Relation-arithmetic Revived

Tom Etter
Boundary Institute
March 2000

Preface

This is a working paper, which is to say, it is not a didactic exposition but a chronicle of results written up more or less as they were thought up during my stay at Hewlett-Packard. It should be and will be considerably expanded, particularly with examples and further commentary. Also, its terminology cannot be regarded as final. However I believe that the present version, when taken together with the material cited in the references, is reasonably self-contained. A number of theorems are stated here without proof, but these are in most cases almost immediate consequences of the axioms and definitions.

The axiomatic system presented here is in part my attempt to give a deeper foundation to an ongoing body of work, some of it done at Interval Research, in which the theory of relations is being developed as a covering theory for quantum mechanics and information science. The present system encompasses this prior work, but grounds it in more fundamental concepts and extends its scope. It also promises to bring about a more formal unification of relation theory with the study of artificial languages, including programming languages, though this project is still in an early stage.

Introduction

Russell and Whitehead had planned to cap off their Principia Mathematica with a last volume devoted to what they called *relation-arithmetic*, which was to be a general theory of mathematical structure. Alas, this intriguing project was never completed. The existing Principia does introduce its basic idea, but, as Russell tells us in his book My Philosophical Development, progress ground to a halt when they came to higher-order relations. "Whitehead was to have dealt with them in the fourth volume, but after he had done a lot of preliminary work, his interest flagged and he abandoned the enterprise for philosophy." [ref 1] In fairness to Whitehead, it must be noted that Russell also abandoned the enterprise for philosophy.

Russell had a vision of relation-arithmetic as a tool that would extend the power of ordinary arithmetic to structure in general, including the structure of the empirical world. His vision will become our point of departure.

"I think relation-arithmetic important, not only as an interesting generalization, but because it supplies a symbolic technique required for dealing with structure. It has seemed to me that those who are not familiar with mathematical logic find great difficulty in

understanding what is meant by 'structure', and, owing to this difficulty, are apt to go astray in attempting to understand the empirical world. For this reason, if for no other, I am sorry that the theory of relation-arithmetic has been largely unnoticed." Bertrand Russell [1]

Here is relation-arithmetic in a nutshell:

Relations A and B are called *similar* if A can be turned into B by a 1-1 replacement of the things to which A applies by the things to which B applies. *Similarity* in this sense is a generalization of the algebraic concept of *isomorphism*. If, for instance, we think of a group (as defined in group theory) as a three-term relation x = yz, then isomorphic groups are similar as relations. The *relation-number* of a relation is defined as that which it has in common with similar relations. *Relation- arithmetic* was to be the study of various operators on relation-numbers.

For reasons that will become clear below, we'll substitute the word *shape* for Russell's term *relation-number*. Thus, in our current language, the *shape* of a relation is what is *invariant* under *similarity.* Note that these three words have analogous meanings in geometry.

Despite its great promise, relation-arithmetic didn't get very far. The problem is that the most important combining operators for relations, such as cross product and join, are *not* invariant under similarity. That is, if A is similar to A' and B is similar to B', it does not follow that the cross product or join of A and B is similar to the cross product or join of A' and B'. This can be seen from a very simple example. Consider a unary relation R with only one tuple (its table has one row and one column). Let the value in that row and column be v. Replacing v by any other value w produces a relation R' that is similar to R. Now consider the cross product (Cartesian join) RR; it consists of the ordered pair vv. But the cross product RR' consists of the ordered pair vw, so RR and RR' are not similar, despite the similarity of their components.

Again there is an analogy to geometry. You can't combine geometric shapes into a single shape if all you know is their similarity classes, since that doesn't tell you their relative sizes. For that you need to know their *congruence classes*, which requires that they be parts of a single encompassing "context shape" called *space*. Is there an analogous solution to the problem of combining relations?

In fact there is. Fortunately, congruence has a relational analogue. To make use of it, however, we can no longer work with relations given separately but must work with *partial* relations within a single encompassing *context relation*. By a *partial* relation will be meant any subset of cells in a relation table (see Section 2). Given a context relation C, part A of C is called *congruent* to part A' of C if we can map the rows and columns of C that define the members of A onto those that define the members of A' in such a way that corresponding cells in A and A' have the same values. If we substitute *congruence* for *similarity* in the definition of *relation-number*, then operators like product and join can in fact be defined in an invariant way, and Russell's conception of relation-arithmetic makes sense. Since

Russell's definition of these words is not in general usage, this substitution should not produce confusion, so let us hereby make it:

A *relation-number* is defined as an equivalence class of partial relations under congruence.

It might seem strange that relation-arithmetic must operate on partial relations, and yet if we think about the set-theoretic underpinnings of ordinary arithmetic, we also find the need for an encompassing whole, a whole within which the operations of union, intersection and Cartesian product can be defined. Operations are relations, relations are about related things, and things can only be related if they coexist as parts of the same universe of discourse. The universe of discourse we need for relation-arithmetic just happens to be itself a relation.

Congruence is a stronger relation than similarity, i.e. congruent parts are similar, but similar parts need not be congruent. Operators on shapes, i.e. operators that are invariant under similarity, will be called *first-class*, while operators that are only invariant under congruence will be called *second-class* (in Section 2 we'll also consider *third-class* operators that depend on the accidents of how shapes are represented). The first-class operators include a variety of single-place *morphing* and *abstracting* operators, but for two or more places there is essentially just one first-class operator, which is a limited case of the Cartesian join. The analogues of addition and multiplication, which are append and general Cartesian join, only appear in the second class, which is where one actually finds the kind of relation-arithmetic that can keep us from going "…astray in attempting to understand the empirical world."

