THE NESTED RECTANGULAR ARRAY AS A MODEL OF DATA

Trenchard More

IBM Cambridge Scientific Center
545 Technology Square
Cambridge, MA USA 02139
617-421-9234

Abstract

Data, like electricity and gravity, are part of the world in which we live. Some occur naturally, as in the genetic code, while most occur as a consequence of language and social organization. The search for a theory of data, which begins with the choice of a model, is as important and interesting as the development of theories in physics, economics, and psychology.

Most models of data are collections, such as the unnested array of APL, the one-axis nested list of LISP, and the set, which is nested but lacks the properties of order, repetitions, type, and multiple axes inherent in rectangular arrangement. Nested rectangular arrays have all these properties. The existence of simple, universally valid equations in both set theory and linear algebra suggests that equally simple equations may hold for all arrays. The principles of nested collections developed in set theory apply with few changes to the nesting of arrays. A one-sorted theory of arrays, in which type is preserved for empty arrays, provides an algebra of operations interpreted not only for data but also types of data.

0. Introduction

The observation that digital computers process data according to algorithms raises in a modern context two complementary questions that have long been considered in the foundations of mathematics: What are data? -- What is an algorithm? There has been considerable success in answering the second question [1-7] and in studying the complexity of computation [8] and the interaction of computation with data from an algorithmic point of view [9-14].

The purpose of this paper is to consider the first question by proposing the nested rectangular array as a model of data and by examining some of the mathematical properties of the model. The resulting theory of arrays [15-24] is easily restricted to a theory of nested lists [17], which are arrays having just one axis. The purpose of array theory or list theory is to carry the mathematics of nested arrays or nested lists far into the province of algorithms before using programming techniques to bestow the full power of effective computability. The intent is to strike a balance between algorithms and data to suit the needs of modern programming.

Intuitively, an algorithm, effective procedure, or program is a set of instructions, expressed in a finite number of words or symbols, that shows how to compute a partial function relating a set of inputs to a set of outputs. Inputs and outputs are examples of data, as are the intermediate results of computation and the words or symbols that convey the instructions specifying an algorithm. In the design of a data processing system, the choice of operations to be used and the way in which an algorithm is written depend on the organization of the data. An approach to such design that relies too much on the study of programs and partial functions tends to carry excess generality and algorithmic detail into the province that properly belongs to data.

For example, there are elegant ways to describe exactly how vectors and matrices can be combined one component at a time but the manipulation of vectors and matrices as entire objects, regarded as spatially completed entities, obviates the need. The reason for taking a direct, physically motivated approach to the question of data is to choose, within the unrestricted freedom of programming and algorithmic construction, a tightly constrained path of development that is easy to apply, verify, and teach.

In his 1970 address on "The meaning of beauty in the exact sciences," Werner Heisenberg, who formulated the uncertainty principle, put the matter thus: "If we look back on the history of the exact sciences, it can perhaps be asserted that the correct representation of natural phenomena has evolved from this very tension between the two opposing views. Pure mathematical speculation becomes unfruitful because from playing with the wealth of possible forms it no longer finds its way back to the small number of forms according to which nature is actually constructed. And pure empiricism becomes unfruitful because it eventually bogs down in endless tabulation without inner connection. Only from the tension, the interplay between the wealth of facts and the mathematical forms that may possibly be appropriate to them, can decisive advances spring." [25]

Perhaps the growing complexity of "conventional programming languages" [13] can be attributed to an insufficient creative tension between the wealth of possible operations and the endless classification of different types and structures of data. One major source of complexity is the use of error messages and ad hoc boundary condition to determine the effect operations have on degenerate structures of data. Such conditions are easier to tolerate in a programming language than a theory because it is possible to program around anomalies by inserting conditional expressions to test for special cases.

Extensive use of if-then-else conditions to restrict domains of definition or to provide case-by-case definitions has the effect of reducing vulnerability to counter examples. The less coherent the foundation of a programming language, the more difficult it is to derive a contradiction or predict what a combination of operations will do. Far from avoiding the tension between fact and form, a mathematical or physical theory seeks to be maximally vulnerable to contradiction by deriving all results from the simplest principles, pushing every boundary to coherent limits just short of contradiction, and designing experiments that have the greatest chance of disproving the theory.

The physicist appeals to a reality outside his mathematics to discover the very few systems that have any chance of being significant. This is because he has the gift of language to create, among an infinite variety of mathematical systems, more plausible systems than he can possibly evaluate. He has confidence that his central principles are sound if they hold unchanged at boundaries. Such extensive validity is so unlikely that he has faith that his principles reflect "the small number of forms according to which nature is actually constructed." [25]

The lambda calculus and the many equivalent formal characterizations of an algorithm [1-7] have, by virtue of their ability to direct any effectively calculable process, an inexhaustible capacity to define new objects and domains of objects representing data and classifications of data [9-12]. For example, the universal domain for Scott's model of the type-free lambda calculus is the set of all subsets of the set of nonnegative integers. This "incredibly rich" [26] domain, which has the power of the continuum, reflects a mathematical rather than physical view of data.

At the end of his work in 1895, Cantor defined a set as a collection into a whole of definite, distinct objects of our perception or our thought, which are called elements of the set." [27] Some of the intuitive principles Cantor used were codified by Frege in 1893. The most important of these principles is called the axiom schema of comprehension: For any concept expressed by an open sentence, there exists a set, called the extension of the concept, that contains just those objects for which the open sentence is true [28].

For example, "2 is an even number" is a closed sentence. The open sentence "X is an even number" corresponds to the concept of an even number. The extension of this concept is the set of all even numbers, namely, the set of all objects X for which X is an even number. The axiom schema amounts to one rule that states infinitely many axioms. To each open sentence there corresponds an axiom.

The second important principle is the axiom of extensionality. It is based on the belief that if two open sentences are logically equivalent (each implies the other), then their extensions must be equal in the sense that they cannot be distinguished by any operation on sets.

The axiom schema of comprehension together with the axiom of extensionality define the intuitive but contradictory calculus of sets, called the ideal calculus, that served as the basis for set-theoretic reasoning until 1901. Two other principles in this reasoning correspond to the axioms of infinity and choice [28].

The lambda calculus and set theory are derived from language to study primarily the mathematics of undecidable formulas, unsolvable problems, transfinite numbers, and the uncountable continuum. The arena of these powerful tools is not the arena in which to examine the manipulation of finite data. The use of set theory to provide models for programming languages and the lambda calculus does not escape the linguistic origin of these disciplines. The practical problem of programming is to match the limitlessly descriptive power of language, which contains the seeds of algorithms, functions, sets, and the

continuum, not to the endless variety of mathematically constructed classes of data but to the common manifestations of data that originate in physical objects.

To ask "what are data?" is like asking "what is gravity?" or "what is electricity?" Answers can range from the obvious to the profound in both physical and mathematical terms. In man-made languages, machines, and structures and in the naturally occurring genetic code, data are seen to be an essential part of ourselves and the world in which we live. The motivation for the work described here is based on the belief that data are as much part of the universe as other phenomena and on the resulting faith that data, like other phenomena, are subject to intrinsic laws. This is the reason for beginning with a few principles abstracted from commonplace observations of tangible objects and then pursuing the consequences relentlessly into the boundaries of emptiness, singularity, and nesting. If the laws that hold for ordinary data hold as well at the boundaries, then there is a chance that the laws may reflect the forms according to which data are actually constructed.

The method just described for considering a question about observable phenomena is well-known to students of the exact sciences. Past success of this method in other areas suggests that a few general principles derived from the observation of data may be usefully compared with the principles of design for certain programming languages and systems, such as APL [29-33], LISP [34-36], SETL [37-39], extended set theory [40-42], relational data base systems [43-46], and functional programming [13]. These languages and systems are mentioned because they build on simple models of data -- models that have some of the properties of nested rectangular arrays.

1. Principles in the organization of data

The word "data," like the words "dose" and "donate," comes from the Indo-European root "do-," which means to "give" [47]. A datum is "Something given or admitted; a fact or principle granted or presented; that upon which an inference or argument is based, or from which an ideal system of any sort is constructed..."[48]. This meaning, which antedates the computer but accurately reflects the primitive, given nature of data as a basis for further construction, has evolved to include "Numerical information in a form suitable for processing by computer."[47] "Data" is preferable to "information" because the latter word has technical meaning in information theory. The concern here is not with entropy or weighted sums for sets of events but rather with the discrete, structural properties of objects capable of carrying messages and controlling processes.

As reflected in the usage of the word, data is/are perceived as being simultaneously one and many: as an object capable of being held in a collection; as many objects in a collection; as the collection itself. Examples come in pairs: symbols and strings of symbols, numbers and tables of numbers, magnetic spots and disks holding magnetic spots. Such objects, whether simple or structured, interpreted or uninterpreted, physical or conceptual, numerical or nonnumerical, are called data.

