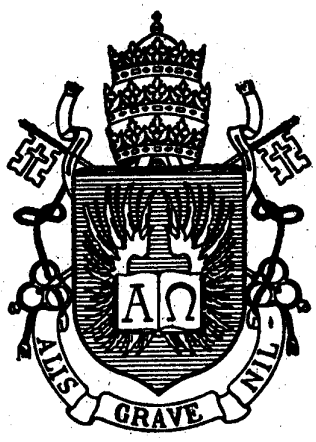

PUC

Series: Monografias em Ciência da Computação
Nº 5/78

A PRACTICAL EXAMPLE OF THE SPECIFICATION OF ABSTRACT DATA TYPES

by

Frank Wm. Tompa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciência da Computação
Nº 5/78
Series Editor: Michael F. Challis January, 1978

A PRACTICAL EXAMPLE OF THE SPECIFICATION OF ABSTRACT DATA TYPES *

by

Frank Wm. Tompa**

* This work has been partially sponsored by FINEP
**Dept. of Computer Science, University of Waterloo
Waterloo, Ontario, N2L 3G1 Canada

ABSTRACT

The algebra of quotient relations,a relationally complete set of operations for data base applications, is formally defined in terms of the algebraic specification technique. The process of constructing an algebraic specification for data type is described in order that future formal definitions are more easely derived. Several improvements to current algebraic presentation techniques are also introduced.

KEYWORDS

abstract data types, algebraic specifications, data independence, data structure formalism, quotient relations, relational model, data abstraction, data bases, conceptual schema.

1. Introduction

Two aspects of data abstraction for data base design can be distinguished: the modelling of an enterprise's global, evolving data and the modelling of a program's local, transient data [Hammer 76, Ledgard 77]. In fact, the separation of the conceptual schema from the external schemas in an Ansi/Sparc data base architecture is made in recognition of these two aspects [Tsichritzis 77].

The relational model (originally, [Codd 70]), is one candidate that has been proposed for describing a conceptual schema [Hall 76, Adiba 76]. Using this model, one would describe the data, meaningful operations and constraints imposed by a given enterprise in terms of n-ary relations and compound operations defined on those relations. Each application administrator would then define an external schema in terms of the conceptual schema's compound operations.

Recently an algebra of quotient relations has been proposed for manipulating partitioned n-ary relations [Furtado 77]. This algebra has been shown to be relationally complete [Codd 72], and furthermore the partitioning of relations provides a convenient set-oriented framework for query processing. Therefore this algebra is a suitable candidate for specifying a conceptual schema.

Before an application administrator can use any of the conceptual schema's compound operations and before the mappings relating the conceptual schema to the rest of the system can be implemented, the objects and operations of that schema must be clearly understood. In particular, if the algebra of quotient relations is used to define certain compound operations, each basic operation in that algebra must have a concise, complete, and

consistent definition of its semantics when it is used in conjunction with the other basic operations.

The notion of an abstract data type [e.g. Liskov 74] is usually viewed in the realm of programming languages rather than in data bases. It can thus easily be seen that the notion fits well into the second aspect of data abstraction mentioned above: the modelling of a program's transient data. In addition, the algebraic specification technique for abstract data types [Liskov 75, Zilles 75, Guttag 77a] has exactly the properties required for the definition of the conceptual schema language, and therefore abstract data types are also useful as part of the first aspect: modelling an enterprise's global data.

This paper contains an algebraic specification for quotient relations as an abstract data type. Section 2 contains an informal view of quotient relations and the operations defined on them. Section 3 serves as an introduction to the algebraic specification technique used by Zilles and by Guttag. Finally, section 4 describes the algebraic specification of the algebra of quotient relations, the complete formal specification being in the appendix. Given this formal definition of quotient relations, designers of a data base system founded on such relations have a common basis for understanding the meanings of their objects and compound operations. The definitions thus serve as a foundation for reliable, correct data base systems.

2. The algebra of quotient relations

An informal description of quotient relations and the operations defined on them was presented recently [Furtado 77]. This section merely serves as a brief overview of the algebra in order that a reader can more easily understand the formal definitions to follow; for further motivation and examples of the use of quotient relations, the reader is referred to the original article. A few minor modifications have been made to the algebra of quotient relations since it was originally introduced; these are indicated herein.

Traditionally a relation is defined as a subset of the cartesian product of a set of attribute domains, and a tuple as one "row" within that relation. In order to avoid any dependence on an arbitrary ordering of the domains an alternative definition is given here.

A tuple defined on a set of attributes is a function from the set of attribute names to a (universal) set of values. (The subset of values that may be associated with a particular attribute are termed a domain.) For example, the traditional notation for tuples, e.g. t = ("JOE",25) defined on "name" and "age", will be replaced by t("name") = "JOE" and t("age") = 25. A set of homogeneous tuples (i.e. those defined on the same attribute set) form a relation. Such a relation can be partitioned by A, a set of attributes, by forming blocks of tuples that all have equal values for the attributes in A:

$$[t]_A^R = \{t' \mid t' \in R \text{ and for all } a \in A,\ t'(a) = t(a)\}$$

where R is a relation and t'(a) is the value for a in the tuple t'. A quotient relation R_A is the set of all such blocks formed from R:

$$R_A = \{[t]_A^R \mid t \in R\}$$

Several operations are defined on quotient relations as follows (more complete definitions will be given in section 4 and the appendix):

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

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

project a quotient relation partitioned on A onto a set of attributes 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 cartesian 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

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

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

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

where Y is a sequence of attribute names, or

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

where C is a set of constant tuples.