Though congruence is stronger than similarity, it is *invariant* under similarity, which means that our new relation-arithmetic is still essentially about *shape.* Substituting congruent for similarity may alter the details of Russell's relation-arithmetic, but it preserves its spirit and its essential aim.

Despite Russell's advocacy, the theory of relations remained a relatively obscure province of abstract mathematics until 1969 when E. F. Codd of IBM published several papers on what he called *The Relational Model for Database Management*. His relational ideas were slow to penetrate the mainstream of computer science, but have over the years come to dominate the design of commercial databases, which certainly would have made Russell happy. What began as a philosophical inquiry by Pierce, Frege, Russell et al has finally turned into applied mathematics. The relationship between philosophy and applied mathematics is a two-way street, however, and Codd's dogged struggles with the real world have brought new basic conceptual problems to light.

Codd's relational model is based on what he called *relational algebra*. This "algebra" includes several truly *functional* operators on relations, that is, operators for which the resultant relation is a function of the argument relations and nothing else. These are among the operators of relation-arithmetic, and part of our task in Section 3 will be to sort them into first-class and second-class (none of them are third-class).

But his model also includes certain so-called operators that might better be called *selectors* (indeed one of them is actually called *selection)*, since their resultants depend not only on the operand relations, but also on the choice of particular columns and values. It could be argued that column names and values belong to relations as such, and Codd and his followers have *defined* relations in that way. However, they certainly don't belong to shapes and relation-numbers. It seems to us better to regard *selection* as something logically distinct from *functional determinism* (think about the axiom of choice in set theory – a so-called *choice function* is a *selector* rather than a *determiner*).

In Section 4 we'll see how to bring selection into the domain of shape as an extension of relation-arithmetic. We'll also see how relation-arithmetic can be brought under the aegis of *invariant selector sequences*, which we'll call *structures.* The theory of *structure* is needed for an entirely *formal* database model, that is, a model which is independent of *implementation* not only in the domain of computers but in the domain of all objects in the empirical world.

As we have described it so far, our new relation-arithmetic is about re-formalizing what are for the most part familiar ideas. However, it also has one radically new feature not found in either Russell's relation-arithmetic or Codd's relational algebra, nor anywhere else so far as we know, which is a certain concept of relational "opposition". This concept leads to a very useful distinction between "negative" and "positive" relation-numbers. Just what all this means will be spelled out in detail in Section 2; suffice to say here that *opposition* is built into our concept of shape at the most eliminatory level, as presented in Section 1.

Section 1. Abstract Shapes

In the introduction we followed Russell in defining a shape as a similarity class of relations. In this section we'll define shapes abstractly without reference to relations or their tables.

Our model (in the sense of an exemplar) is Hausdorff's point-set topology. Topology began as "rubber sheet" geometry, where the topology of a figure was defined as what is invariant under bi-continuous transformations of that figure. Hausdorff discovered that this invariant topological "shape" could be captured by a remarkably simple set of axioms that don't mention geometry at all, but are stated entirely in terms of the relation of *inclusion* among a set of so-called point-sets [ref 2]. To study particular topologies you still need the apparatus of geometry, including coordinates, triangulation and all the rest. However, the fundamental concept of a continuous function, along with other important topological concepts like boundary, connectivity and dimension, can be defined directly in the abstract system. More important, the abstract approach reveals a wealth of formal possibilities which would not occur to the geometric imagination without it, many of which go beyond geometry altogether

We shouldn't try to carry the parallel to topology too far. The important thing is that relational shape, like topological shape, can be captured by very simple axioms that don't involve the specialized concepts we use for representing and working with particular

instances. Though we'll use the language of sets in our informal exposition, we don't actually need sets or set membership in the abstract formalism of shape itself. In fact, we really need only one basic concept, namely that of *equality*, though we need it in five different flavors. We don't even need the identity predicate "x=y", since our five kinds of equality can together take its place. These five, plus the logical apparatus of predicate calculus, are enough to tell the whole story (more about this in Section 2 and 3). As in the case of topology, we still need a more concrete formalism to study particular relational shapes, and we'll introduce this in Section 2. But a surprising number of basic relational ideas can be expressed in the language of abstract shapes alone, and it appears that once again abstraction reveals its power to show us formal possibilities that would not occur to the more concrete imagination.

Definition: A *shape* is defined as a set of elements called *cells* on which there are the five equivalence relations satisfying axioms 1, 2 and 3 below. Let x and y be cells. Here are these equivalence relations, which we'll call *shape equalities*, or simply *equalities* when the context allows.

1) *In the same row,* written row(x=y).
2) *In the same column,* written col(x=y).
3) *Have the same value,* written val(x=y).
4) *Are row-friends,* written rf(x=y)
5) *Are column-friends,* written cf(x=y)

A brief note on the word *equality*, which in mathematics is sometimes used interchangeably with the word *identity* for the relation "=". We are not following that custom here, but are adopting the more colloquial usage which allows things to be equal in different ways, for instance, *equal* in size, having *equal* rights etc.

We'll write row(x≠y) for the negation of row(x=y), and similarly col(x≠y) for the negation of col(x=y), etc.

The first three equalities are all we need to define shape in Russell's sense. The last two bring in the new concept of *relational opposition* mentioned at the end of the introduction, and we'll focus on this concept first.

Definition: If x and y are not friends, we say they are *foes*. More formally, this means that x and y are *row-foes* if rf(x≠y), and that x and y are *column-foes* if cf(x≠y).