The inquiry into the nature of data leads to the following question: What principles of organization are common to certain objects we see and touch? What common properties can be abstracted from a shelf of books, a printed page, a deck of punched cards, a filing cabinet, a calendar, a coordinate map, a chessboard, a paneled door, a wall of windows, a flight of stairs, and a stack of boxes? These given objects are all quite different but the principles of aggregation, nesting, well-ordering, repetition, smoothness, valency, arrangement, and orientation are the same.

We organize our perceptions by treating many objects as one collective object. This is the principle of aggregation. The aggregation of objects into a collection is described by saying that the collection holds the objects as items. For example, a word holds certain letters and each letter so held is an item of the word. Reference to an object as an item connotes the existence of a location in a collection and the existence of a relationship between the collection and the object: the collection holds the object at the location and the object is said to occur in or be an item of the collection.

A collection is an object just as its items are objects. Hence the items of a collection may themselves be collections. This is the principle of nesting. For example, the words held in a sentence are collections of letters. A punched card is a collection of columns and each column is a collection of punched and unpunched spots.

The kinds of collections that are most often used to organize objects are well-ordered in the sense that every nonempty subcollection has, according to some sort of convention, a first item. This principle of well-ordering prevents a collection from being dense or continuous. The process of striking or deleting zero or more items from a well-ordered collection and then closing up the gaps without reordering the remaining items results in a subcollection, which is also well-ordered.

Every nonempty well-ordered collection holds a first item, which if deleted results in a subcollection that again, if nonempty, holds a first item, etc. This process identifies successive items -- first, second, third, etc. Like letters in a word,

words in a sentence, books on a shelf, or panels in a door, identical copies of an object may occur as different items in a collection. Repetitions are significant and therefore preserved. This is the principle of repetition.

The most common well-ordered collection is the list. Each item of a list is adjacent to at most two other items along a single axis. For example, a word is a list of letters. An odometer displays a list of digits. A flight of stairs is a list of steps where each step is a pair (list of two items) holding a riser followed by a tread. A book is a list of pages where each page is a list of lines and each line is a list of characters. Books on a shelf are items of a list in which the leftmost book is the first item. If some, none, or all of the books are removed from the shelf and the remaining books are simply pushed together without further rearrangement, then the result is a sublist of the original list.

The next most common well-ordered collection is the table. A table has two distinct axes of adjacency and an orientation that indicates which axis is first. The first and second axes are called the 0-axis and 1-axis, respectively. Each item of a table is adjacent to at most four other items: two at most in a 0-row parallel to the 0-axis and two at most in a 1-row parallel to the 1-axis.

0-rows are usually called columns, which are displayed vertically. 1-rows are usually called lines, which are displayed horizontally. When this convention is reversed so that 0-rows are called lines and displayed horizontally, an explicit indication must be given that the 0-axis is horizontal. Governmental agencies use the terminology "line," "column," and "table" on printed forms for gathering data. In mathematical contexts, the lines of a displayed table are also called rows.

A table is smooth in the sense that all of its 0-rows have the same extent (hold the same number of items) and all of its 1-rows have the same extent, which may differ from the common extent of the 0-rows. A list is necessarily smooth because it has only one row. This principle of smoothness extends to arrays that have more than two axes: all rows (in the general sense) that are parallel to the same axis have the same extent. For example, when boxes of the same shape are stacked evenly on a loading pallet so that the sides and top of the stack are smooth, the result is an array having three axes. Each row of an array is well-ordered like a list.

The number of axes for an array is an intrinsic property [21] that is called the axis valency, or simply the valency [49] because other kinds of valencies [50] are not considered in array theory. (The words "dimensionality" and "rank" are not used for the number of axes because the valency of a tensor differs from the dimensionality of the underlying vector space and also from the rank of a matrix or linear transformation.)

Lists and tables are special cases of arrays. A univalent array has just one axis and is called a list. A bivalent array has two axes and is called a table. A nilvalent array, which has no axes, is called a single because it must hold precisely one item [21, Sect. 7]. An array may have any finite number of zero or more axes. This is the principle of valency.

Each item of an array occurs at a location in the array: items and locations are in one-to-one correspondence. Identical copies of an object may occur as different items in an array because the items may occur at different locations (provided that the array is large enough). The locations in a list or in a row of an array are visualized as being in one-to-one correspondence with positions on an axis parallel to the list or row.

How are axes perceived? Locally. "Physics is simple only when viewed locally: that is Einstein's great lesson." [51] This principle applies to data. The same system of distinct, directed axes is perceived at every location in an array. For example, gravity and magnetism permeate every box in a stack of boxes. An observer at any location in the stack can determine which direction is down, which is north, and therefore which is east. Axes are intrinsic to the placement of one box beside or on top of another. There is no coordinate system of separate axes existing outside the arrangement of items.

The valency of an array is discovered by counting the axes present at any location in the array. A location is determined by a position on each axis. Each item of an array occurs in as many different rows as there are axes for the array. This is the principle of arrangement. For example, every location on a map is identified by the intersection of a column of quadrangles with a row of quadrangles.

In common experience, many arrangements of data have a generally accepted convention as to which item is first. Most map atlases indicate the column position first followed by the row position but some do the opposite. In either case, as for a matrix of coefficients or the buttons on a telephone, the top left location represents the first item. A frequently occurring alternative is the bottom left location.

The potential for determining a first item is also present in the genetic code. Given a segment of DNA, the living cell knows which of the two strands to read and in which direction. This well-orders the nucleotides or bases in a pair and

well-orders the sequence of base pairs. Both axes of DNA have functional significance. A cell repairs DNA by reading in the direction of the short axis, which has length two, and makes protein by reading in the direction of the long axis.

A segment of DNA can be represented by the following unoriented table:

A C A T G G T

T G T A C C A.

The four possible bases are represented by letters. If the short axis point down and the long axis points to the right, then the first item is located at the top left corner. Which item is second? A second item should be adjacent to the first, but on which axis? The cell apparently does not need to know. It suffices to distinguish between the axes and to locate a second item adjacent to the first on each axis.

Orientation is to some extent an artificial convention that well-orders the collection of axes for an array. The effect is to place the collection of axes in one-to-one correspondence with an initial segment of the finite ordinal numbers. It is then possible to refer to the axes by axis number as the 0-axis, 1-axis, 2-axis, etc. This is the principle of orientation. The well-ordering of each row and of the collection of axes permits the well-ordering of an entire array so that the items of the array can be listed one after another in a standard succession called main order.

For example, a typewritten page with lines of the same length can be treated as a table of characters (including blanks) in which the first item is at the top left corner and the 0-axis is first and vertical. The usual order of reading from left to right and top down corresponds to main order, which is the order used for matrices in both linear algebra and APL.

Main order for an array cycles through all positions on the n+1-axis for each position taken on the n-axis. This same principle of lexicographic ordering causes the last wheel of an odometer to cycle most often. It follows that the second item of a list, table, or array is adjacent to the first item in the direction of the last axis (provided that the extent of the last axis is two or more positions).

The axes for an array have the same orientation at every location in the array. Let the three, directed axes for a stack of boxes point down, north, and east in the order mentioned. The stack is well-ordered by the following rule for determining the next box: take the box adjacent to the east; if none, move west to the opposite boundary and take the box adjacent to the north; if none, move south to the opposite boundary and take the adjacent box below; if none, move up to the opposite boundary and stop. This rule stops at the first box in the stack after moving through successive boxes in main order.

2. A model of data

Many of the objects we see, touch, make, and use are ordered collections of other objects arranged spatially in rectangular array along one, two, or three axes of adjacency. The method of translating a point to a line, a line to a rectangle, and a rectangle to a parallelepiped can be used to increase the valency of an array. For example, a memory chip has two axes, a stack of such chips has three, and a linear arrangement of such stacks has four. Although each bit in a direct access storage device of this nature can be reached through three levels of nesting by a path appropriate for a list of lists of tables of bits, it is more practical to address the device on one level as a quadrivalent array of bits.

Some care is needed to describe the principles of organization inherent in such nested or multivalent arrays. As a consequence, the theory that codifies the principles appears at first glance to be more complex than necessary. Similarly, the description of an activity, such as swimming, often makes the activity appear difficult. Despite complicated descriptions, the eight principles discussed in the preceding section are understood intuitively and used implicitly by almost everyone in modern society. These eight principles are the basis of the nested rectangular array as a model of data.

Principles for the organization of data are abstracted from the great variety of physical objects by incorporating the principles in a mathematical model, called an array. Arrays are taken as primitive objects in the universe of discourse for a theory of arrays. A theory is considerably simplified if it is one-sorted [52-53] in the sense that there is only one universe of discourse, which contains only one kind of object, standard [53] in the sense that all primitive operations are defined for all objects in the universe, and closed in the sense that all operations of the theory, when applied to objects in the universe, result in objects again in the universe.