To illustrate the use of these operators, a table from [Furtado 77] is repeated here as Figure 1. The operators of Codd's relational algebra [Codd 72] are expressed in terms of operations on quotient relations (where the notation $\alpha(R)$ is used to indicate the defining set of attributes for the tuples in R.

relational algebra	algebra of quotient relations
projection: $R[A]$	$R_{\phi}[A]$
restriction: $R[A\theta B]$	$R_{\alpha}(R)[A\theta B] * \alpha(R)$
join: $R[A\theta B]S$	$(R_{\alpha(R)} \otimes S_{\alpha(S)})\ [A\theta B] * \alpha(R) * \alpha(S)$
division: $R[A \div B]S$	$((R_{\alpha(R)} * A) \otimes S_{\phi}[B])[A \supseteq B][\alpha(R)-A] * (\alpha(R)-A)$
cartesian product: $R \times S$	$R_{\phi} \otimes S_{\phi}$
union: $R \cup S$	$R_{\phi} \oplus S_{\phi}$
intersection: $R \cap S$	$(R_{\alpha(R)} \otimes S_{\alpha(S)})[\alpha(R) = \alpha(S)][\alpha(R)] * \alpha(R)$
difference: $R-S$	$(R_{\alpha(R)} \otimes S_{\phi})[\alpha(R) \neq \alpha(S)][\alpha(R)] * \alpha(R)$

FIGURE 1

Encoding Codd's relational algebra

3. The algebraic specification of abstract data types

Liskov and Zilles separate specification techniques for abstract data types into several classes, one of which is termed algebraic [Liskov 75]. This technique has been described in detail by Zilles and by Guttag. In this section one presentation method for algebraic specifications will be described, based mostly on Guttag's approach [Guttag 77b].

An algebraic specification of a data type is composed of four sections: the type heading, the syntax (interface), the constraints (restrictions), and the equivalences (axioms). The specification technique is derived from the discipline of many-sorted algebras; the equations that comprise the equivalences section generate an algebraic object that defines the (representation-independent) semantics of the data type [Levy 77, Goguen 75]. The definition of a set in Figure 2 will be used as an illustrative example throughout this section.

The type heading consists of the name of the type being defined and optionally one or more parameters to the type. In the example, the type being defined is set, and the definition is based on a single parameter value which can represent any type. Thus the definition is, in fact, not of one type but rather of a family of related types that share similar operators. The example thus serves as a shorthand definition for the types set of integers, set of reals, set of relations, set of set of relations, etc.

The second section consists of a set of functional descriptions of the operators for the type being defined. Each description includes the name of an operator, the names of the types whose cross-product is the operator's domain, and the name of the type that is the operator's range; the operator is a function from the cross-product of certain (input) types

type

set[value]

syntax

```
EMPTY:                  → set      {φ}
INSERT:  set × value → set         {1←2}
CONT:  value × set → logical       {1∈2}
SUBSET:  set × set → logical       {1⊇2}
EQUIV:  set × set → logical        {1≡2}
ADD:  set × set → set              {1∪2}
SUBT:  set × set → set             {1-2}
INT:  set × set → set              {1∩2}
ONEOF:      set → value
```

constraints

ONEOF(S) >> S ≠ φ

equivalences

```
CONT(v,φ) = FALSE
CONT(v,s←v') if v ≡ v'
                then TRUE
                else CONT(v,s)
SUBSET(s,φ) = TRUE
SUBSET(s,s'←v) = if v∈s
                    then SUBSET(s,s')
                    else FALSE
EQUIV(s,s') = s⊇s' ∧ s' ⊇ s
ADD(s,φ) = s
ADD(s,s'←v) = ADD(s←v,s')
SUBT(φ,s) = φ
SUBT(s←v,s') = if v∈s'
                  then SUBT(s,s')
                  else SUBT(s,s')←v
INT(s,s') = s-(s-s')
ONEOF(s←v) = v
```

FIGURE 2

An algebraic specification for set

into one (output) type. Each operator definition may be followed by an alternative syntactic form indicated by illustrating the positions of the operands (numbered from left to right) in relation to the symbolic operator. For example, the operator EMPTY, alternatively written ϕ whenever convenient, is a nullary operator, that is, it is constant [Grätzer 68], yielding an object of type set; and the operator INSERT, alternatively written in infix form using $\leftarrow$, is a function from set cross value into set, where value plays a parametric role.

Because the definition for set actually defines a family of types, the operators in the example each represent a family of operators, one appropriate for each type. That is, the set of operators corresponding to one realization for the parameters is distinct from the set corresponding to another. In the example, the type set [attribute] has operators $\text{EMPTY}_{attribute}$, $\text{INSERT}_{attribute}$ (alternatively, $\phi_{attribute}$, $\leftarrow_{attribute}$), etc. whereas the type set [tuple] has operators EMPTY_{tuple}, INSERT_{tuple}, (alternatively ϕ_{tuple}, $\leftarrow_{tuple}$), etc; $\text{INSERT}_{attribute}$ accepts a set of attributes and an attribute and returns a set of attributes, whereas INSERT_{tuple} accepts a set of tuples and a tuple and returns a set of tuples. For the remainder of the paper, the subscripts for the operators will be omitted whenever no confusion will result, thus denoting a particular operator by its family's representative.

The labelling of the second section of a type definition by the keyword syntax is intended to remind a reader that the functional expressions implicitly describe the set of all well-formed formulas. For

example, $EMPTY_{attribute}$ is a syntactically well-formed expression of type set [attribute], and $CONT_{tuple}$ (t, ϕ_{tuple}*$_{tuple}$t') is a syntactically well-formed expression of type logical whenever t and t' are tuples. Notice that for any expression t, neither of the following expressions are syntactically well-formed according to this example: ADD_{tuple}(ϕ_{tuple}, $ONEOF_{tuple}$(t)) -- ADD_{tuple} must take two sets as parameters -- and $INSERT_{attribute}$(ϕ_{tuple}, $ONEOF_{tuple}$(t))-- $INSERT_{attribute}$ is defined for set [attribute] not set [tuple].