These five relations are assumed to satisfy the three standard axioms for an equivalence relation:

AE1 Reflexivity. x=x
AE2 Symmetry. If x=y then y=x
AE3 Transitivity. If x=y and y=z then x = z

Note! The following theorems are stated for the equality *row-friends*. However, all of the axioms of shape theory are symmetrical in rows and columns, so for each of these theorems there is a *dual* theorem in which *row-friends* (abbreviated *rf*) is replaced by *column-friends*.

Theorem. My friend's friend is my friend, i.e., if rf(me=friend) and rf(friend=friendsfriend) then rf(me=friendsfriend).

Proof: This is just a restatement of AE3 with different variables.

Note: The words "me", "friend", "friendsfriend" etc. are just variable names like x, y and z, and have no further formal significance.

Theorem. My friend's foe is my foe. In formal terms, if rf(me = friend), and if rf(friend ≠ badguy), then rf(me ≠ badguy).

Proof: If my friend's foe were my friend, i.e. if rf(me = badguy), then, by AE2, rf(badguy = me). By assumption rf(me = friend), so by AE3, rf(badguy = friend), which means that it would not be true that rf(badguy ≠ friend), i.e. badguy would not be my friend's foe.

Theorem. My foe's friend is my foe, i.e., if rf(me ≠ badguy) and rf(badguy=badfriend) then rf(me≠badfriend).

Proof: By AE2, rf(badfriend=badguy). If rf(me=badfriend), then by AE3, rf(me=badguy), contrary to assumption.

What about my foe's foe? Is he my foe or my friend? The AE axioms have nothing to say about him, so we'll have to introduce a new axiom:

Axiom 1. **The warrior's axiom**. My foe's foe is my friend**,** i.e. if rf(x≠y) and rf(y≠z) then rf(x=z), and also, if cf(x≠y) and cf(y≠z) then cf(x=z).

Theorem. There are at most two equivalence classes under rf, i.e. for any x, y and z, either rf(x=y) or rf(y=z) or rf(x=z); ditto cf.

Proof. If there were more, we would be able to find three cells of which no two are friends, i.e. we would have rf(x≠y) and rf(y≠z) and rf(x≠z), contrary to Axiom 1.

The following definitions are intended as aids to the imagination, and strictly speaking belong to set-theoretic models of our formal system rather than to the system itself. Note, however, that the more formal statements of our axioms and theorems in terms of row(x=y) etc make no use of them.

Definition. A *row* is defined as an equivalence class of cells under row(x=y).
Definition. A *column* is defined as an equivalence class of cells under col(x=y).
Definition. A *value* is defined as an equivalence class of cells under val(x=y).

Definition. An *RFclass* is defined as an equivalence class of cells under rf(x=y).
Definition. A *CFclass* is defined as an equivalence class of cells under cf(x=y).

Axiom 2. **The row-mates axiom.** Cells in the same row are row friends, i.e. if row(x=y) then rf(x=y). Also, cells in the same column are column friends, i.e. if col(x=y) then cf(x=y).

Theorem. If x and y are in different rows and rf(x≠y), then for all z in the row of y, rf(x≠z). Informally, if x and y are foes, then all of x's row-mates are foes of all of y's row-mates. Ditto for columns.

Proof. Suppose z were a friend of x. Since by Axiom 2, any row-mate z of y is a friend of y, by transitivity y would have to be a friend of x, contrary to assumption.

For any two rows X and Y, either all the X cells are friends of all the Y cells, or else all the X cells are foes of all the Y cells (ditto columns). Thus friend and foe can be seen as equivalence relations on rows and columns taken as a whole.

Axioms 1 and 2 define a very rudimentary concept of shape, but like Hausdorf's first three axioms for a topological space, they are too general for most purposes. A third axiom is needed that makes the value of a cell into a function of its row and column:

Axiom 3. The unique value axiom. If row(x=y) and col(x=y) then val(x=y).

Taken together, axioms 2 and 3 say that row and column are the "independent variables" on which the other three depend.

Definition: A *shape* is a cell set satisfying axioms 1, 2 and 3.

Finally, the following two axioms define two very important kinds of relational shape. The fourth puts us into the domain of relations, while the fifth creates a tighter integration between the concepts of rows, columns and values and the concept of friendship

Special axiom 4. The grid axiom. For any x and y there exists a z such that row(x=z) and row(y=z).

Definition: A shape satisfying axioms 1, 2 and 3 plus the grid axiom is called a *relational shape.*

Russell and Whitehead defined a relation as a set of ordered n-tuples, and this definition has become standard in set theory [ref 3]. If you write these n-tuples in a list with their places aligned as columns, you get a grid of cells containing their values, which is why axiom 4 is called the grid axiom. Notice, however, that axiom 4 does not specify any particular ordering of the columns. We believe this lack of order is an essential feature of relational shape, even though it makes our definition more abstract than Russell's [ref. 4].

Codd also got rid of column order, but kept its ghost in the form of column names. However, we can easily exorcise this ghost by turning its job over to horizontal *keys*.

Definition: Row R is called a *duplicate* of row R' if for every column C, any cell in R and C has the same value as any cell in R' and C (axioms 3 and 4 guarantee the existence and uniqueness of these two cells). Ditto columns.

Special axiom 5. The friendly twins axiom. Duplicate rows are always friends. Ditto columns.

Definition: A shape satisfying axiom 5 is called *compact*.