Is it possible to construct a one-sorted, standard, closed theory of arrays? Yes. The construction has been given in previous papers [15-21]. All objects in the universe of discourse are arrays. Even objects such as numbers, truth-values, and characters are defined as arrays. The result of every array-theoretic operation is an array. Each item of an array must again be an array.

Since arrays are primitive objects, there is no way to define an array in array theory just as there is no way to define a set in

set theory. The properties of the primitive objects are discovered by applying array-theoretic operations to the objects and examining the results for systematic relationships. One would expect the primitive operations of the theory to reflect the eight principles found to be inherent in the organization of data. The correspondence between operations and principles, although indirect, can be made one-to-one.

The structural, Boolean, and ordinal part of the theory, called pure array theory, can be generated from any given array by eight primitive operations. The null, which is the empty list of ordinal numbers, the four primitive dyadic operations of equating, pairing, sublisting, and reshaping, and the four primitive monadic operation of shaping (measuring the rectangularity), numerating, uniting, and extracting (getting the first item) suffice to generate the two truth-values, the finite ordinal numbers 0, 1, 2, ..., and all arrays comprising truth-values and ordinals as leaves and shade leaves [21] at the deepest levels of nesting. Truth-values and ordinals may occur in the same array.

Before examining in Section 7 the individual primitive operations and their relationship to the organization of data, let us discuss further the general outlines of the theory. Additional primitive operations are needed in an applied theory of arrays to extend the structural, Boolean, and ordinal operations of the pure theory to arrays comprising other types of elementary objects at the deepest levels of nesting. In the applied theory, the leaves and shade leaves of an array may be any selection of integer, rational, real, and complex numbers as well as truth-values, characters, phrases, and faults. The class of integers includes the negative integers and the finite ordinals.

Both the pure and the applied theories are one-sorted, standard, and closed. The pure universe is included in the applied universe. The eight, primitive, structural operations are total in the applied theory, and therefore in the pure theory, in the sense that each of these operations is defined for all arrays taken as arguments. Truth-values and ordinals are the only elementary objects in the pure theory: no errors or faults occur.

The technique of recursive definition and the naming of certain defined operations, together with the successive application of primitive and defined operations to arguments, provide the means for defining all other basic operations that are added to the pure and applied theories for the sake of convenience and clarity. It is surprising that no operations for Boolean or ordinal arithmetic need be primitive. Such arithmetics are a consequence of the primitive operations for comparing and combining arrays as geometric, structural objects.

What are the effects of the primitive operations at the boundaries of emptiness, singularity, and nesting? If the items of arrays are always arrays, then how does nesting terminate? The extract or first item of any nonempty array is again an array. Since nesting must continue deeper in nonempty arrays, perhaps nesting terminates at empty arrays. But if extracting is a total operation and the theory is one-sorted and closed, then the extract of an empty array is also an array. Nesting cannot terminate at empty arrays. Since it is always possible to extract an array as "first item" from any given array, the nesting of arrays can never terminate.

The question should be rephrased: How does nesting effectively terminate? Answer: in arrays that hold themselves. This question, which was resolved in 1968 [15, p.6], was the easiest to answer [21] because the author had learned from Quine in 1951 how to cope with the similar question of individuals in set theory [53-55]. All numbers, truth-values, characters, phrases, and faults are incorporated in the universe of arrays as motes, which are singles that hold themselves. The first and only item of a mote is the mote itself.

Certain boundary conditions for emptiness proved to be much more difficult. What is the extract of an empty array? How are empty arrays reshaped to nonempty arrays? What is the result of applying an arbitrary monadic operation to every item of an empty array? What is the Cartesian product of an array of empty arrays? Since empty arrays must be typical in some sense; how are empty arrays related to the types of nonempty arrays? What is the type of an array? These six questions are closely related. Some but not all [18] of the relationships are developed correctly in Version III of the theory [17].

3. Operators

A satisfactory answer to the general question of emptiness, which is developed in Version IV [18-20] and used in Version V [21], depends on a fundamental equation that was not discovered until 1973. The equation, in turn, depends on the transformation of a monadic operation to another monadic operation, not necessarily different, that applies to each item of an array rather than the array itself. Can such transformation be done consistently and coherently for all monadic operations and all arrays? This question has remained central in the development of the theory [56].

Reduction, inner and outer products, and component-by-component application of the monadic and dyadic scalar functions of APL transform functions that apply to scalars to other functions that apply to arrays of

scalars. The difference between functions and these useful methods of transformation was not clearly drawn [33] in 1968 when the APL/360 User's Manual [31] was first widely published. In the fall of 1968, soon after he joined the APL Group at IBM Research, the author introduced the word "operator" [57] to distinguish functions, called operators, that take functions as arguments from functions that take arrays as arguments. The particular interest of identities involving operators was noted at that time in an abstract on "An extension of APL to a theory of arrays" [15, pp. 6, 83]. Defined operators, such as the derivative operator, were discussed in 1970 [15, pp. 55-59].

In the context of set theory, there is a sharp distinction between operations and functions. A function belongs to the universe of sets because a function is defined to be a set of ordered pairs that is single-valued: distinct values must correspond to distinct arguments. One of the principal uses of set theory is the construction of functions as sets. An operation of set theory does not belong to the universe of sets even though the operation is a function in the mathematical sense and can be represented as a set of ordered pairs.

This same distinction is kept in array theory. An operation of the theory is a function in the mathematical sense but does not belong to the universe of arrays. A function can be represented in a variety of different ways as an array, such as an array of pairs. In fact, every array can be interpreted directly as a function in at least two, different, useful ways. From a mathematical point of view, an array can be construed as a family [58-59], which is a function that maps an index set of primitive addresses onto an indexed set that contains the items of the array. An empty array is construed as the unique empty family, which is the empty set.

An array A can also be construed as a correspondence that relates the domain of A, which is the array of all extracts of the items of A, to the range of A, which is the array of all responses of the items of A. The domain and range of A have the same rectangularity [21] as A. The extract of a nonempty array is simply the first item. The extract of an empty array is discussed in Section 5. The response of a two-item array is the second item. The response of every other array is the rest of the array, namely, the list of all items but the first. For each item X of A, the correspondence sends the extract or first item of X to the response of X. If two or more items of A have the same extracts but different responses, only the first occurrence of such an item in A has meaning for the function, which is a set of ordered pairs of the form (extract, response). The function is a subset of the correspondence.

Arrays represent functions and may be construed as functions. Operations take arrays as arguments and result in arrays. Operators take operations as arguments and result in operations, which are called transforms rather than derived operations. This choice of terminology avoids a conflict with the meaning of derivation in mathematical logic, as in "derived rules of inference" or "the derivation of a theorem."

The problem of finding consistent boundary conditions for the reduction and "each" (item-by-item) operators led in 1970 to the generation of a semigroup for the composition of array-theoretic operations [20]. This semigroup was repeatedly regenerated and enlarged over a period of six years. The expectation, until recently, was that a large class of primitive and defined operators would be the most important and useful part of the theory. It was surprising to discover that no primitive operators are required in a theory of finite arrays (although some are needed in a theory encompassing countably transfinite arrays).

All operators for the finite theory can be defined, as in LISP, either recursively or in terms of recursively defined operators [60]. The theory was developed initially from an algebraic point of view: operations and operators were defined to support a rich algebra of universally valid equations. When the theory was examined later from a recursive point of view, it was found that recursions having the simplest boundary conditions conformed to the same identities.

Perhaps the most striking example of correspondence between these two points of view can be seen at the boundary of emptiness in the definition of the fundamental operator of the theory, which is called the replacement or each operator. This operator, which is primitive in transfinite array theory, corresponds to the axiom schema of replacement in Zermelo-Fraenkel-Skolem set theory, MAPCAR in LISP, and item-by-item application of scalar functions in APL. The behavior of the replacement operator at the boundary of emptiness is discussed in Section 6.

If F is an arbitrary monadic operation, then EACH F is a monadic operation called the replacement transform of F. When EACH F is applied to an array A, the result is an array, having the same rectangularity (measured by shape) as A, in which each item F X is the result of applying F to the locationally corresponding item X in A. For example, if A is taken as the pair P Q, then EACH F A is the pair (F P) (F Q) in which F P is the first item and F Q is the second.

The replacement operator has three distinct dyadic counterparts, called each-left, each-right, and each, that transform any given dyadic operation to three distinct dyadic transforms. If F is an arbitrary dyadic operation, then

EACHRIGHT F is a dyadic operation called the each-right transform of F. Similarly, EACH F is the dyadic each transform of F, etc. When EACHRIGHT F is applied to arrays A and B as left and right arguments, the result is an array, having the same shape as the right argument B, in which each item A F X is the result of combining A by F with the locationally corresponding item X in B. Each-left transforms are defined in a similar but converse manner.

