in RDB and then applying $\overline{CHANGE}$.

If we have more than one external schema, say DB(E) and DB(E'), then each can be represented separately in terms of DB. However, we now have a

consistency problem. Namely, an update in DB(E), for example, must not cause anything untoward to happen to the schema DB(E'). This can happen if an update in DB(E) causes a change in a relation $\bar{R}$ in DB (used in the representation of both DB(E) and DB(E')) such that the new value of this relation $\bar{R}'$ is "inconsistent" with the axiomatisation of DB(E'). The new value of the relation in DB(E') can be "inconsistent" with the axiomatisation if the new value of any relation which uses $\bar{R}$ in its representation cannot be obtained from the old value using only available updates in DB(E').

A sufficient condition for proving the consistency of external schemas DB(E) and DB(E') can be obtained by showing that either one can be used to represent the other. That is, we can define a representation RE'DB(E) for DB(E) in terms of DB(E') and a representation REDB(E') for DB(E') in terms of DB(E). (This is quite a strong condition and might possibly be weakened.)

The interface between the external schema(s) and the conceptual schema is again the "representation data type" developed in the process of representing the external schema(s) by the conceptual schema. The interface between the external schema(s) and the internal schema is the representation defined by combining the external/conceptual and conceptual/internal interfaces. That is, the fact that we can represent the data type defined by the external schema(s) in terms of the data type defined by the conceptual schema and we can represent the latter by the data type(s) used in the internal schema guarantees that we can represent the external schema(s) in the internal schema.

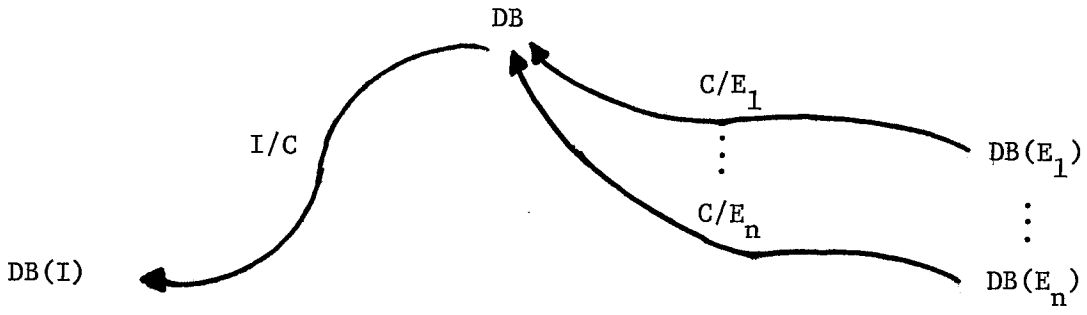
We should point out, however, an extremely important difference between the external/conceptual and conceptual/internal interfaces. In the latter case, we are allowed to pick any data types we need (and can implement)

to define a representation for the conceptual schema. In the former case, our implementation language is fixed - we must represent the external schema(s) only in terms of the types defined in the conceptual schema.

3. Conclusions

Beginning with the concept of abstract data type, we have illustrated a method which allows us to model data base systems in such a way that both queries and updates are an integral part of the whole. This method is general and can be used in conjunction with any of the existing models for data bases or any other model which seems suitable.

Once it has been established that data base systems are examples of abstract data types, it is then possible to assign a precise mathematical semantics for the ANSI/SPARC architecture. This architecture may be described diagrammatically as follows:



Here DB is the abstract data type(s) used to model the data base system and represents the conceptual schema. DB(I) is (are) the abstract data type(s) determined by the machine architecture, the programming language and the characteristics of the conceptual schema used to provide a representation for the conceptual schema DB. $DB(E_i)$ is the i'th external schema - again specified as an abstract data type. The notation A⇝B means that the data type(s) B is (are) used in the representation of the data type A. I/C and C/E_i are the internal/conceptual and conceptual/external schema interfaces described in the previous section. The external/internal schema interface is provided by combining the above. That is $DB(I) \leftsquigarrow DB \leftsquigarrow DB(E_i)$ gives us $DB(I) \leftsquigarrow DB(E_i)$.

Finally, we would like to conclude with a remark concerning the nature of the programming languages to be used in data base systems modelled on this architecture. As indicated before, the query portion of our models can be used only to evaluate expressions (denoting relations, tuples, or components of tuples) without permanently affecting the "constant" relations of the data base. The update operations do leave a permanent mark by changing the values of these constant relations. The analogous comparison for programming languages is between declarative and imperative languages. Thus a language appropriate for data base applications should consist of two parts: a declarative query sublanguage and an imperative update language.