There are a number of other kinds of shape that we'll distinguish by name but not by special axioms. Here are three important "degenerate" cases:

Definition: A shape is called *peaceful* if it has only one rf class and one cf class. If it has only one rf class it is called *row-peaceful*, while if it has only one cf class it is called *column-peaceful*.

In a peaceful shape, the relations rf(x=y) and cf(x=y) can be simply ignored. Peaceful shapes are of course always compact.

Definition: A shape is called *homogeneous* if it has only one value.

Definition: A *natural number* is a compact homogeneous shape with only one column.

Notice that a "natural number" is indeed characterized by the number of its rows. As we'll see in the next section, we can, using the above definitions, construct the arithmetic of natural numbers without using any set theory, or even the identity relation "=".

Section 2. Tables and their shapes

A *table*, informally speaking, is a shape in which every row, column, value and friends class has been given an *identifier*. Tables are to relational shapes what coordinate systems are to geometric shapes, i.e., a coordinate system supplies identifiers to those featureless points of which geometric shapes are made. Most of the time we'll be working with tables, but we should always aim to think in terms of *invariant* features of tables. What this means in practice is that we should try to avoid letting our mathematics depend on the *meaning* of our ID's. These should be regarded purely and simply as *names*, replaceable at will by any others.

We now come to a subtle but important point. Section 1 was nominally about *cells*, and in our informal discussions we applied the ordinary language of sets to these so-called *elements*. However, the more formal statements of our axioms and theorems never mention sets or individual cells, and make no use of the two predicates "=" and "$\in$" of set theory,

either explicitly or implicitly. Pure shape theory, properly understood, is based only on the five basic equivalence relations we called *shape equalities*. It should be noted in passing that the axioms of topology, though they seem to be about sets, don't really use "∈" either.

If the five equalities are the only predicates of our formal language, they can together be made to do some of the work of the *identity* predicate "=" of standard predicate calculus, with the meta-axiom of substitution becoming a theorem about their conjunction. And yet without a universal "=", you can't formalize *definite description* in a general way that applies to new predicates. You need "=" to say "The cup on the table", which means "Any cup x on the table, it being the case that if y is also a cup on the table then y = x." You need "=" even to say abstract things like "The sum of x and y" or "The least prime greater than 10", or "The cell which is in the same row as x and in the same column as y" (see refs. on definite description). Without "=", there is no way to use the properties of a thing to speak of that thing in the *singular* [ref 6].

Nor, fortunately, is there any need to do so. Pure shape theory can be formulated as *mathematics without ontology*.

Table theory, on the other hand, starts out with definite things called *identifiers*, which are elementary linguistic expressions that function as names or "tags". Identifiers are divided into five kinds: RowID, ColumnID, ValueID, RFclassID, and CFclassID. We can think of these identifiers as corresponding to row, column, value, row-friend class and column-friend class of shapes, but table theory as such doesn't depend on abstract shape theory as presented in Section 1.

***Definition*:** A *table-cell* is defined as an expression of the form <RowID, ColumnID, ValueID, RFclassID, CFclassID>

Definition: A *table* is defined as any set of table-cells which satisfies the following axioms:

Table Axiom 1. The warrior's axiom. There are at most two RFclassID 's, ditto CFclassID's

Table Axiom 2. The row-mates axiom. The RFclassID of a cell is a function of its RowID. Ditto CFclassID

Table Axiom 3. The unique value axiom. The ValueID of a cell is a function of its RowID and its ColumnID

Theorem. Every subset of cells of a table is a table

Proof. Let S be a subset of table T. Clearly there are no more identifiers in S than in T, so S satisfies Axiom 1. A function confined to a subset of its domain is still a function, so S satisfies Axioms 2 and 3.

Definition: A *row* of a table is the subset of all cells sharing a particular RowID

Definition: A *column* of a table is the subset of all cells sharing a particular ColumnID

The familiar meaning of table requires that there be a cell in the intersection of every row and column, i.e.:

Special Table Axiom 4. The grid axiom. For any row identifier R and column identifier C, there is a cell containing R and C.

Definition: A *relation* is defined as a table satisfying axiom 4.
Definition: A *relationship* is defined as a row of a relation.

Axiom 4 is not included in the basic definition of table, since it is often useful to consider cell-assemblies of a more general kind, such as those consisting of several rectangular parts having no rows or columns in common. This added generality makes it easier to make the connection between abstract shape theory and certain computer languages based on so-called *triple-stores*.

Definition: A *tuple* is defined as a pair <RowID, RFclassID>
Definition: An *attribute* is defined as a pair <ColumnID, CFclassID>
Definition: A *triple-cell* is defined as the triple <tuple, attribute, ValueID>

Clearly there is a one-one mapping between triple-cells and table-cells. A set of triple-cells satisfying the three cell axioms will be called a *triple-store*. We won't discuss triple-stores here; let it suffice to note that we can apply many of our table results to them.

Let's now return to the concept of shape. What is the shape of a table?

First let's ask what does it mean for table T to have the same shape as table T'? Intuitively this means simply that we can get T' by consistently substituting new ID's in the cells of T. This is essentially Russell's definition of *similarity*, which we met in the Introduction, so we'll keep his term. To state this definition precisely, it helps to first define two kinds of one-one mapping:

Definition: An *ID mapping* from table T to table T' is defined as a one-one mapping of the identifiers of T to those of T' that preserves kind, i.e. row identifiers map onto row identifiers, column identifiers onto column identifiers, etc.

Definition: A *cell mapping* is defined as a one-one mapping of the cells of T onto those of T'.