Experience with building up definitions, ultimately from eight primitives, of dozens of structural operations for the convenient manipulation of arrays has lead to the following observations. Inner and outer transforms, which correspond to the inner and outer products of APL, can be used in structural definitions but are not wanted in practice. In the few cases where the inner and outer operators might be used, it is almost as convenient and somewhat clearer to construct the desired transforms from more basic operations and operators [19, pp. 31, 55]. Outer transforms are usually replaced by Cartesian products, each-left transforms, and each-right transforms. In the applied theory, if special symbols are used for the traditional matrix products of linear and Boolean algebra, then there is some question whether other kinds of inner products are used frequently enough to justify a special symbol for the inner operator.

The much used monadic operations of sum and product, Boolean sum and product, max and min, and unite and intersect correspond to the familiar dyadic operations of plus and times, "or" and "and," maximum and minimum, and union and intersection, respectively. The first six of these eight dyadic operations are basic scalar functions in APL from which the corresponding monadic operations are defined by reduction. For example, the APL sum +/2 3 4 5 of the four numbers in the list 2 3 4 5 equals 2+3+4+5. How does this concept of reduction extend to arbitrary, nonscalar, dyadic operations, such as union and intersection? This question is considered in greater detail in the Appendix.

A dyadic operation must be applied to a left argument and a right argument in order to return an array as result. Empty arrays and singular arrays fail to provide the equivalent of two distinct arguments to a reduction transform. There appears to be no uniform way to define the reduction operator so that reduction transforms give intuitively acceptable results in every case when applied to empty or singular arrays [17, p. 152]. Nevertheless, it is desirable to keep the theory standard and closed. The recursive definition of reduction can be completed as follows: a reduction transform applied to an empty or singular array results in the extract or first item of the array.

Although this method of completing the definition often gives useful results, most reduction transforms have little mathematical significance when applied to empty or singular arrays. In many of the cases where reduction transforms do have such significance, array theory proceeds oppositely from APL by taking the corresponding monadic operations as primitive or basic. (Basic operations and operators are those that are designated by special symbols or reserved words. Certain basic operations are also primitive.)

Nesting is so fundamental to the organization of data that the restriction to flat (unnested) arrays in APL requires various conventions and special constructions to represent nested arrays. Certain operations that would be quite natural for nested arrays, such as testing equality, are represented by more elaborate operations on constructions involving flat arrays. When nested arrays are permitted in full generality, the need for such constructions and operations greatly diminishes.

Instead of finding that many different kinds of primitive and defined operators evolve from the study of nested arrays to play a central role in the development of the theory, one finds that (i) the four replacement "each" operators do most of the work, (ii) similar "perleaf" and "pertwig" operators for applying operations to the leaves and twigs of arrays are convenient but much less used, and (iii) the reduction and fold (multiple application) operators are useful but infrequently used. One can safely exercise restraint in introducing additional basic operators because it is just as easy to define operators for particular applications as it is to define operations.

4. Each-left and each-right operators

The purpose of this section is to show how the consideration of type arises from the application of an operation to every item of an array, and why the each-left and each-right operators must be distinct from the dyadic-to-dyadic each operator. The latter point is technical but represents a substantial clarification in Version V of the theory. The argument is illustrated by a data base example.

If F is a dyadic operation and DATA is the pair P Q, then A EACHRIGHT F DATA is the pair (A F P) (A F Q) and 2 EACHRIGHT PICK DATA is the pair (2 PICK P) (2 PICK Q). To be more specific, let DATA be a list that represents a relational data base. Each item of DATA is a triple in which the first item is a phrase that represents a student's name, the second item is a Boolean quadruple that represents attendance, and the third item is a real number that represents his cumulative average. The picking operation uses the left argument as an address (in 0-origin) to pick an item from the right

argument. For example, if P is the triple _BYRD_K_S 1101 3.89 that occurs as an item of DATA, then 0 PICK P is the phrase _BYRD_K_S and 1 PICK P is the Boolean quadruple 1101 and 2 PICK P is the real number 3.89.

It follows that 0 EACHRIGHT PICK DATA is a list of phrases, 1 EACHRIGHT PICK DATA is a list of Boolean quadruples, and 2 EACHRIGHT PICK DATA is a list of real numbers. This continues to be the case even when DATA is an empty list of triples: the three results of each-right picking are an empty list of phrases, an empty list of Boolean quadruples, and an empty list of real numbers. These four empty lists have the same rectangularity but differ in type. The relationship of type to emptiness will be discussed in next section.

In the interest of clarity, one would prefer to have distinct notations for the dyadic-to-dyadic operators each-left, each-right, and each. But such notation would be hard to justify if the first two could be subsumed under the third. Fortunately, it was found in Version V that the three operators must be distinct. If A is the pair M N and B is the pair P Q, then A EACH F B is the pair (M F P) (N F Q) just as 2 3 + 4 5 equals 6 8 in APL.

The monadic singling operation results in a single. Thus SINGLE A is a nilvalent array that holds the array A as sole item. An array A is a mote if and only if SINGLE A equals A. For example, SINGLE 2 equals 2. If B is the pair P Q, then (SINGLE A) EACH F B equals the pair (A F P) (A F Q) just as 2 + 4 5 equals 6 7 in APL.

Since 2 is a mote, it follows that 2 EACH PICK P Q equals the pair (2 PICK P) (2 PICK Q) and that 2 EACH PICK DATA is the same as the list 2 EACHRIGHT PICK DATA as long as DATA is nonempty. The attempt in Versions I-IV of the theory to subsume EACHRIGHT under EACH failed because emptiness in either argument of an each transform forces both arguments to be taken as typical. As explained in the next section, the type of an empty array is the empty array itself but the type of an integer, such as 2, is the integer 0. Thus when DATA is empty, 2 EACH PICK DATA is the same as 0 EACH PICK DATA, which is the empty list of phrases instead of the expected empty list of real numbers.

5. Type and emptiness

What is the type of an array and how is type related to emptiness? For example, what is the type of a pair of integers? The phrasing of the question has already given the answer: it is a pair of integers. How can the phrasing be made precise? By capturing the connotation of typicalness in the use of the word "integer." What is the typical integer? Select a particular integer and call it typical. The type of an integer is defined to be the integer 0 in array theory because many equations involving summation and emptiness are valid for 0 but not for other integers [20]. The type of every pair of integers is a pair of integer 0's.

Let TYPE be the monadic operation that returns the type of an array. Thus TYPE 4 5 equals the pair (TYPE 4) (TYPE 5), which is the pair 0 0. Similarly, TYPE 0 0 equals the pair (TYPE 0) (TYPE 0), which again is the pair 0 0. This simple example suggests the following two universally valid equations, which show that the typing operation is pervasive and idempotent:

pervasive TYPE A = EACH TYPE A
idempotent TYPE A = TYPE TYPE A.

An array is typical iff (if and only if) all of its items are typical, which is the case iff all the items of items are typical, etc. This condition pervades down to the leaves, which are motes such as integers, truth-values, characters and phrases. Thus an array is typical iff all of its leaves are typical motes. The typical motes are the integer 0, the rational 0/1, the real 0.0, the complex 0R, the Boolean 0, the blank ' ', the typical phrase _ , and the cipher or typical fault (which is denoted in the APL alphabet by the quote-quad symbol). The type of any mote is the corresponding typical mote. The typical motes equal their own types. It follows that an array A is typical if and only if A equals TYPE A. Since there are no items or leaves in an empty array, all of its items and leaves are typical. Every empty array is typical and so equals its type.

This definition of type, which differs from many others in the literature of programming [61], partitions the universe of arrays into equivalence classes of arrays having the same type. The partition is induced by the classification of motes as integer, real, Boolean, etc. One representative from each equivalence class is taken as the typical array in the class.

An array is monotypic iff all of its items have the same type. Since the typing operation is pervasive, an array A is monotypic iff all items in TYPE A are equal. Empty and singular arrays are necessarily monotypic. Polytypic arrays must hold at least two items of different types. For example, consider the list DATA, which was discussed in the preceding section. Every item in DATA is a triple of the same type as _BYRD_K_S 1101 3.89. Hence DATA is monotypic because every item of TYPE DATA equals the typical triple _ 0000 0.0. The second item of this typical triple is the typical Boolean quadruple 0000.

An array A typifies every array B for which A equals TYPE B. For example, the pair 0 0 typifies every pair of integers. Arrays are said to depict each other iff they have the same type. Thus A depicts B

iff TYPE A equals TYPE B. Every pair of integers depicts every other.

Which item is depictive of the items in an array? This question makes sense only for monotypic arrays. Any item of a monotypic array depicts every item. The type of any item of a monotypic array typifies every item. For example, what are the objects in a bookcase? Pick out any one object. That object is a book. When a child first performs this experiment alone, he does not ordinarily look outside the bookcase for a description of the objects inside. The objects describe themselves. Any object will do because he is interested only in abstracting the properties of a typical book. The same principle applies to arrays.