The third section of a type definition is a set of constraints that indicate necessary conditions for a syntactically well-formed expression to be semantically well-formed. The constraint in the example is to be read: "the use of ONEOF(S) requires that S≠ϕ". Each constraint denotes a requirement which is interpreted as follows: any formula involving a (sub) expression of the form preceding the separator symbol >> is valid only if the consequent logical expression is well-formed syntactically and semantically and is equivalent to TRUE (a nullary operator defined on the type logical). For the example of set, because ONEOF is intended to select an element of a set, ONEOF(ϕ) is not semantically well-formed. This section of contraints provides implementers of the type with hints for error-checking by listing explicitly some necessary conditions for meaningful expressions.

Finally, the section of equivalences defines a partitioning of all (syntactically and semantically) well-formed expressions of one type into equivalence classes, each class being interpreted as a distinct value for that type. In the example, applications of the equations indicate that,

whenever all the constituent expressions are well-formed, ADD(S,ϕ) = ϕ, ADD(S,ϕ←v) = S←v, and ADD(S,(ϕ←v)←v') = (S←v')←v. The second equation

CONT(v,S←v') = <u>if</u> v ≡ v'
<u>then</u> TRUE
<u>else</u> CONT(v,s)

uses three operators that are assumed in the example to be defined elsewhere:

IF: logical × value × value → value {<u>if</u> 1 <u>then</u> 2 <u>else</u> 3 }
TRUE: → logical
EQ: value × value → logical { 1 ≡ 2 }

(the first using a special alternative syntax [Goguen 75]). The equations, although theoretically symmetric in nature, can often in practice be interpreted as simplifying or substitution rules.

The complete definition of a type conveys to a reader (or implementer) exactly the information that characterizes the type <u>and no more</u>. The next section will discuss the creation of such type definitions through an extended example concerning the algebra of quotient relations.

4. Constructing an algebraic specification for quotient relations

The task of specifying a data type formally seems to most programmers very formidable. In fact, after very little practice, it is comparable to writing and debugging a program of similar length. As in programming, there is likely to be some requirement for insight but mostly the requirement for discipline and style.

The first step is to write down the first two sections of the specification. For quotient relations the type heading is

type

relation

and the initial set of functional descriptions is

syntax

CREATE: set [tuple] → relation	
DISPLAY: relation → set[tuple]	
PARTITION: relation × set[attribute] → relation .	{1/2}
DEPARTITION: relation × set[attribute] → relation	{1*2}
UNION: relation × relation → relation	{1 ⊕ 2}
PROJECT: relation × set[attribute] → relation	{1[2]}
CROSS: relation × relation → relation	{1 ⊗ 2}
RENAME: relation × substitution → relation	{1{2}}
RESTRICT: relation × sequence[attribute] × comparator × restrictor → relation	{1[2 3 4]}

where tuple, attribute, set[value], sequence[value], substitution, comparator, and restrictor are other data types.

It should be noted that all the names used are arbitrary, e.g. the type could equally well be called quotient_relation or QR. Some further (auxilliary) operators may be required during the development of the equivalences, but at least these nine must be present.

When attempting to construct the equivalences for an algebraic specification of a data type, one great pitfall is to use an operational instead of a denotational approach. "In an operational specification

instead of trying to describe the properties of the abstract data type, one gives a recipe for constructing it" [Guttag 77a, p. 397]. Supposedly the algebraic specification technique is denotational in nature, but, especially for programmers, it is very easy to misuse the tool. For example, one could describe a stack algebraically by defining it in terms of operations on a contiguous array and its index set, exactly the representational detail one is attempting to avoid. (In fact, Guttag demonstrates how this introduction of representational information may be used to advantage in refining data type descriptions [Guttag 77a], but such refinement should take place only after the representation-independent specification has been completed.)

For the specification of quotient relations, this pitfall manifests itself throughout the following approach. A relation is defined as a set of blocks, and each relational operation is described in terms of operators on the blocks. Next a block is defined as a set of typles, and the block operators are described in terms of operations on individual tuples. Thus, for example, the partition operation for relations is defined to produce the union of the sets of blocks that result from splitting each component block into sub-blocks according to the values of the partitioning attributes; the split operation for blocks is defined to produce a set of blocks by examining each tuple in turn and classifying it according to the values of the partitioning attributes. This approach is procedural in nature, paralleling the programming language concept of nested iteration loops. The result is an overspecification of the type. "The introduction of extraneous detail [through such overspecification] places unnecessary constraints on the choice of an implementation and may potentially eliminate the best solutions to the problem" [Guttag 77a, p. 398]. In fact, the only

behavioural characteristics of partitioning are those that affect later operations such as restriction.

Given the nine operators to be defined, how can a complete and consistent behavioural specification be constructed? The first observation, made by Guttag in his development of the notion of "sufficiency-completeness" [Guttag 75], is that the only observable characteristic of a data type specification is its impact on elements of other data types. In particular, the behaviour of quotient relations is only visible to the outside when a value of some other data type is produced; for example, the operators PARTITION, PROJECT, and UNION can behave in an arbitrary way as long as DISPLAY has the expected results when applied to a relation resulting from their application. This line of reasoning leads a designer to attempt to construct equations that are all of the form

DISPLAY (X) = y

where X is any relation. However, one soon realises that X may be the result of any one of a very large set of potential series of operations, the last operation of which is any of the eight operators other than DISPLAY.