Definition: Two tables T and T' are called *similar* if there is an ID mapping from T to T' for which there exists a cell mapping M that takes the identifiers of each cell x of T onto corresponding identifiers of M(x) . Such an ID mapping is called a *similarity transformation,* or *similarity*.

We can think of a similarity as the analogue of a bi-continuous mapping of geometric figures. The topology *of a figure* was originally defined as that about the figure which is invariant under bi-continuous mappings. Before we can associate this "what" with a topological space in Hausdorff's sense, we must first translate Hausdorff's basic concepts, namely points and neighborhoods, into the language of figures. We can then restate the idea of two figures having the *same topology* by saying that their Hausdorff topologies are "isomorphic", i.e. that points map onto points in a way that preserves open sets.

Likewise, before we can associate what is similarity-invariant about a table with a shape as defined in Section 1, we must first define the five shape equalities in the language of tables. Fortunately, this is very easy:

Cross-system Definition: Given two table-cells x and y, we say that they are *in the same row,* written row(x=y), if they have the same row ID's. Ditto *same column*, etc.

We can now restate the idea of two figures having the *same shape* by saying that their "shapes" are "isomorphic", i.e. that cells map onto cells in a way that preserves these five equalities.

Definition: A one-one cell mapping M of T onto T' is called an *isomorphism* if for cells x and y of T, row(x=y) if and only if row(M(x)=M(y); ditto columns etc.

A table similarity maps *identifiers*, transforming the table in a way that preserves its cell positions in "identifier space". A table isomorphism, on the other hand, maps the *cell-set*, ignoring the identifiers individually but preserving their equalities.

Fundamental Theorem of Shape. Two tables T and T' are similar if and only if they are isomorphic.

Proof: Assume that T and T' are isomorphic. Let x and y be cells of T and let r be the row identifier of both. Let x' = M(x) and y' = M(y), and let r' be the row identifier of x'. The isomorphism of T and T' means that row(x=y) is preserved by M , hence r' is also the row identifier of y'. The same argument holds for the other four equalities, so M generates consistent identifier mappings that make it a table mapping satisfying the conditions of similarity. Conversely, assume that T and T' are similar. Similarity says that ID's are substituted consistently, which guarantees the existence of an isomorphism M.

We thus see that there are two ways to abstract the concept of shape from that of table. As mentioned, we can think of a table as a kind of coordinate system on a shape. On the one hand, we can start with certain kinds of "coordinate transformations" called similarities, which is somewhat like defining continuous transformations on a metric space in terms of convergence. This was Russell's approach, and it only brings shape into the picture with the concept of invariance. On the other hand, we can start by directly defining shape within the coordinate system, which is somewhat like constructing open sets in terms of spherical neighborhoods. After we take this step, we already have the *idea* of shape, and the only thing left is to "purify" it by requiring invariance under isomorphism. This further step

requires no further construction, since the meaning of isomorphism is already given by our definition of shape.

Which should we use, similarity or isomorphism? Isomorphism is easier to talk about, since a shape is conceptually much simpler than a table, and a cell mapping is more intuitively natural than a five-fold identifier mapping. On the other hand, the identifiers are what we use to describe and manipulate tables, so similarity is much closer to the practical world. In practice we'll use both, and in most cases it won't be worth the trouble to distinguish them, since concepts defined for either have their counterparts for the other.

The question arises whether it is possible to specify a particular shape in pure equality language. Since that language cannot mention *things* of any kind, the answer, strictly speaking, is no. However, we can, for any shape S, produce a *statement* in pure equality language that is true of just those tables having S, and no others. To see the idea here, consider a table T containing two cells in the same row which are column friends. Notice that this description of T is almost in equality language already; the main thing we have to do is rephrase "containing two cells". Here is T's *shape declaration* more precisely:

$$
\begin{array}{l}
(\exists\ x, y) \\
(\\
\qquad (\forall z)(\ \ (\mathrm{row}(z{=}x)\ \&\ \mathrm{col}(z{=}x))\ \ \mathrm{or}\ \ (\mathrm{row}(z{=}y)\ \&\ \mathrm{col}(z{=}y))\ ) \\
\ \ \& \\
\qquad \mathrm{row}(x{=}y)\ \ \&\ \mathrm{cf}(x{=}y)\ \ \&\ \ \mathrm{NOT}\ \mathrm{col}(x{=}y)\ ) \\
\ ) \\
\end{array}
$$

The variables x and y correspond to the two cells. We start by saying they exist. The first line inside the main quantified brackets says that any other cell is row and column equal to either x or to y, while the third line says there are at least two distinct cells x and y with the specified equalities. Although we cannot conclude that there are literally two cells in the table, we can conclude that any other cell would be indistinguishable from x or y, given only the resources of equality language.

The reader may be wondering why there is not a condition on z in the second clause that would confine it to the shape in question. The reason is simply that there is no way to express such a condition. Every pure shape is a universe unto itself, so-to-speak.

Here is the a more formal account of shape declarations in general:

***Shape language*:** The predicate calculus with the five equalities as its only predicates, where these are subject to the equivalence axioms of Section 1 together with Axioms 1, 2 and 3.

***Atomic predicate*:** Either a statement of the form E(x=y), where E is one of the five equality predicates or a statement of the form (NOT E(x=y)).

***Grid-equal*:** Define *grid-equal* notated grid(x=y), to mean (row(x-y) & col(x=y)).

Discriminator: An open statement in equality language that says that anything is grid-equal to one of a certain set of free variables. Informally, it says "x,, y, z etc are all the things there are."