Since every nonempty array has a first item, the extracting operation FIRST, which extracts the first item, is the operation that is most likely to result in an item of the array. If an array is monotypic, then the extract depicts every item. Otherwise, there is no sense in considering a depictive item. No item can depict all the items of a polytypic array. In the search for a depictive item in an array, it suffices to consider the first item. The type of the extract or first item of an array is called the prototype of the array whether or not the array is monotypic and whether or not the array is empty. The prototype of a monotypic array typifies every item. The prototype of a polytypic array typifies at least the first item.

Let FIRST and REST be the array-theoretic monadic operations for extracting the first item of an array and sublisting all items but the first, respectively. These operations are similar to CAR and CDR in LISP. For example, if A is the triple P Q R, then FIRST A equals P and REST A equals the pair Q R. If B is the pair Q R, then FIRST B equals Q and REST B equals SOLITARY R, which is the solitary or one-item list that holds R. The effect of FIRST and REST on empty and singular arrays is discussed at the end of this section.

The construction of prototypes evolved from that of "patterns" [15-17] and was motivated in 1969-70 by the need to calculate from any given array some sort of typical object that could be used to fill the newly created locations in an APL-like expansion of the array [15, pp. 34, 40]. The construction was also motivated by the desire to reshape empty arrays to arrays of arbitrary shape [17, P. 141]. Gull and Jenkins [61] compare prototypes with other fill items suitable for certain extensions of APL to nested arrays. Brown [62] gives examples that illustrate the usefulness of prototypes.

What is the extract of an empty array? The result must be an array if the theory is to be one-sorted, standard, and closed. Every nonempty array A has the same first item as LIST A, which is the sublist of all the items of A in main order. This is vacuously true if A is empty. Thus FIRST A equals FIRST LIST A for every A. Since the list of an empty array is an empty list, the extract of an empty array is the same as the extract of the corresponding empty list. The monadic operation VACATE makes an empty sublist of its argument by striking or deleting every item. Given an array A, what is FIRST VACATE A?

This question is answered by deriving a formula from two conjectures that relate type to emptiness [19, p. 7]. When the formula is taken as an axiom and combined with other axioms, the conjectures become theorems. The first conjecture is that the extract or "first item" of an empty list is a typical array. (Since an empty array is typical, how could the extract not be typical?) This typical array must equal its type.

FIRST VACATE A = TYPE FIRST VACATE A (1)

The second conjecture is that the first item of an array is depictive of the extract or "first item" of the corresponding empty list. In other words, an array and its empty sublist have the same prototype.

TYPE FIRST VACATE A = TYPE FIRST A (2)

This second conjecture is less convincing than the first because of its asymmetry [63]. How can one particular item of a polytypic array depict the extract of the corresponding empty list? The problem is to compute from any given array some sort of prototype or archetype that can typify the nonexistent items in an empty listing of the array. Should the computation depend on whether the given array is monotypic or polytypic? Equation (2) excludes such dependence. But this raises an important question: How is an empty sublist constructed?

There are two extreme ways to strike the items from an array: all at once or one at a time. The simultaneous way is abstract and global. The sequential way is more physical and local. Array theory is motivated by physical intuition: data are simple when viewed locally. Emptiness depends only on the last item to be deleted, but which item is last in this sense?

From a abstract point of view it makes no difference whether all items are deleted from a copy of an array or whether no items are selected from the array itself. When viewed physically, these two ways of creating an empty list can differ greatly in cost. It is much cheaper to select no items provided that one does not first test all items for sameness of type.

Physical representations of arrays are not ideal collections, like sets, in which all elements are equally accessible. Instead,

such representations are often strongly asymmetrical collections in which the first item is the most accessible and the starting point for most operations. The operation of selecting no items from an array is done locally by beginning with the first item: the first item is not selected. When all items are struck from an array in one operation to make an empty sublist, it is assumed that the first item is the last to be deleted.

This assumption identifies the local operation of selecting nothing from the beginning of an array with that of deleting everything by deleting the first item last. The two operations amount to the same vacating operation, whose asymmetry is expressed in equations (3) and (4). If no items are to be taken from a given array, then one may as well take no items from a singular array, such as a single or solitary, that holds only the first item of a given array.

VACATE A = VACATE SINGLE FIRST A (3)
VACATE A = VACATE LIST SINGLE FIRST A (4)

By the transitive law of equality, it follows from equations (1) and (2) that

FIRST VACATE A = TYPE FIRST A. (5)

If A is nonempty, then it is clear that the extract of the empty sublist of A is simply the prototype of A. This continues to be the case when A is empty. Although an empty array holds no items and is truly empty, the array keeps a prototype, which, according to the type of the empty array, may be any typical array. There are as many different empty lists as there are typical arrays.

Let A be the single of an array B. If SINGLE B is substituted for A in the preceding equation, then

FIRST VACATE SINGLE B = TYPE B

because B equals FIRST SINGLE B. Since typing is a pervasive operation, this last result means that the composition FIRST VACATE SINGLE is a pervasive monadic operation. The operation TYPE is now defined to be this composition. The fundamental axiom relating type to emptiness is simply

TYPE B = EACH TYPE B,

where it is understood that

EACH TYPE B = EACH(FIRST VACATE SINGLE)B.

The replacement operator EACH causes a monadic operation F to apply to each item of an array instead of the array itself. In particular, EACH F SINGLE A equals SINGLE F A for every array A. A pervasive operation, such as TYPE, pervades to the leaves and shade leaves [21] of an array where the operation then applies to motes. How does the pervasion stop?

Let the array D be a mote, so that D equals SINGLE D. Then

TYPE D = EACH TYPE D
= EACH TYPE SINGLE D
= SINGLE TYPE D.

This means that TYPE D is a mote. A pervasive operation must send every mote to a mote. The TYPE operation is now constrained further by requiring that the type of a mote be the corresponding typical mote.

The properties of the typing operation reflect back to the primitive operation of extracting and the basic operation of vacating, which is defined in terms of the primitive operation of sublisting. Empty sublisting must result in an empty list that has a prototype. The extract of an empty array is the unique prototype of the empty array.

How do the operations of FIRST and REST behave at the boundary of emptiness? This question is relevant to LISP, APL, and SETL. If C is a singular array that holds only the item R, then FIRST C equals R and REST C is the result of striking the first item from C and listing the remaining items. But there are no items in C other than the first. Thus REST C equals VACATE C, which is the empty sublist of C got by striking every item from C. The prototype of REST C equals FIRST VACATE C equals TYPE FIRST C equals TYPE R.

Finally, if D is an empty array having TYPE R as prototype, then FIRST D equals TYPE R. REST D is the result of striking the nonexistent first item from D and listing the remaining items. REST D equals VACATE D. The prototype of REST D again equals TYPE R.

6. Recursive definition of replacement

If no operator need be primitive in finite array theory, then how is the most important operator defined, namely, the replacement operator EACH? Answer: by recursion. Given any monadic operation F, the monadic operation EACH F must send an array A to the array EACH F A that has the same rectangularity as A and holds for each item X at a particular location in A the item F X at the same location in EACH F A. The basic idea for the recursive definition is to apply F to the first item of A and EACH F to the rest of A. For example, if A is a 2-by-2 table that holds the items P, Q, R, and S in main order, then F FIRST A equals F P and EACH F REST A equals the triple (F Q)(F R)(F S).

The next step is to hitch F P to the triple and reshape the resulting quadruple (F P)(F Q)(F R)(F S) to a 2-by-2 table having the same rectangularity as A. The

hitching is done by using the dyadic union operation to list the one item in SINGLE F P followed by all the items in the triple. Unlike set-theoretic unions, array-theoretic unions preserve repetitions. In general, B UNION C is the list that holds all the items of B in main order followed by all the items of C in main order.

The essential equation for the recursion is therefore the following:

```
EACH F A  equals                                    (6)
  (SHAPE A)RESHAPE(
    (SINGLE F FIRST A)UNION(EACH F REST A)).
```

As in APL, the reshaping operation [21] takes items cyclically in main order from the right argument to make an array having the rectangularity indicated by the left argument. In this case, the left argument is SHAPE A, which is a suit or list that measures [21] the rectangularity of A.

Equation (6) is certainly true if A holds two or more items. With each step in the recursion, the replacement transform EACH F applies to a sublist of A that is shorter by one item. The recursion must stop because A is finite. The usual way of ending the recursion is illustrated by the following definition, which is quoted from Burge [12, p. 110]:

```
def rec map f x =
        if null x                                   (7)
        then ( )
        else (f(h x)):(map f(t x)).
```

In this definition, map f is the replacement transform that corresponds to EACH F in array theory and MAPCAR F in LISP. The operations h and t, called head and tail, correspond to FIRST and REST in array theory and CAR and CDR in LISP. The definition means that if the list x is not null or empty, then map f x equals a result equivalent to that given by the last line in equation (6). If the list x is empty, then map f x is the unique empty list ( ), which corresponds to NIL in LISP.