One is therefore led to consider another heuristic for deriving the equivalences: expressing the behaviour for pairs of operators. Recall that the algebraic specification technique is one which presents the inter-relationships between operators. It is therefore reasonable to define, for each operator taking a relation as input, the results when that relation was produced by any of the operators that yield relation as an output. Thus one has equations of the form:

```
DISPLAY(CREATE(t)) = ...
DISPLAY(PARTITION(r,a)) = ...
        ⋮
DISPLAY(RESTRICT(r,a,c,a')) = ...
PARTITION(CREATE(t),a) = ...
PARTITION(PARTITION(r,a),a') = ...
        ⋮
RESTRICT(RENAME(r,s),a,c,a') = ...
RESTRICT(RESTRICT(r,a,c,a'),a'',c',a''') = ...
```

For quotient relations, eight operators yield relations as output and of the eight operators that take relations as input, two require a pair of relations. Thus there would need to be at least $(8\times8) \times 2 + (8) \times 6 = 176$ equations in the type definition (and this includes neither auxilliary operations nor equations involving more than two operators at a time)!

The best way to reduce the number of equations is to limit the number of possible input forms that a relation can assume. Thus the proper solution to creating the equivalences is to devise a canonical form for expressions in the type, if possible, and to write each equation as a transformation on the canonical form. Looking back at the example of set[value], the canonical form for a set expression is

$$\phi \leftarrow v_1 \leftarrow v_2 \leftarrow v_3 \leftarrow \ldots \leftarrow v_{n-1} \leftarrow v_n$$

where $0 \leq n$. Every operator that takes set as input assumes this form as an operand for the left hand side of each relevant equation, and every operator that yields set as output always uses this form for the right hand side of each relevant equation. The only exception to the latter is INT, which can instead be thought of as a shorthand notation for an expression involving SUBT. It is important to realize that this is a canonical form for set expressions, that is, a standard ordering of the operators, and does not imply a unique expression for each value of type set.

For quotient relations, the key observation is that every quotient relation is equal to some other relation which is created from a (possibly empty) set of tuples and then partitioned once by some set of attributes. Thus a canonical form is a triple

<set of attributes, set of tuples, set of attributes>

representing the defining set of attributes for the relation, the current constituents, and the current partitioning of the relation. This is encoded for the algebraic specification as an auxilliary ternary operator:

R: set[attribute] × set[tuple] × set[attribute] → relation

For non-empty sets of tuples, R(a,t,a') is equivalent to PARTITION(CREATE(t),a'); by recording the defining set of attributes, a, the definition of R also allows for the complete specification of relations that happen to contain no tuples.

The operator CREATE thus must produce an unpartitioned relation in canonical form,given a set of tuples.

CREATE(ϕ) = R(ϕ,ϕ,ϕ)
CREATE(τ←t) = R(COLUMNS(t),τ←t,ϕ)

where τ is a set of tuples and COLUMNS returns the defining set of attributes for a single tuple as specified for the data type tuple (see the appendix). Notice that this definition depends on the canonical form for a set.

Because a relation is defined only for a homogeneous set of tuples, a heterogeneous set should be noted as an error if encountered in practice as an argument for CREATE. Thus the first constraint can be simultaneously encoded:

CREATE(t) >> HOMOGENEOUS(t)

where

HOMOGENEOUS: set[tuple] → logical
HOMOGENEOUS(ϕ) = TRUE
HOMOGENEOUS(ϕ←t) = TRUE
HOMOGENEOUS(τ←t←t') = (COLUMNS(t) ≡ COLUMNS(t'))∧HOMOGENEOUS(τ←t)

Many of the other equations also follow directly from the definition of the canonical form:

DISPLAY(R(a,t,a')) = t
PARTITION(R(a,t,a'),a'') = R(a,t,a' ∪ a'')
DEPARTITION(R(a,t,a'),a'') = R(a,t,a' - a'')

where ∪ and - are the operators for set union and set difference defined on the type set[attribute]. Once again some constraints must be noted, since both PARTITION and DEPARTITION can only be defined when the partitioning set of attributes are a subset of the defining set of attributes.

PARTITION(r,a) >> ATTRIBS(r) ⊇ a
DEPARTITION(r,a) >> ATTRIBS(r) ⊇ a

where ⊇ denotes set containment and ATTRIBS returns the defining set of attributes for a relation.

ATTRIBS: relation → set[attribute]
ATTRIBS(R(a,t,a')) = a

The operator UNION follows from the canonical form almost as easily by observing that the union of two equally partitioned relations is equivalent to the union of the tuples, only subsequently partitioned.

UNION(R(a,t,a'), R(a,t',a')) = R(a,t ∪ t',a')

The union of two relations that are not defined over the same set of attributes or not partitioned identically is undefined.

UNION(r,r') >> ATTRIBS(r) ≡ ATTRIBS(r') ∧ PARTS(r) ≡ PARTS(r')

where again PARTS is an auxilliary operator as follows:

PARTS: relation → set[attribute]
PARTS(R(a,t,a')) = a'

Projecting a relation on a set of attributes requires the

projection of each tuple in the relation. Thus, in addition to the operator COLUMNS, another tuple operator must be defined to project an individual tuple (see the appendix):

PIECE: tuple × set[attribute] → tuple

PROJECT thus involves the formation of a set of tuple pieces as follows:

PROJECT(R(a,ϕ,a'),a'') = R(a'',ϕ,a')

PROJECT(R(a,τ←t,a'),a'') =

R(a'',DISPLAY(PROJECT(R(a,τ,a')a''))←PIECE(t,a''),a')

Notice that because PROJECT is defined on relations and ← is defined on sets of tuples, the two operators R and DISPLAY must be used to effect type conversions. The constraint for PROJECT is that the attribute set involved be a subset of the defining set of attributes but a superset of the current partitioning

PROJECT(r,a) >> ATTRIBS(r) $\supseteq$ a $\wedge$ a $\supseteq$ PARTS(r).