Unifier: An open statement in shape language from which one can derive either E(x=y) or (NOT E(x=y)) for every pair of its free variables.

Shape declaration: The existential closure of the conjunction of a discriminator and a unifier, where the discriminator and the unifier have the same free variables.

Pure shape language is equality language plus the shape axioms of section 1. This means that we don't have to individually specify all of the equalities and inequalities in a table in order to determine them all, since the shape axioms enable us to derive some from others. Thus the literal expression itself is not the shape declaration; rather, any two such expressions represent the *same* shape declaration if they are logically equivalent given the shape axioms.

Theorem: Let D be the shape declaration of T. Then D is true of T' if and only T' is similar to T, it being understood that the variables of D range over the cells of T'.

Section 3. Congruence and table-arithmetic

The key idea that makes relation-arithmetic possible is congruence. Informally speaking, T and T' are congruent if you can get T' by substituting new row and column identifiers in the cells of T. In a context where one is assembling components of various types into the design of complex system such as a computer or a computer program, congruence formalizes the concept of *same type*.

Here is a more formal definition:

Congruence: Two tables T and T' are called *congruent* if there is an isomorphism from T to T' which preserves value and friend identifiers, i.e. which maps ValueID's onto themselves, CFclassID's onto themselves, and RFclassID's onto themselves. Such an isomorphism is called a *congruence*. Congruent tables will be said to have the same *relation-number*.

Theorem: Congruence among the parts of a table is invariant under similarity.

Floating cell: A triple of the form < ValueID, CFclassID, RFclassID>

Table-number: A set of floating cells together with the two equalities row(x=y) and col(x=y) which satisfy Axioms 1, 2 and 3.

Relation-number: A table-number that satisfies Axiom 4.

Table-numbers are midway between tables and shapes in their abstractness. As with similarity, there is a theorem to the effect that being congruent is logically equivalent to having the same table-number. Relation-arithmetic is about operators on table-numbers.

Table operators fall into three basic classes:

Definition: *First-class operators* are operators that are invariant under similarity. These are also loosely called *shape operators*; more accurately, they always correspond to shape operators (we'll see an example below).

Definition: *Second-class operators* are operators that are invariant under congruence but not under similarity.

Definition: *Third-class operators* are operators that are not invariant under either congruence or similarity.

As we remarked in the introduction, Russell tried to do relation-arithmetic with first-class operators alone, but that simply doesn't work. We need second-class operators even for Cartesian joins. The question is whether we need third-class operators for an "algebra" like Codd's. The answer is yes and no. If we work with operators in isolation, the answer is yes. But if we work with *sequences* of operations, it turns out that the whole enterprise can go first-class, although it's beyond the scope of the present paper to spell this out..

These three classes are a rough classification, and there are various kinds of hybrid. For instance, an operator may only be invariant when applied to subtables that are related to each other in certain particular ways; such hybrids are essential for understanding the invariant structure of databases.

We'll first look at a very simple first-class single-place operator to get a more concrete sense of what it means to be first-class. We'll then introduce a pair of first-class two-place operators called *sum* and *product* and use them to formalize the arithmetic of natural numbers as shapes. Next we'll show how to use friendship classes to create a second-class arithmetic of *signed* numbers. Finally, we'll extend this second-class arithmetic to tables in general.

Definition: Let T be any table and let H be a homogeneous table (a table in which all cells have the same value; see Section 1). Define *hom(T, H)* to be the table that results from replacing the value of every cell of T by the value of H. We'll say that applying hom to T is *homogenizing* T.

Theorem: hom(T, H) is first-class, and its shape depends only on T.

Proof: Let T' be similar to T and H' be similar to H. Let U = hom(T, H) and U' = hom(T', H'). We must show that U' is similar to U. By the definition of hom, U is like T except for its values. Ditto U' and T'. We must thus show that there is a value mapping from U to U'

which is preserved by the row-column mapping of T onto T'. But since there is only one value v in U and one value v' in U', there is only one possible mapping, which is the mapping that takes v onto v', and this one obviously does the trick.

Because hom is first-class, there is a corresponding two-place operator on shapes. But since the choice of H' doesn't affect the shape of U, the second place in this shape operator is otiose, and we can simplify hom to a single-place shape operator hom(S) which gives the shape of any table that results from *homogenizing* any table whose shape is S.

If you think this definition of hom(S) is more complicated than it should be, you are right. Notice how simply we can do the same thing with the language of section 1:

Definition: **Homogenizing** (First class). To *homogenize* a *shape* means to replace its val(x=y) equality by one satisfying the condition (For all x and y, val(x=y)).

To summarize: Homogenizing a table is a two-place operator, since it depends on the table and the value that replaces its values. On the other hand, the corresponding operator on shapes is single-place, and can be defined very simply using the shape equivalence relations. Here is another important shape operator of the same general kind:

Definition: **Pacifying** (First-class). To *pacify* a *shape* means to replace its rf(x=y) equality by one satisfying the condition (For all x and y, rf(x=y)). Ditto for columns.

Definition: A ***natural number*** (First-class) is a homogeneous compact one-column shape.

Notice that a natural number is peaceful. It's column-peaceful because it has only one column. It's row-peaceful because it's homogeneous; if it were not peaceful, it would have two duplicate rows that are foes, contrary to Special Axiom 5.

Definition: **Addition** (First-class)**.** Let M be a natural number with m rows and N be a natural number with n rows. To *add* M and N, first create tables T(M) and T(N) representing M and N whose column ID's agree and whose row ID's all disagree, then take the union of these two tables, then homogenize it, then pacify it, and then take its shape. The resulting shape, which we'll call M+N, clearly has m+n members.