Definition (7) is disturbing. The axiom schema of replacement provides the most powerful method of construction in set theory [28, p.52]; MAPCAR is the most useful functional in LISP [36, p.21]; replacement EACH is the fundamental operator in array theory. Here, if anywhere, one would expect to gain insight into the behavior of operations at the boundary of emptiness.

But at the boundary, just where concepts become most difficult and most interesting, the if-then-else conditional lets the programmer escape from a general law appropriate for results away from the boundary to a special law introduced to handle the particular ending conditions. The structure of the conditional does not require that the special law conform in all respects to the general law or that the general law be formulated so as to subsume the special law. Even though the if-then-else construction is certainly useful and necessary in many areas of programming, the power of this conditional to circumvent any difficulty represents an inherent weakness in programming as a conceptual discipline [64].

It is obvious from the last lines of definitions (6) and (7) that EACH F and map f should result in empty lists when applied to empty lists. The uniqueness of the empty set in set theory suggests that the empty list should also be unique. Since set theory provides the foundation for modern mathematics [65], it would be surprising if the empty list were not unique.

Altough definition (7) expresses a set-theoretic truth about the nature of emptiness, the potential for arbitrary choice at the boundary of emptiness can hardly be called beautiful in the mathematical sense. Why should it be necessary to use so powerful a construction as the if-then-else conditional for the definition of so fundamental a concept as replacement? Why should the definition provide the option to posit some other boundary condition? There may be no choice if universal laws for data exist.

We know from experience that empty collections differ according to type. An empty box of eggs differs from an empty box of apples: one holds no eggs; the other, no apples. The same kind of distinction can be made in linear algebra [17, pp. 142-143]. If F is the operation of changing eggs to apples, is it not the case that EACH F changes an empty box of eggs to an empty box of apples?

Why is the empty set unique? Because sets are abstracted from open sentences. The empty set is the class of all objects x for which some logical condition about x is always false. All such open sentences are contradictions and therefore logically equivalent. It follows from the axiom of extensionality that the empty set is unique. This kind of reasoning leads to the elegant theory of sets, which is necessary for the study of infinite classes but awkward for the represention of lists [66] and not appropriate for the study of data.

Abraham Fraenkel, who in 1922 was one of the early and principal developers of Zermelo-Fraenkel set theory, observed in 1953 in his book on Abstract Set Theory that: "From a psychological viewpoint, there can be no doubt that somehow the ordered set is the primary notion, yielding the plain notion of set or aggregate by an act of abstraction, as though one jumbled together the elements which originally appear in a definite succession. As a matter of fact, our senses offer the various objects or ideas in a certain spatial order or temporal

succession. When we want to represent the elements of an originally non-ordered set, say the inhabitants of Washington D.C., by script or language, it cannot be done but in a definite order." [67]

Here the physicist and mathematican might agree to explore a path different from set theory. Arrays have a reality lacking in sets. Given the preceding doubts and conflicting observations, how does one choose between the set-theoretic truth at the boundary and the beauty of a general law reflecting common experience?

A recent biography quotes Freeman Dyson in an observation on his education as a physicist: "From Bethe and Feynman, I learned that mathematical elegance is not enough, a hard lesson for a pupil of Hardy to learn. To do good work in physics, one must also have an instinct for reality, an intuitive sense of the intrinsic importance of things. ... On the other side stood the great mathematician Herman Weyl, who once said to me, 'When I am forced to make the choice between truth and beauty, I always choose beauty.'" [68] As Heisenberg observes, so does the physicist: "The significance of the beautiful for the discovery of the true has at all times been recognized and emphasized." [25, p. 174]

The power manifest in the conditional is too great and too arbitrary to be trusted for the "discovery of the true." With this belief, the author attempted to develop a theory of arrays as far as possible through algebraic methods without using recursive techniques dependent for termination on the conditional. After exploring some incorrect conjectures in Versions I and II of the theory, he found the following equation in 1971 [15, p. 77], which was the result of considering monadic Cartesian products of single and solitary empty lists.

```
EACH F VACATE A =                                    (8)
  VACATE SINGLE F FIRST VACATE A.
```

The derivation of this result was published in 1973 [69]. It is remarkable that Cartesian products, which embody the essence of rectangularity [21], should lead to the above result for replacement transforms.

Equation (8) means that if the replacement transform EACH F is applied to the empty list VACATE A having a prototype P equal to FIRST VACATE A, then the result is an empty list in which the prototype equals TYPE F P. In other words, if F changes eggs to apples, then EACH F changes typical eggs to typical apples in an empty list.

In order to define arbitrary pervading operations that operate on the leaves of an array, the author had to use the technique of recursive definition. This led in 1978 to equation (6) and to a re-examination of some of the earlier results of the theory from a recursive point of view. It was a surprise to discover that (8) is a direct consequence of (6). This means that (6) expresses a recursive relationship about replacement transforms that implies the boundary conditions for termination without involving the conditional.

Such coherence has intrinsic beauty, even though it may contradict one's perception of the truth. Equation (6) is taken as the definition of the replacement operator. All the properties of the operator follow directly from (6), which can be restated in a form similar to (7) to suit the needs of conventional programming.

As mentioned earlier, equation (6) is clearly valid when A holds two or more items. When A is singular, the transform EACH F must apply to an empty list because REST A then equals VACATE A. The behavior of replacement becomes fully known when (6) is solved for A equal to an empty list. Let VACATE B be substituted for A in the equation. It follows immediately that EACH F VACATE B must be an empty list because SHAPE VACATE B equals 0, which indicates the shape of the result. Zero reshaping is the same operation as vacating. The right argument of UNION has no effect on the rest of the equation. It follows that

```
EACH F VACATE B =                                    (9)
  VACATE LIST SINGLE F FIRST VACATE B.
```

Equation (9) implies (8) by (3) and (4).

7. The primitive operations

The purpose of this section is to discuss the eight primitive operations for pure, finite array theory. These operations are related to the eight principles for the organization of data and to the nine axioms of Zermelo-Fraenkel-Skolem set theory.

Equate. Arrays are equal iff they have the same rectangularity, hold equal items at the same locations, and, if empty, have equal prototypes. Sameness of rectangularity means that the arrays have the same valency, the same length on each axis, the same orientation, and therefore the same arrangement of locations. Given any two arrays A and B, does A equal B? The answer A=B is either "yes," which is the Boolean one 1 for truth, or "no," which is the Boolean zero 0 for falsity. These two Boolean numbers are motes and are called truth-values. Thus A=B is a truth-value that equals SINGLE(A=B).

Two arrays may be the same in all respects except orientation. For example, let a 2-by-3 table A be reflected across the main diagonal to give a 3-by-2 table B. Then A is the transpose of B and differs from B only in orientation. In addition to the other principles for the organization of data, orientation is the essential remaining principle needed to define equality in a way that distinguishes A from B.

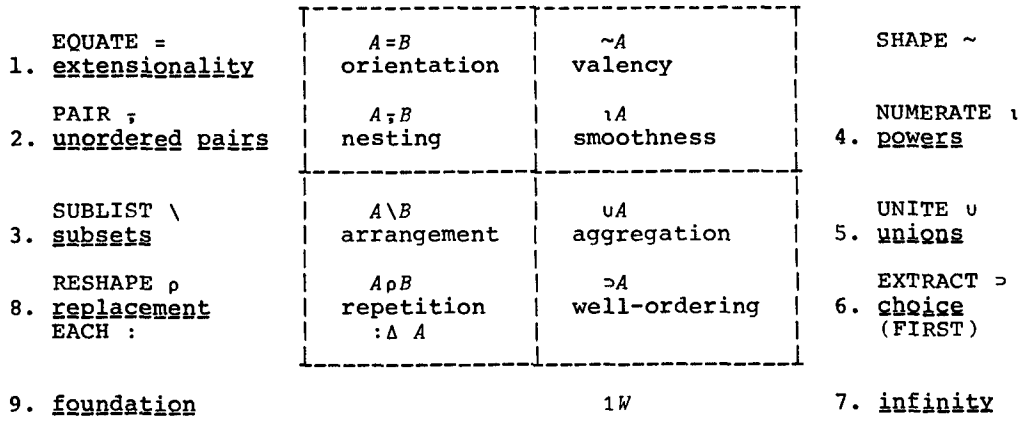

Figure. Except for the anomalies in the last two lines, the figure shows eight groups of correspondences between four areas of consideration. Each group relates an array-theoretic operation name (upper case) to an operation symbol and syntax (italic) to a principle of data organization (lower case) to a set-theoretic axiom name (underlined). The first seven of the latter are numbered in the original order given by Zermelo [70].