The cartesian product of two relations follows analogously, given a function that composes a set of tuples with an individual tuple by concatenating the latter to each member in the set:

COMPOSE: set[tuple] × tuple → set[tuple]

The equations and constraint for CROSS are thus:

CROSS(R(a,τ,a'),R(a'',ϕ,a''')) = R(a ∪ a'',ϕ,a' ∪ a''')

CROSS(R(a,τ,a'),R(a'',INSERT,(τ',t),a''') =

R(a ∪ a'',DISPLAY(CROSS(R(a,τ,a'),R(a'',τ',a''')) ∪

COMPOSE(τ,t), a' ∪ a''')

CROSS(r,r') >> (ATTRIBS(r) ∩ ATTRIBS(r')) = ϕ

It is this restriction on attribute names for cartesian products that requires the introduction of the RENAME operator: only through

renaming can relations be joined to themselves. RENAME thus takes a relation and a substitution as arguments to form a new element of type relation with a different set of attribute names. A substitution is a mapping from old attribute names to new attribute names, and includes the operator MAP to effect the renaming of a set of attributes:

MAP: substitution × set[attribute] → set[attribute]

Thus RENAME also has a definition analogous to PROJECT and CROSS:

RENAME(R(a,ϕ,a'),s) = R(MAP(s,a),ϕ,MAP(s,a'))
RENAME(R(a,τ←t,a'),s) =
R(MAP(s,a),DISPLAY(RENAME(R(a,τ,a'),s))←ALIAS(t,s)),MAP(s,a'))
RENAME(r,s) >> (ATTRIBS(r) ≡ ϕ) ∨ (ATTRIBS(r) ≡ COLMS(s))

where ALIAS is the renaming operator for individual tuples, and COLMS is an operator on the type substitution that returns the defining set of attributes for a substitution:

ALIAS: tuple × substitution → tuple
COLMS: substitution → set[attribute]

The last of the initial nine operators to be defined is RESTRICT. This is by far the most complicated operator in that it is the only one that inseparably involves the notion of a block, the multitude of possibilities for the type comparator, and the need for the type restrictor, a discriminated union [Hoare 72].

The restriction operator forms a new relation from a given one by accepting or rejecting all the tuples in each block based on the properties of the block as a whole. Thus a useful auxilliary operator

BLOCK: set[tuple] × set[attribute] × tuple → set[tuple]

finds the subset of tuples having equivalent values as a given tuple for a given set of attributes.

```
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')
```

That is, the result is the given tuple together with all the tuples from the set that match on the given set of attributes, where

MATCH: tuple × tuple × set[attribute] → logical

is defined for the type tuple.

As well as the operator BLOCK and the operator PIECE which was defined for projecting a tuple, the definition of RESTRICT also requires various set and sequence operators to derive the acceptability of blocks according to any given element of type comparator in addition to an operator SEQ which converts a tuple into an ordered sequence of values so that set comparisons can be made without regard to attribute names. Using these one can derive the principal auxilliary operator OK:

OK: set[tuple] × set[attribute] × comparator × set[attribute] → logical

which determines the acceptability of the given set of tuples under a restriction (see the appendix for details). The complete specification for RESTRICT is thus

```
RESTRICT(R(a,ϕ,a'),a'',c,NAMES(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''')
RESTRICT(r,a,c,CONST(t))=RESTRICT(r ⊗ CREATE(t),a,c,NAMES(SEQ(ATTRIBS
                                        (CREATE(t)))))[ATTRIBS(r)]
```

The first equation is obvious, observing that NAMES(a''') converts a sequence of attribute names into the type restrictor. The second states

that if a block is acceptable, its tuples are to be added to the restriction of the remainder of the relation after deleting the block; otherwise only the restriction of the remainder of the relation need be considered. The final equation indicates that the constants can be included by augmenting the attributes with a cross-product, performing the standard restriction, and then projecting the result. The only requirements for semantic well-formedness are

$$\text{RESTRICT}(r,a,c,a') >> (\text{ATTRIBS}(r) \geq \text{MEMBERS}(a) \wedge (\text{ATTRIBS}(r) \geq \text{MEMBERS}(a')) \wedge \text{COMPARABLE}(c,a,a',r)$$

where MEMBERS converts a sequence into a set and COMPARABLE is an operator defined on the type comparator to determine whether the given attribute sequences are c-comparable.

Except for minor details, this completes the construction of the type definition. The formal definitions for the types relation, restrictor, tuple, substitution, and sequence may be found in the appendix.

5. Conclusions

The main objective of this paper has been to demonstrate the feasibility and the usefulness of applying the notions of abstract data types to solve some problems of data independence for data bases. The first aspect, feasibility, was demonstrated through the algebraic specification of quotient relations. The construction of the formal specification is intended to be instructive for the derivations of other data types for data base systems. Similarly to the development of programming, the refinement from a partial description of the syntax to a complete specification of the syntax and semantics was fairly straightforward after some initial insight. The most important step is to determine a canonical form for the type: for quotient relations, a defining set of attributes, a set of tuples, and a partitioning set of attributes. If a single canonical form cannot be devised it is advisable to choose as minimal a set of forms as possible. Subsequently equations can be constructed for each operation involving the type such that, assuming canonical forms as input, appropriate values in canonical form are produced as output. Simultaneously, error conditions can be noted in terms of constraints on the values of each operator's arguments for inclusion in a separate section of the specifications. The proof that the definition developed here is correct with respect to the original definition for quotient relations given by Furtado and Kerschberg is left for future work.

The second aspect , the justification for constructing an algebraic formalism, rests on the utility of the resulting definition. The significance of a complete, precise definition of each object to be manipulated and of each operation to be employed is obvious: implementers can be

given more direction than through vague guidelines, and users can rely on a true understanding of the system that will serve as their tool. Furthermore, everyone concerned can readily learn and understand the error conditions that might arise during processing.