Acknowledgements: The author would like to thank M.R. Levy and F.W. Tompa for their helpful suggestions and interest in this work.

APPENDIX A

type

Q(partitioned relations)

syntax

R: set[attribute] × set[tuple] × set[attribute] → relation
CREATE: set[tuple] → relation
HOMOGENEOUS: set[tuple] → logical
DISPLAY: relation → set[tuple]
ATTRIBS: relation → set[attribute]
PARTS: relation → set[attribute]
PARTITION: relation × set[attribute] → relation
DEPARTITION: relation × set[attribute] → relation
UNION: relation × relation → relation
PROJECT: relation × set[attribute] → relation
CROSS: relation × relation → relation
RENAME: relation × substitution → relation
RESTRICT: relation × sequence[attribute] × comparator
× restrictor → relation
BLOCK: set[tuple] × set[attribute] × tuple → set[tuple]
OK: set[tuple] × sequence[attribute] × comparator
× sequence[attribute] → logical
VALUES: set[tuple] × sequence[attribute] → set[sequence[value]]
MIN: set[tuple] × sequence[attribute] → sequence[value]
MAX: set[tuple] × sequence[attribute] → sequence[value]

variables

a,a',a",a"': set[attribute]
τ: set[tuple]
r: relation
t,t': tuple
A,A': sequence[attribute]
c: comparator