The dyadic, array-theoretic operation EQUATE, which is designated by the equal sign and given the infix syntax 'A=B', corresponds to the principle of orientation in the organization of data and to the axiom of extensionality in set theory (Sect. 0). Two sets are equal iff they contain the same elements.

Logic. The operation of equating is the source of all Boolean operations [18, p.80]. These operations are total because every array has two-valued, logical significance. Typical arrays, such as 0, 0, and all empty arrays, are interpreted as false. Atypical arrays, such as 1, 1, and 2, are interpreted as true. The truth-values can be constructed from any array other than 1. In particular, the null is assumed to exist.

truth 1 for NULL = NULL
falsity 0 for NULL = 1

A UNEQUAL B for (A=B)=0

A is typical iff A = TYPE A equals 1.
A is atypical iff A = TYPE A equals 0.

The following two universally valid per-recursive equations are used in the definition of denying and Boolean summing:

NOT A = EACH NOT A
BOOLSUM A = EACH BOOLSUM PACK A.

Per-recursive means "through in depth." Packing trims an array, causing all items to have the same rectangularity, and then interchanges the top two levels [19, 22]. For example (P Q R)(S T U V) is trimmed to a pair of triples (P Q R)(S T U) and packed to a triple of pairs (P S)(Q T)(R U).

NOT must pervade (a copy of) an array to the leaves and shade leaves and there replace such motes by motes. If A is a mote, then NOT A equals the truth-value A = TYPE A. The denial of a mote is 1 or 0 according as A is typical or not.

BOOLSUM must pervade (a copy of) a recursively packed array to the twigs and shade twigs and there replace such simple arrays by motes. The items of a twig are leaves just as the items of a simple array are motes. An array A is simple iff A equals EACH SINGLE A. If A is simple, then BOOLSUM A equals the truth value A UNEQUAL TYPE A. This definition is appropriate for Boolean summation because an atypical array must hold at least one atypical item. Simple, atypical arrays sum logically to 1. The remaining operations of propositional algebra are easily defined in terms of NOT and BOOLSUM.

Pair. Given arrays A and B as left and right arguments, the dyadic pairing operation results in the pair A PAIR B, which equals the two-item list A B that holds A as first item and B as second item. Pairing is the only primitive operation that can nest arbitrary arrays as items in a more deeply nested array. The aggregative effect of pairing is restricted to two items. The aggregation of more than two arbitrary items depends also on the uniting operation, which decreases rather than increases the depth of nesting. Pairing is therefore associated with the principle of nesting.

Sublist. Given arrays A and B as left and right arguments, the dyadic sublisting operation results in the list A SUBLIST B, which is the sublist of B got by striking from B those items that correspond in location to the typical items occurring in the result of reshaping A to the rectangularity of B. For example, 0 1 2 SUBLIST P Q R S T equals Q R T because 0 1 2 is reshaped to 0 1 2 0 1, which indicates that the first and fourth items are to be

struck from P Q R S T. Listing and vacating are defined in terms of sublisting. Listing corresponds to raveling in APL.

LIST B for $\underline{1}$ SUBLIST B
VACATE B for $\underline{0}$ SUBLIST B

A SUBLIST B equals each of the following:

(LIST A)SUBLIST LIST B
(A EACH UNEQUAL TYPE A)SUBLIST B
((SHAPE B)RESHAPE A)SUBLIST B.

(In the syntax of array theory, all parentheses in the above three lines are redundant [17, P.138].)

Sublisting corresponds to compression in APL and to the axiom schema of subsets or separation, which is Zermelo's repair of the axiom schema of comprehension discussed in Section 0. Sublisting introduces the capacity to select particular items from an array by matching the arrangement of items in the array with the arrangement of atypical items in another array.

The operation of picking [21, Sect. 7] is defined in terms of sublisting: A PICK B is the extract of the sublist of B that holds the item of B, if any, occurring at the location addressed by A. If A does not address a location in B, then this sublist of B is empty and A PICK B is the prototype of B. Otherwise A PICK B is the item of B addressed by A. Reference 21 shows how every array can be interpreted as an address or as a measure of rectangularity.

Reshape. If DNA appears to have the structure of an unoriented table, then why should tables and arrays be oriented? A major reason has to do with the transfer of data between different kinds of storage devices and between such devices and ourselves. Registers, disks, drums, tapes, memory chips, and direct access storage devices can be represented as lists, tables, and multivalent arrays. The items of these arrays are bits, lists of bits, such as bytes, or lists of bytes, such as half-words and words, which may also be construed as lists of bits. Data are usually transferred one bit, byte, half-word, or word at a time. This means that there is frequent need to list an array, reshape a list, reshape one array to another, or unite an array.

Given arrays A and B as left and right arguments, the dyadic reshaping operation results in the array A RESHAPE B, which has a rectangularity measured by A and holds in main order items taken cyclically in main order from B. The left argument of reshaping may be any array [21]. The nested rectangular structure of this array has no significance for reshaping. Only the leaves and the order of the leaves matter. The result of reshaping has as many axes as there are leaves in the left argument. An empty array, such as the null, has shade leaves [21] but no leaves. Thus singling can be defined as null reshaping:

SINGLE A for NULL RESHAPE(A PAIR A).

A RESHAPE B equals both of the following:

A RESHAPE LIST B
(SHAPE NUMERATE A)RESHAPE B.

This same principle of ignoring valency and levels of nesting applies also to the left argument of picking. For example, 2 and LIST 2 and SINGLE LIST 2 all measure the rectangularity of a pair and may be used as the left argument of reshaping to make a pair. Each of these three arrays can also be used to address the third item of a list. When construed as measures of rectangularity, the arrays 2(3 4) and (2 3)4 are equivalent to the primary measure 2 3 4, which is the shape of a 2-by-3-by-4 array. When construed as addresses, these two arrays are equivalent to the primary address 2 3 4.

It is necessary in transfinite array theory to take the EACH operator as primitive instead of the reshaping operation. This may also be done in the finite theory. Reshaping or replacement are needed to construct arrays of any given rectangularity holding repetitions of an array. By choosing reshaping to be primitive in the finite theory, one can avoid primitive operators, derive rather than assume the boundary condition for replacement (Sect. 6), and facilitate the rapid development of the theory in exposition.

Shape. How does one express the rectangularity of an array as an explicit measure? The first task is to determine the extent of each axis of the array. There are no numbers on a chessboard or in a segment of DNA. The extent of an axis, which refers to the number of positions available in the direction of the axis, is discovered by counting the items in any row parallel to the axis.

How does one count? By using physical devices, such as bullae of clay tokens [71] or the successive states of an odometer, to represent successive locations in a spatial or temporal list. In keeping with the physical motivations of array theory, let a finite ordinal number be a counter in a particular state, such as an electronic register in a stable configuration of currents. The understanding is that new counters can be manufactured and set to any particular state and that new components can be added to a counter whenever the need arises to handle big numbers. We use the numerals '0', '1', and '2' to mention the ordinals 0, 1, and 2, which are counters in the first, second, and third states.

The operation of counting the items in a row, such as the unique row in a list, proceeds as follows: Reset a counter to its first state, which is 0. The extent of the row, which is an intrinsic property of the row, is represented by the current state of the counter, called the length of the row,

after all items in the row have been counted. If there is no first item in the row, then there is no way to make a state of the counter correspond to an item in the row. The length of an empty row is 0 because the current state of the counter is 0. The extent of an axis is expressed by the length of the axis, which is the same ordinal number as the length of a row parallel to the axis.

The shape of a list is length of the unique row in the list. The shape of any array other than a list is the list of lengths for the axes of the array. The lengths in a shape are given in the same order as the axes. The shape of a single is the null, which is the list of no lengths.

The count of an array is the shape of the list of the array. Valence is the count of the shape. Thus valence is an ordinal that expresses the intrinsic property of valency, which has to do with the number of distinct axes. The valence of a single is 0.

Numerate. The shaping operation sends a geometrically arranged object to an arithmetic representation of the object. Numerating does the opposite by sending the shape or primary measure of a given array to the grid of the array, which is the array that has the rectangularity of the given array and holds at each location the primary address for the location [21, Sect. 9]. Numerating generates smooth arrays because the operation is constructed per-recursively in terms of the monadic Cartesian product, which causes all rows parallel to an axis to have the same length. [21, Sect. 9].

NUMERATE 2 2 equals the 2 2 reshape of the quadruple (0 0)(0 1)(1 0)(1 1). Each item of this quadruple can be used to construct a different sublist of the pair P Q. Thus

(NUMERATE 2 2)EACHLEFT SUBLIST P Q

is a 2-by-2 array that holds all sublists of P Q. This result corresponds to the axiom of powers, which posits the existence of the set of all subsets of any given set.