The advantage of the algebraic approach described here is primarily that every operation is described behaviourally. Thus the implementers and users are not distracted by superfluous representational details, but instead they can concentrate on understanding the interrelationships among the operations they intend to perform on the data. Although this is primarily of benefit to designers and to users, this same representation-independence aids implementers by freeing them of others' preconceived notions of efficiency, therefore encouraging them to choose a suitable representation for their particular environment. In the example of quotient relations, the specification of a canonical form that does not involve explicit blocks quickly leads to the realization that only the RESTRICT operator relies on the concept of a block; thus if the number of applications of that operator are relatively few, it may be desirable not to provide such blocks at all. At the same time, however, the algebraic formalism does not mislead users to expect any particular implementation, for example, when actually partitioning will or will not be performed.

A secondary objective for this paper has been to improve some aspects of the presentation of an algebraic specification. As previously stated, the method used here is derived directly from that used most recently by Guttag et al., the major differences being in the specification of errors. Guttag augments the syntax specification to include the specification of elements outside the output domain, e.g.

ONEOF: set → value ∪ {ERROR}

where ERROR is a constant (nullary operator). The output domain is different from the domain value although a "natural" partial one-to-one mapping can be defined between them. Thus, in a formal sense, the expression

$\phi \leftarrow$ ONEOF(τ)

where τ is a set of tuples would not be syntactically well-formed, as the second input domain for $\leftarrow$ must be tuple rather than tuple ∪ {ERROR}. Algebraically, the method used in this paper treats ONEOF(ϕ) as an object of type value which is merely not equivalent to any other object, as the constraints insure that no equivalences hold.

Even informally the presence of {ERROR} does not significantly aid understanding, as this information is easily derived from the constraints (which are most usefully organized by operation); rather its presence can distract a reader, especially if more than one sort of error may arise. The claim that having several different values (e.g., ERROR vs. UNDEFINED) explicitly named by the data type designer allows types of errors to be distinguished cannot be substantiated: the same effect results from having separate entries within the constraints section; furthermore the name of the error is left as the constraint itself.

The specification of constraints may easily be extended to include the specification of error recovery. For example, Furtado and Kerschberg note that a projection is always possible on an unpartitioned relation (assuming, of course, that the projected set of attributes are a subset of the defining set of attributes for the relation) [Furtado 77]. Thus a data type designer may choose to specify the constraint on projections as follows:

$$\text{PROJECT}(r,a) >> \text{ATTRIBS}(r) \supseteq a \wedge a \supseteq \text{PARTS}(r)$$
$$== \text{PROJECT}(r*\text{ATTRIBS}(r), a \cap \text{ATTRIB}(r)) / (\text{PARTS}(r) \cap a)$$

that is, whenever the constraint is not satisfied, signal the error and then continue processing on the unpartitioned relation, projecting on only those attributes which are in the defining set and then re-establishing the original partitioning as far as possible.

The final two improvements to the presentation method are the extensions to provide parameterized data types and alternative operational notations. These are, of course, not new to programming nor to data type specifications [e.g., Wulf 76], but they have not been included previously in algebraic specifications. Both features were extremely useful here, particularly in the specification of the type set[value] together with its conventional operator notation.

Naturally the final test of this specification technique must await a description of some conceptual schema, including its meaningful operations, in terms of quotient relations. Only then can it be seen to what extent the algebraic formalism is an aid to the enterprise, database, and applications administrators in constructing their schemas and mappings.

Acknowledgements

The motivation to specify quotient relations using the algebraic formalism resulted from discussions with A.L. Furtado, who also patiently described their characteristics. Much of the material included here was solidified after discussions with various other colleagues, especially K.C. Sevcik, who read an earlier draft. The research was conducted as part of the Hyades project at the Pontifícia Universidáde Católica do Rio de Janeiro with financial support from that university, the Canadian International Development Agency, the University of Waterloo, and the National Research Council of Canada.

References

[Adiba 76] M. Adiba, C. Delobel and M. Leonard, "A unified approach for modelling data in logical data base design", Modelling in Data Base Management Systems (Nijssen, ed.) North-Holland (1976) 311-338.

[Codd 70] E.F. Codd, "A relational model of data for large shared data banks", Comm. ACM 13, 6 (June 1970) 377-387.

[Codd 72] E.F. Codd, "Relational completeness of data base sublanguages", Data Base Systems (Rustin, ed.) Courant Comp. Sci. Symposia Series, Vol. 6, Prentice-Hall (1972).

[Furtado 77] A.L. Furtado and L. Kerschberg, "An algebra of quotient relations", Proc. of Sigmod Conf. (1977) 1-8.

[Goguen 75] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright, "Abstract data types as initial algebras and the correctness of data representation", Proc. of the Conf. on Computer Graphics, Pattern Recognition, and Data Structures, UCLA extension, IEEE, and Siggraph (May 1975) 89-93.

[Grätzer 68] G. Grätzer, Universal Algebra, D. Van Nostrand, Co. Inc., Princeton, 1968, 368 pp.

[Guttag 75] J.V. Guttag, "The specification and application to programming of abstract data types", Ph.D. thesis, Tech. Rept. CSRG-59, Dept. of Computer Science, University of Toronto (1975).

[Guttag 77a] J.V. Guttag, "Abstract data types and the development of data structures", Comm. ACM 20, 6 (June 1977) 396-404.

[Guttag 77b] J.V. Guttag, E. Horowitz, and D.R. Musser, "Some extensions to algebraic specifications", Proc. of an ACM Conf. on Language Design for Reliable Software, Sigplan Notices 12,3 (March 1977) 63-67.

[Hall 76] P. Hall, J. Owlett, and S. Todd, "Relations and entities", Modelling in Data Base Management Systems (Nijssen, ed.) North-Holland (1976) 201-220.

[Hammer 76] M. Hammer, "Data abstractions for data bases". Proc. of Conf. on Data; Sigplan Notices 11 (1976) 58-59.

[Hoare 72] C.A.R. Hoare, "Notes on data structuring", Structured Programming (Dahl, Dijkstra, Hoare, ed.) Academic Press (1972) 83-174.