semantics

```
CREATE(φ) = R(φ,φ,φ)
CREATE(τ←t) = R(COLUMNS(t),τ←t,φ)
   if HOMOGENEOUS(τ←t)
HOMOGENEOUS(φ) = true
HOMOGENEOUS(φ←t) = true
HOMOGENEOUS(τ←t←t') = (COLUMNS(t) ≡ COLUMNS(t')) ∧ HOMOGENEOUS(τ←t)
DISPLAY(R(a,τ,a')) = τ
ATTRIBUTE(R(a,τ,a')) = a
PARTS(R(a,τ,a')) = a'
PARTITION(R(a,τ,a'),a'') = R(a,τ,a' ∪ a'')
   if a'' ⊆ a
DEPARTITION(R(a,τ,a'),a'') = R(a,τ,a' - a'')
   if a'' ⊆ a
UNION(R(a,τ,a'),R(a,τ',a')) = R(a,τ ∪ τ',a')
PROJECT(R(a,φ,a'),a'') = R(a'',φ,a')
   if a'' ⊆ a ∧ a' ⊆ a''
PROJECT(R(a,τ←t,a'),a'') = R(a'',DISPLAY(PROJECT(R(a,τ,a'),a'')
   ← PIECE(t,a''),a')
   if a'' ⊆ a ∧ a' ⊆ a''
CROSS(R(a,τ,a'),R(a'',φ,a''')) = R(a ∪ a'',τ,a' ∪ a''')
   if a ∩ a'' = φ
CROSS(R(a,τ,a'),R(a'',τ←t,a''')) = R(a ∪ a'',DISPLAY(CROSS(R(a,τ,a'),
   R(a'',τ',a'''))) ∪ COMPOSE(τ,t),a' ∪ a''')
   if a ∩ a'' = φ
RENAME(R(a,φ,a'),s) = R(MAP(s,a),φ,MAP(s,a'))
   if a = COLMS(s)
RENAME(R(a,τ←t,a'),s) = R(MAP(s,a),DISPLAY(RENAME(R(a,τ,a'),s))
   ← ALIAS(t,s),MAP(s,a'))
   if a = COLMS(s)
RESTRICT(R(a,φ,a'),A,c,NAMES(A')) = R(a,φ,a')
   if MEMBERS(A) ⊆ a ∧ MEMBERS(A') ⊆ a ∧ COMPARABLE(c,A,A',R(a,φ,a'))
RESTRICT(R(a,τ←t,a'),A,c,NAMES(A')) =
   if OK(BLOCK(τ,a',t),A,c,A')
      then R(a,DISPLAY(RESTRICT(R(a,τ-BLOCK(τ,a',t),a'),A,c,A'))
                                                  ∪ BLOCK(τ,a',t),a')
      else RESTRICT(R(a,τ-BLOCK(τ,a',t),a'),A,c,A')
   if MEMBERS(A) ⊆ a ∧ MEMBERS(A') ⊆ a ∧ COMPARABLE(c,A,A',R(a,τ←t,a'))
RESTRICT(r,A,c,CONST(t)) = RESTRICT(r ⊗ CREATE(t),A,c,NAMES(SEQ(ATTRIBS
   (CREATE(t)))))[ATTRIBS(r)]
   if MEMBERS(A) ⊆ ATTRIBS(r) ∧ MEMBERS(A') ⊆ ATTRIBS(r) ∧ COMPARABLE
      (c,A,A',r)
BLOCK(φ,a,t) = t
BLOCK(τ←t,a,t') =
   if MATCH(t,t',a)
      then BLOCK(τ,a,t') ← t
      else BLOCK(τ,a,t')
OK(τ,A,/c,A') = ¬ OK(τ,A,c,A')
OK(τ,A,c,A') = case c of
```

```
'=':  VALUES(τ,A) ≡ VALUES(τ,A')
'≈':  VALUES(τ,A) ∩ VALUES(τ,A') ≠ ϕ
'⊇':  VALUES(τ,A) ⊇ VALUES(τ,A')
'⊃':  VALUES(τ,A) ⊇ VALUES(τ,A') ∧¬(VALUES(τ,a) ≡ VALUES(τ,a'))
'⊆':  OK(τ,A',⊇,A)
'⊂':  OK(τ,A',⊃,A)
'•≤': (MAX(τ,A) < MAX(τ,A') ∨ (MAX(τ,A) ≡ MAX(τ,A'))
'•<': MAX(τ,A) < MAX(τ,A')
'≤':  (MIN(τ,A) < MAX(τ,A')) ∨ (MIN(τ,A) ≡ MAX(τ,A')
'<':  MIN(τ,A) < MAX(τ,A')
'≤•': (MIN(τ,A) < MIN(τ,A')) ∨ (MIN(τ,A) ≡ MIN(τ,A'))
'<•': MIN(τ,A) < MIN(τ,A')
'•≤•':  (MAX(τ,A) < MIN(τ,A')) ∨ (MAX(τ,A) ≡ MIN(τ,A')
'•<•':  MAX(τ,A) < MIN(τ,A')
'≥•': OK(τ,A',•≤,A)
'>•': OK(τ,A',•<,A)
'≥':  OK(τ,A',≤,A)
'>':  OK(τ,A',<,A)
'•≥': OK(τ,A',≤•,A)
'•>': OK(τ,A',<•,A)
'•≥•':  OK(τ,A',•≤•,A)
'•>•':  OK(τ,A',•<•,A)
VALUES(ϕ,A) = ϕ
VALUES(τ←t,A) = VALUES(τ,A) ← SEQ(PIECE(t,MEMBER(A))A)
MIN(ϕ←t,A) = t
MIN(τ←t←t',A) =
   if BEFORE(t,t',A)
      then MIN(τ←t,A)
      else MIN(τ←t',A)
MAX(ϕ←t,A) = t
MAX(τ←t←t',A) =
   if BEFORE(t,t',A)
      then MAX(τ←t',A)
      else MAX(τ←t,A)
```

We have used in this definition some further types: set[value] (a parameterised type), restrictor (used in the RESTRICT function), tuple (maps from sets of attributes to values), substitution (used in the RENAME function), and sequence[value] (used in the RESTRICT function). These will not be defined here but the reader is referred to (T) for details. (The operations which appear above and are not defined above come from some of these types.)

APPENDIX B

The Extended Type Q

The new functions are those of Figure 3 together with:

EATTRIBS: relation × dbi → set[attribute]

EPARTS: relation × dbi → set[attribute]

The additional axioms may be obtained by anology with the rules for EDISPLAY and EPARTITION presented in the text. There are two rules for each E-function Efn. The first reduces Efn to the corresponding original function n when the argument of sort relation is $R(a,\tau,a')$. Otherwise Efn applied to $\bar{R} \in$ name (and other possible arguments) and $\sigma \in$ dbi is fn applied to $\bar{R}(\sigma)$ (and the other possible arguments).

Bibliography

A: J.R. Abrial - Data Semantics, "Data Base Management", eds. J.W. Klimbie and K.L. Koffeman, North-Holland, 1974.

AD&L: M. Adiba, C. Delobel, M. Leonard - A Unified Approach for Modelling Data in Logical Data Base Design, "Modelling in Data Base Management Systems", ed. Nijssen, North-Holland, 1976.