Definition: **Multiplication** (First-class). Let M be a natural number with m rows and N be a natural number with n rows. To *multiply* M by N, first create tables representing M and N with the same RFclassID's, then replace every cell of T(M) by a copy of T(N), then homogenize the resulting column, then take its shape, which of course has mn rows.

Definition: **Compacting** (First-class). To *compact* a shape S, notated Compact(S), choose any table with shape S, remove duplicate row-pairs that are foes, ditto column pairs, and then take the shape of the resulting table .

Definition: **Row Count** (First-class). Define *RowCount(S)* for shape S as follows: Choose any table with shape S, pick any column, homogenize it, then compact it, and then take its shape.

The result of applying RowCount is not the number of rows but a natural number that is the absolute *difference* between the numbers of rows in the two rf classes. It is the compacting operation that subtracts them, thus making it possible to have the equivalent of signed numbers without introducing negative and positive as basic ideas.

Though row count, which subtracts the foe count from the friends count in a single table, is first class, there is no first-class subtraction for natural numbers, since it makes no sense to speak of friend and foe between shapes. There is, however, a second-class subtraction operator, which operates on table-numbers rather than shapes:

Definition: An ***Integer*** (Second-class) is a homogeneous compact one-column table-number.

Definition: **Subtraction** (Second-class). Let M be an integer with m rows and N be an integer with n rows. To *subtract* M and N, first create tables T(M) and T(N) representing M and N whose column ID's agree and whose row ID's all disagree, then pacify each of them separately into opposite classes, then take the union of these two pacified tables, then homogenize it, then compact it, and then take its table-number. The resulting integer, call it M-N, has m-n members, and the rf identifier of the larger of M or N.

If we arbitrarily call the two rf identifiers "+" and "-", then we can classify integers as "positive" or "negative", and the usual rules of integer arithmetic apply. Notice that reversing "+" and "-" makes no difference.

What do we mean by *separate* tables? There are five primary kinds of separation, corresponding to the five equalities:

Definition: **Value separation** (Second class). Two tables are called *value-separate* if they have no row ID's in common.

Definition: **Friend-class separation** (Second class). Two tables are called *rf-separate* if they have no row ID's in common. Ditto *cf-separate*.

Definition: **Row separation** (Third class). Two tables are called *row-separate* if they have no row ID's in common. Ditto *column-separate*.

However, these five can occur in many combinations. The combination that comes closest to the commonsense meaning of separate tables is row-plus-column. Notice that some kinds of separation are second-class, some third.

Separation is of course not an operation on tables but a relation between tables. Though *class* was defined above for operations, class also applies to such relations, where a

relational statement is called *invariant* under certain transformations if its truth value is unaffected by transforming its terms. Indeed, functional invariance becomes a special case of relational invariance if we regard functions as functional relations.

We now come to what is perhaps the single most important operation in table arithmetic, which is the Cartesian join. It's a second-class operation in general, though it has a limited special form that is first-class. The Cartesian join is a familiar concept in database theory (Codd called it TIMES), but our more abstract definition of tables calls for a careful new treatment in the language of cells. To that end, it will be helpful to be more specific about the syntax of identifiers

First of all, let's assume that the identifiers are expressions enclosed in angle brackets, i.e. <id1>, <id2> etc. Given this convention, let's next assume that the rf and cf identifiers are <+> and <-> (it doesn't matter which is used for which class). We'll now introduce a syntactical operation on identifiers called *join*.

Definition: **Row Identifier join** (Third-class). If <id1> and <id2> are row identifiers, then their join, written <id1> <id2>, is the expression <id1.id2>. Ditto columns.

Definition: **rf Class Identifier join** (Third-class). If <id1> and <id2> are row identifiers, then their join, written <id1> <id2>, is the row identifier formed according to the four rules (Ditto cf Class identifier join):

1) My friend's friend is my friend, i.e. <+><+> = <+>
2) My friend's foe is my foe, i.e. <+><-> = <->
3) My foe's friend is my foe, i.e. <-><+> = <->
4) My foe's foe is my friend, i.e. <-><-> = <+>

Definition: **Row-join of cells** (Second-class). Let C and D have different column identifiers. Let their row identifiers be <rC> and <rD>, and their rf identifiers be <rfC> and <rfD>. Replace the row identifiers in both C and D by <rC><rD>. Then replace the rf identifiers in both C and D by <rfC><rfD>. The resulting cells, call them C' and D', lie in the same row, and also satisfy Axiom 2 (rf class is a function of row), so the set {C', D'} is a single-row two-column table called the row-join of C and D, written C*D.

Definition: **Row-join** (Second-class). Let M and N be any two table-numbers. Choose column-separate tables T and U representing M and N. Let S be the set of all row joins C*D, where C is in T and D is in U. The table-number of the union of S is called the row-join of M and N, written M*N.

As mentioned, congruence formalizes the idea of *same type*. To assemble components of various types into a compound system, you first have to bring them together as *independent parts* within a context where you can compare their types. This is what the row-join does.

The general row-join is not first-class, since similarity does not preserve the value equalities between M and N. However, in the special case where M and N are value-separate (no value identifiers in common), there are no such equalities, so their join can be elevated to first-class status:

Definition: **Limited Row-join (**First-class**) .** Let A and B be any two shapes. Choose tables T and U representing A and B that are both column-separate and value-separate. Let S be the set of all row joins C*D, where C is in T and D is in U. The shape of the union of S is called the row-join of A and B, written A*B.