[Legard 77] H.F. Legard and R.W. Taylor, "Two views of data abstraction", Comm. ACM 20, 6 (June 1977) 382-384.

[Levy 77] M.R. Levy, "Some remarks on abstract data types", Sigplan Notices 12, 7 (July 1977).

[Liskov 74] B.H. Liskov and S.N. Zilles, "Programming with abstract data types", Proc. of ACM Symp. on Very High Level Languages; Sigplan Notices 9, 4 (April 1974) 50-59.

[Liskov 75] B.H. Liskov and S.N. Zilles, "Specification techniques for data abstractions", IEEE Trans. on Software Engineering SE-1, 1 (March 1975) 7-19.

[Tsichritzis 77] D. Tsichritzis and A. Klug, "The ANSI/SPARC DBMS framework", Tech. Note 12, Computer Systems Research Group, Univ. of Toronto (1977).

[Wulf 76] W.A. Wulf, R.L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs", IEEE Trans. on Software Engineering SE-2, 4 (Dec 1976) 253-265.

[Zilles 75] S.N. Zilles, "Abstract specifications for data types", IBM Research Lab., San Jose, California (1975).

APPENDIX

type

relation

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 {1/2}
DEPARTITION: relation × set[attribute] → relation {1*2}
UNION: relation × relation → relation {1 ⊕ 2}
PROJECT: relation × set[attribute] → relation {1[2]}
CROSS: relation × relation → relation {1 ⊗ 2}
RENAME: relation × substitution → relation {1{2}}
RESTRICT: relation × sequence[attribute] ×
comparator × restrictor → relation {1[2 3 4]}
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]

constraints

CREATE(t) >> HOMOGENEOUS(t)
PARTITION(r,a) >> ATTRIBS(r) ⊇ a
DEPARTITION(r,a) >> ATTRIBS(r) ⊇ a
UNION(r,r') >> ATTRIBS(r) ≡ ATTRIBS(r') ∧ PARTS(r) ≡ PARTS(r')
PROJECT(r,a) >> ATTRIBS(r) ⊇ a ∧ a ⊇ PARTS(r)
CROSS(r,r') >> (ATTRIBS(r) ∩ ATTRIBS(r')) = ϕ
RENAME(r,s) >> ATTRIBS(r) ≡ COLMS(S)
RESTRICT(r,a,c,a') >> (ATTRIBS(r) ⊇ MEMBERS(a)) ∧ (ATTRIBS(r) ⊇ MEMBERS(a'))
∧ COMPARABLE(c,a,a',r)
MIN(t,a) >> t ≠ ϕ
MAX(t,a) >> t ≠ ϕ

equivalences

```
CREATE(φ) = R(φ,φ,φ)
CREATE(τ←t) = R(COLUMNS(t),τ←t,φ)
HOMOGENEOUS(φ) = true
HOMOGENEOUS(φ←t) = true
HOMOGENEOUS(τ←t←t') = (COLUMNS(t) ≡ COLUMNS(t')) ∧ HOMOGENEOUS(τ←t)
DISPLAY(R(a,t,a')) = t
ATTRIBUTE(R(a,t,a')) = a
PARTS(R(a,t,a')) = a'
PARTITION(R(a,t,a'),a'') = R(a,t,a' ∪ a'')
DEPARTITION(R(a,t,a'),a'') = R(a,t,a' - a'')
UNION(R(a,t,a'),R(a,t',a')) = R(a,t ∪ t',a')
PROJECT(R(a,φ,a'),a'') = R(a'',φ,a')
PROJECT(R(a,τ←t,a'),a'') = R(a'',DISPLAY(PROJECT(R(a,τ,a'),a'')
                                          ← PIECE(t,a''),a')
CROSS(R(a,τ,a'),R(a'',φ,a''')) = R(a ∪ a'',τ,a' ∪ a''')
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''')
RENAME(R(a,φ,a'),s) = R(MAP(s,a),φ,MAP(s,a'))
RENAME(R(a,τ←t,a'),s) = R(MAP(s,a),DISPLAY(RENAME(R(a,τ,a'),s))
                                          ← ALIAS(t,s),MAP(s,a'))
RESTRICT(R(a,φ,a'),a'',c,NAMES(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''')
RESTRICT(r,a,c,CONST(t)) = RESTRICT(r ⊗ CREATE(t),a,c,NAMES(SEQ(ATTRIBS
                                        (CREATE(t))))) [ATTRIBS(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(t,a,/c,a') = ¬ OK(t,a,c,a')
OK(t,a,c,a') = case c of
      '=':  VALUES(t,a) ≡ VALUES(t,a')
      '≈':  VALUES(t,a) ∩ VALUES(t,a') ≠ φ
      '⊇':  VALUES(t,a) ⊇ VALUES(t,a')
      '⊃':  VALUES(t,a) ⊇ VALUES(t,a') ∧ ¬(VALUES(t,a) ≡ VALUES(t,a'))
      '⊆':  OK(t,a', ⊇,a)
      '⊂':  OK(t,a' ,⊇,a)
     '.≤':   (MAX(t,a) < MAX(t,a') ∨ (MAX(t,a) ≡ MAX(t,a'))
      '.<':  MAX(t,a) < MAX(t,a')
      '≤':   (MIN(t,a) < MAX(t,a')) ∨ (MIN(t,a) ≡ MAX(t,a')
      '<':   MIN(t,a) < MAX(t,a')
      '≤.':   (MIN(t,a) < MIN(t,a')) ∨ (MIN(t,a) ≡ MIN(t,a'))
      '<.':  MIN(t,a) < MIN(t,a')
```

```
'.≤.':  (MAX(t,a) < MIN(t,a')) ∨ (MAX(t,a) ≡ MIN(t,a')
'.<.':  MAX(t,a) < MIN(t,a')
'≥.':  OK(t,a',.≤,a)
'>.':  OK(t,a',.<,a)
'≥':  OK(t,a',≤,a)
'>':  OK(t,a',<,a)
'.≥':  OK(t,a',≤.,a)
'.>':  OK(t,a',<.,a)
'.≥.':  OK(t,a',.≤.,a)
'.>.':  OK(t,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)
```

type

restrictor

syntax

CONST:	set[tuple] → restrictor	{1}
NAMES:	sequence[attribute] → restrictor	{1}

constraints

-

equivalences

-

Comments: An element of type restrictor is converted from either a set[tuple] or a sequence[attribute]. The alternative syntax provides for an implicit conversion. The lack of constraints and equivalences indicates that the conversion will likely not involve any implementation overhead.