ADJ(1): J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright - An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, "Current Trends in Programming Methodology, Volume IV", ed. R.T. Yeh, Prentice Hall, 1978.

ADJ(2): J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright - Data Type Specification: Parameterization and the Power of Specification Techniques, Proceedings of the 10th SIGACT Symposium on the Theory of Computing, 1978.

B&S: M.L. Brodie, J. Schmidt - What is the Use of Abstract Data Types in Data Bases? Proceedings of the 4th International Conference on Very Large Data Bases, ed. S. Bing Yao, 1978.

Ch: P.P.S. Chen - The Entity Relationship Model - Toward a Unified View of Data, ACM Transactions on Data Base Systems, Vol. 1, No. 1, 1976.

C: E.F. Codd - A Relational Model for Shared Data Banks, CACM, Vol. 13, No. 6, 1970.

Co: P.M. Cohn - "Universal Algebra", Harper and Row, 1965.

DD&H: J. Dahl, E.W. Dijkstra, C.A.R. Hoare - "Structured Programming", Academic Press, 1972.

D: C.J. Date - "An Introduction to Database Systems", Addison Wesley, 1976.

EK&W: H.Ehrig, H-J. Kreowski, H. Weber - Algebraic Specification Schemes for Database Systems, Proceeding of the 4th International Conference on Very Large Data Bases, ed. S. Bing Yao, 1978.

F&K: A.L. Furtado, L. Kerschberg - An Algebra of Quotient Relations, Proceeding of the SIGMOD Conference, 1977.

G: J.A. Goguen - Abstract Errors for Abstract Data Types, Semantics and Theory of Computation Report #6, UCLA, 1977.

G&G: C.C. Gottlieb, L.R. Gottleib - "Data Types and Structures", Prentice-Hall, 1978.

Gu: J.V. Guttag - Abstract Data Types and the Development of Data Structures, CACM, Vol. 20, No. 6, 1977.

GH&M: J.V. Guttag, E. Horowitz, D.R. Musser - Some Extensions to Algebraic Specifications, Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, No. 3, 1977.

L: M.R. Lévy - Verification of Programs with Data Referencing, Proceedings of the 3e Colloque International sur la Programmation, Dunod, 1978.

L&M(1): M.R. Levy, T.S.E. Maibaum - Continuous Data Types, in preparation.

L&M(2): M.R. Levy, T.S.E. Maibaum - Data Types with Sharing and Circularity, in preparation.

L&M(3): M.R. Levy, T.S.E. Maibaum - in preparation.

L&Z(1): B.H. Liskov, S.N. Zilles - Programming with Abstract Data Types, Proceedings of the ACM Symposium on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, 1974.

L&Z(2): B.H. Liskov, S.N. Zilles - Specification Techniques for Data Abstraction, IEEE Transactions on Software Engineering SE-1, Vol. 1, 1975.

M(1): T.S.E. Maibaum - Mathematical Semantics and a Model for Data Bases, Proceedings of IFIP Congress 77, ed. B. Gilchrist, North-Holland, 1977.

M(2): T.S.E. Maibaum - The Semantics of Shared Variables in Parallel Processing, submitted for publication.

M&L: T.S.E. Maibaum, C.J. Lucena - Higher Order Data Types, to appear in the International Journal of Computing and Information Sciences.

P: C. Pair - Formalizations of the Notions of Data, Information, and Information Structure, "Data Base Management", eds. J.W. Klimbie and K.L. Koffeman, North-Holland, 1974.

S&S: J.M. Smith, D.C.P. Smith - Data Base Abstraction, Proceedings of the ACM Conference on Data Abstraction, Definition, and Structure, 1976.

S: SPARC Interim Report, ANSI document no. 7514TS01, 1975.

T: F.W. Tompa - A Practical Example of the Specification of Abstract Data Types, submitted for publication.

T&K: D.C. Tsichritzis, A. Klug - The ANSI/SPARC DBMS Framework, Tech-Note 12, Computer Systems Research Group, University of Toronto, 1977.

T&L: D.C. Tsichritzis, F.H. Lochovsky - "Data Base Management Systems", Academic Press, 1977.

W: H. Weber - A Software Engineering View of Data Base Systems, Proceedings of the 4th International Conference on Very Large Data Bases, ed. S. Bing Yao, 1978.

Z: S.N. Zilles - Abstract Specifications for Data Types, IBM Research Report, San Jose, Ca., 1975.