If C = A*B, then A and B both "occur" as subtables of C, though not quite in their original forms. Rather, A, as an independent factor of C, has n duplicates of each row of A by itself, where n is the row-count of B; similarly, factor B has m duplicates of each of its rows, where m is the row-count of A. The row-count of A*B is of course mn.

It has been shown that every rectangular shape, i.e. shape that satisfies the grid axiom, can be uniquely factored into independent prime rectangular shapes. [ref 5]. Presumably the prime factorization theorem also holds for shapes in general. This presents a curious situation from the viewpoint of arithmetic, since multiplying the prime factors of a given shape (using first-class row-join) does not in general produce that shape.

The row-join, as defined above, is what in database theory is called the *Cartesian join*. Databases also use a non-Cartesian join that allows A and B to have common column identifiers. The trouble with these, from our present point of view, is that they are third class, since the overlap of column identifiers in T and U is not a function of the table numbers of T and U. This isn't a problem for the Cartesian join, since we can always choose row-separate tables T' and U', and it doesn't matter how we choose them. However, it does matter if the column sets overlap, since there is no way to learn from the table-numbers of T and U just *how* they overlap.

We don't need to take non-Cartesian joins as primitive, however, since they can always be produced by combining Cartesian joins with another kind of operation called *linking*.

Definition: **Linking (**Third-class**).** Let T be a table and let C and C' be two of its columns. To *link* C and C', discard all rows in T which the values of C and C; are unequal. The resulting *linked* table T' will be notated Link(T, C, C').

To create a non-Cartesian join of T and U, first create a Cartesian join and then link the columns you want to identify.

Definition: **Projection (**Third-class**).** Choosing a subtable consisting of the rows defined by a set of row identifiers.

Notice that row-join, linking and projection together can produce *relational composition* in the usual sense. To compose R and U, first form R*U, next link an R

column in R*U with a H column in R*U, and then "hide" these columns by projecting onto the others.

The three operations of row-join, linking and projection, together with a certain vocabulary of particular shapes, constitute a basis for a powerful relational algebra that has been used to simply and naturally represent a wide variety of mathematical objects ranging from quantum computers to circuit designs and AI constraint systems. The basic flaw in this algebra is that it is third-class. In the introduction it was stated that we could salvage Russell's relation-arithmetic by bringing in the concept of congruence. If we had stayed with his original definition of a relation, which made the column order into part of the *definition* of a relation, then this is true, since linking and projection can then be treated as first-class operators that operate on column numbers. However, this is not an option when column order is considered arbitrary, as it is here.

The solution for us involves a new basic concept called *selection.* Unfortunately, an adequate treatment of this concept would take us well beyond the bounds of the present paper. The use of *selector sequences* is discussed in some detail in [ref 5], but the treatment there is not invariant. Here is a very brief informal introduction to invariant selection:

Selection is in a manner of speaking the antithesis of shape. If we think of the *shape* of a table is what remains fixed under free (but consistent) replacement of its identifiers, then selection, roughly speaking, is about identifiers we are *not* free to replace. More exactly, a selection is a set of identifiers that we can't replace without taking into account their role in a larger context.

We've already met a "natural" selection in Section 2 in connection with table-numbers. If we want to specify the table-number of a partial table in isolation, we have no choice but to specify its value identifiers, since we must be able to compare these with the identifiers of other parts. It's only when we consider the context table as a whole that we can substitute for these values. Thus, within the context of the context table, the value identifiers must remain fixed, as indeed they do in most applications of tables in computer science.

The distinction between things we can freely replace and things we can't is familiar in symbolic logic as the distinction between *free* and *bound* variables. Unfortunately, the terminology of logic has it exactly backwards, since what it calls *bound* variables are those we *can* freely replace, whereas its "free" variables are those that point to something outside of the expression where they occur, and thus must remain fixed. But, terminology aside, the parallel to logical variables is important, and we'll make use of it here by turning again to a concept we briefly encountered in Section 2, which is that of a *shape declaration*.

Recall that a shape declaration is a closed statement in predicate calculus that uniquely characterizes a shape, and whose only predicates are equality predicates. Such a statement begins with the existential quantification of a set of variables corresponding to the cells of the shape, and within the scope of these quantifiers makes two assertions, which were called the *discriminator* of these variables and a *unifier* of these variables.

By a selector we'll mean an "alien" free variable that is inserted into the unifier of the shape declaration within an atomic predicate that equates it to a bound variable there. In effect, a selector is a third-class "input variable" to a shape that can be bound outside that shape in order to "morph" it. Linking is one such morphing. If the selector is tied by an atomic predicate to a selector in another shape, it produces an "external equality" between the two. This is the basic mechanism for creating *invariant* selection sequences that build up compound table numbers, where these sequences as a whole are second-class.

Because the formalism of selectors involves linguistic structure and even "input" variables in close connection with invariant relational structure, it is likely that selectors will be an important avenue for bringing relation-arithmetic into computer science.

References

1 *My Philosophical Development* by Bertrand Russell
2 *Elementary Concepts of Topology* by Paul Alexandroff, Dover 1961
3 *Intermediate Set Theory* by F. R. Drake and D. Singh
4 *Structure Theory and the Quantum Core* Ch.2 Section 2 by Tom Etter, Interval Research (unpublished)
5 *Process, System, Causality and Quantum Mechanics* by Tom Etter and H. Pierre Noyes SLAC – PUB – 7890
6 The Scope of Logic, in *Philosophy of Logic* by W. V. Quine, Prentice Hall 1970