type

tuple

syntax

NEW: set[attribute] → tuple
STORE: tuple × attribute × value → tuple
COLUMNS: tuple → set[attribute]
READ: tuple × attribute → value
PIECE: tuple × set[attribute] → tuple
CATENATE: tuple × tuple → tuple
COMPOSE: set[tuple] × tuple → set[tuple]
ALIAS: tuple × substitution → tuple
MATCH: tuple × tuple × set[attribute] → logical
BEFORE: tuple × tuple × set[attribute] → logical
SEQ: tuple × sequence[attribute] → sequence[value]

constraints

STORE(t,a,v) >> a ∈ COLUMNS(t)
READ(t,a) >> a ∈ COLUMNS(t) ∧ t ≠ NEW(a)
PIECE(t,a) >> COLUMNS(t) ⊇ a
CATENATE(t,t') >> (COLUMNS(t) ∩ COLUMNS(t')) = ϕ
ALIAS(t,s) >> COLUMNS(t) ≡ COLMS(s)
MATCH(t,t',a) >> COLUMNS(t) ⊇ a ∧ COLUMNS(t') ⊇ a
BEFORE(t,t',a) >> COLUMNS(t) ⊇ a ∧ COLUMNS(t') ⊇ a
SEQ(t,a) >> COLUMNS(t) ⊇ a

equivalences

COLUMNS(NEW(a)) = a
COLUMNS(STORE(t,a,v)) = COLUMNS(t)
READ(STORE(t,a,v),a') =
    if a = a'
        then v
        else READ(t,a')
PIECE(NEW(a),a') = NEW(a')
PIECE(STORE(t,a,v),a') =
    if a ∈ a'
        then STORE(PIECE(t,a'),a,v)
        else PIECE(t,a')
CATENATE(NEW(a),NEW(a')) = NEW(a ∪ a')
CATENATE(NEW(a),STORE(t,a',v)) = STORE(CATENATE(NEW(a),t),a',v)
CATENATE(STORE(t,a,v),t') = STORE(CATENATE(t,t'),a,v)
COMPOSE(ϕ,t) = ϕ
COMPOSE(τ←t,t') = COMPOSE(τ,t') ← CATENATE(t,t')
ALIAS(NEW(a),s) = NEW(MAP(s,a))
ALIAS(STORE(t,a,v),s) = STORE(ALIAS(t,s),LOOKUP(s,a),v)
MATCH(t,t',ϕ) = TRUE
MATCH(t,t',α←a) =
    if READ(t,a) = READ(t',a)
        then MATCH(t,t',α)
        else FALSE

type

substitution

syntax

IDENT: set[attribute] → substitution
ALTER: substitution × attribute × attribute → substitution
COLMS: substitution → set[attribute]
LOOKUP: substitution × attribute → attribute
MAP: substitution × set[attribute] → set[attribute]

constraints

ALTER(s,a,a') >> a ∈ COLMS(s)
LOOKUP(s,a) >> a ∈ COLMS(s)
MAP(s,a) >> COLMS(s) ≥ a

equivalences

COLMS(IDENT(a)) = a
COLMS(ALTER(s,a,a')) = COLMS(s)
LOOKUP(IDENT(a),a') = a'
LOOKUP(ALTER(s,a,a'),a'') =
    if a = a''
        then a'
        else LOOKUP(s,a'')
MAP(s,φ) = φ
MAP(s,α←a) = MAP(s,α) ← LOOKUP(s,a)

Comments: The four operators IDENT, ALTER, COLM, and LOOKUP are closely parallel to NEW, STORE, COLUMNS, and READ for the tuple. The major difference is that the "values" for a substitution are themselves attribute names and the default substitution is an identity mapping rather than being everywhere undefined.

```
BEFORE(t,t',φ) = FALSE
BEFORE(t,t',α←a) =
    if MATCH(t,t',α)
       then if READ(t,a) < READ(t',a)
                 then TRUE
                 else FALSE
       else BEFORE(t,t',α)
SEQ(t,φ) = NULL
SEQ(t,α←a) = APPEND(SEQ(t,α),READ(t,a))
```

type

sequence[value]

syntax

```
NULL:  → sequence                              {φ}
APPEND:  sequence × value → sequence           {1←2}
SAME:  sequence × sequence → logical           {1≡2}
LESS:  sequence × sequence → logical           {1<2}
MEMBERS:  sequence → set[value]
```

constraints

-

equivalences

```
SAME(φ,φ) = TRUE
SAME(φ,s←v) = FALSE
SAME(s←v,φ) = FALSE
SAME(s←v,s'←v') =
    if SAME(s,s')
        then v=v'
        else FALSE
LESS(φ,φ) = FALSE
LESS(φ,s←v) = TRUE
LESS(s←v,φ) = FALSE
LESS(s←v,s'←v') =
    if SAME(s,s')
        then v<v'
        else LESS(s,s')
MEMBERS(φ) = EMPTY_value
MEMBERS(s←v) = INSERT_value(MEMBERS(s),v)
```