Unite. Uniting an array makes a list, called the union of the array, that holds all items of each item of the array [19, p.26]. This operation converts an array of bytes to a list of bits and converts an array of words, where each word is a list of bytes, to a long list of bytes. For example, UNITE(P Q Q)(R Q)(S R) equals P Q Q R Q S R. Repeated items are preserved in array-theoretic unions but have no significance in set-theoretic unions. When combined with pairing, uniting gives the capacity to aggregate. The union of a pair of pairs is a quadruple. The union of a quadruple of pairs is an octuple, etc.

Extract. Extracting is discussed in Section 5. The word "first," which connotes nonemptiness, is used interchangeably with the word "extract," which connotes the possibility of emptiness. These connotations are often useful in exposition. The operation EACH FIRST is like a choice function in that it selects a particular item, the first, from each item of an array of nonempty arrays [17, Sect. 17]. This corresponds to the axiom of choice, which is a theorem for finite sets.

Infinity. Transfinite array theory is created by introducing the mote lW, which is the first transfinite ordinal number (in array-theoretic notation). When applied to lW, the numerating operation results in the infinite list of the finite ordinals. This is the purpose of the axiom of infinity in set theory.

Foundation. The set-theoretic axiom of foundation prevents a set from containing itself at any deeper level of nesting. This principle needs to be modified in array theory because motes hold themselves. W. G. Bouricius partially implemented Version IV without any restrictions on loops of self-containment [18, p.54]. A weak form of foundation in Version III [17, p.145] was removed in Version IV [19, p.20].

8. Conclusion

Why represent data by nested rectangular arrays when nested lists are simpler in structure and have almost the same representational convenience? The first reason is that data are simple when viewed locally. A 2-by-1000 table, such as a segment of DNA, can be represented as a pair of lists, each of length 1000. But the adjacency of an item in the first row of the table to an item in the second row is changed to an adjacency in depth between two long lists. The global operation of aggregating two long lists intervenes in the immediacy of the adjacency.

A second reason is that nested lists cannot represent both nesting and valency by the usual method of representing a table as a list of lists. For example, a pair of triples of quadruples represents both a 2-by-3 table of quadruples and a pair of 3-by-4 tables.

The central position set theory occupies in modern mathematics is clearly stated by Gleason in his book on the Fundamentals of Abstract Analysis: "Today's mathematics is based on set theory. Every mathematical concept is described by sets and all mathematical relationships are represented by the interlocking membership relations between the various sets of what we shall call configurations. Mathematics is thus reduced, in a sense, to glorified combinatorial problems. While this approach is decried by some for making mathematics nonintuitive, it does, in fact, lead to a new kind of intuition which is indispensable in modern algebra and valuable in all of mathematics." [65]

In the process of abstracting sets from open sentences, all the principles in the organization of data are lost except those of aggregation and nesting. The other six principles can be represented but at too great a cost in complexity [66, 67, 39-42]. Array theory is conservative in that it makes only one change in the set-theoretic method of developing and applying a mathematics for a model of structured objects. The change is from the set to the array.

Appendix The reduction operator

The array-theoretic reduction operator REDUCE transforms an arbitrary dyadic operation F to the monadic operation REDUCE F, called the reduction transform of F. If the array A is neither empty nor singular, then REDUCE F A equals

(FIRST A)F(REDUCE F REST A).

For example, REDUCE F P Q R should equal P F(Q F R). This means that REDUCE F Q R should equal Q F R. But REST Q R equals SOLITARY R. Thus REDUCE F SOLITARY R should equal R, which is the first item of SOLITARY R. If A is empty or singular, then REDUCE F A equals FIRST A.

Monadic sum and product, Boolean sum and product, max and min, and unite and intersect are applied to pairs to define the corresponding dyadic operations. The reduction transforms of these dyadic operations give exactly the same results as the corresponding monadic operations when applied to arrays that hold two or more items, but not when applied to empty or singular arrays. For example, A PLUS B is defined as SUM A B, which equals REDUCE PLUS A B. However, SUM SINGLE A equals A PLUS 0 but REDUCE PLUS SINGLE A equals FIRST SINGLE A equals A. If a mote other than a number or the cipher (typical fault) occurs as a leaf within A, then A must differ from A PLUS 0 at that leaf. In particular, the addition of 0 converts a Boolean number to the corresponding integer. Monadic summing has an arithmetic effect on empty and singular arrays but the corresponding transform does not.

Similarly, A UNION B is defined as UNITE A B, which equals REDUCE UNION A B. UNITE SINGLE A equals LIST A, which is the sublist of A that holds all the items of A in main order. But REDUCE UNION SINGLE A equals FIRST SINGLE A equals A, which differs from LIST A when A is not a list. Monadic uniting has a listing effect on empty and singular arrays but REDUCE UNION does not. What one wants is a simple, universally valid formula that has mathematical meaning appropriate for each operation and applies to all reduction transforms at all empty and singular arrays. But the differences between summing, uniting, and other monadic operations appear to prevent such a formulation.

Acknowledgments

M. Schatzoff, R. Creasy, R. MacKinnon, H. Kolsky, and L. Robinson have encouraged this work on Version V of the theory. Although acknowledgments for the earlier versions are given in References 15 through 20, it is appropriate to mention here the early substantial influence and support of W. Bouricius, B. Dunham, A. Falkoff, K. Iverson, and J. McPherson. Version V evolved from the task of documenting Version IV in sufficient detail to support an experimental implementation by A. Hassitt, L. Lyon, and R. Creasy. Discussions with them, and with P. Penfield, M. Jenkins, P. Berke, N. Sorensen, and E. Kleban, helped in the transition to the present theory. I am also indebted to J. Brown, J. Michel, P. Marsh, V. Pratt, and A. Tritter for recent technical discussions. I am grateful to L. Rosenzweig and particularly B. Whitehill for considerable care in typing the manuscript and preparing camera-ready copy.

References and Notes

1. Davis, M., ed. The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions. Raven Press, Hewlett NY, 1965.

2. Kleene, S.C. Introduction to Metamathematics. D. von Nostrand Company, Inc. Princeton NY, 1952.

3. Curry, H.B. and Feys, R. Combinatory Logic. Vol. 1. North Holland Publishing Company, Amsterdam, 1958.

4. Church, A. The Calculi of Lambda-Conversion. Annals of Mathematics Studies No. 6. Princeton University Press, Princeton NY, 1941.

5. Markov, A.A. Theory of Algorithms. Vol. 42. Academy of Sciences of the USSR, Moscow, 1954. Translated and published for the National Science Foundation by The Israel Program for Scientific Translations, Jerusalem, 1961. Distributed by clearinghouse, U.S. Dept. of Commerce, TT60-51085.

6. Smullyan, R.M. Theory of Formal Systems. Revised ed. Annals of Mathematics Studies No. 47. Princeton University Press, Princeton NJ, 1961.

7. Rogers, H., Jr. Theory of Recursive Functions and Effective Computability. McGraw-Hill Book Company, New York, 1967.

8. Rabin, M.O. "Complexity of computations." Comm. ACM 20, 9 (Sep. 1977), 625-633.

9. Scott, D.S. "Data types as lattices." SIAM J. Computing 5, 3 (Sep. 1976), 522-587.

10. Scott, D.S. and Strachey, C. "Toward a mathematical semantics for computer languages." Proc. Symposium on Computers

and Automata, Polytechnic Inst. of Brooklyn, Vol. 21 (1971) 19-46. Also: Technical Monograph PRG-6, Oxford Univ. Computing Lab., Programming Research Group (1971).

11. Tennent, R.D. "The denotational semantics of computer languages." Comm. ACM 19, 8 (Aug. 1976), 437-453.

12. Burge, W.H. Recursive Programming Techniques. Addison-Wesley Publishing Company, Reading, Mass., 1975.

13. Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." Comm. ACM 21, 8 (Aug. 1978), 613-641.

14. Cartwright, R. and McCarthy, J. "First order programming logic." Sixth Annual ACM Symposium on Principles of Programming Languages, (Jan. 1979), 68-80.

15. More, T. "Notes on the development of a theory of arrays." Rep. 320-3016, IBM Scientific Ctr., Philadelphia, Pa., 1973.

16. More, T. "Notes on the axioms for a theory of arrays." Rep. 320-3017, IBM Scientific Ctr., Philadelphia, Pa., 1973.

17. More, T. "Axioms and theorems for a theory of arrays." IBM J. Res. Develop. 17, 3 (1973), 135-175.

18. More, T. "A theory of arrays with applications to databases." Rep. G320-2106, IBM Scientific Ctr., Cambridge, Mass., 1975.

19. More, T. "Types and prototypes in a theory of arrays." Rep. G320-2112, IBM Scientific Ctr., Cambridge, Mass., 1976.

20. More, T. "On the composition of array-theoretic operations." Rep. G320-2113, IBM Scientific Ctr., Cambridge, Mass., 1976.

21. More, T. "Nested rectangular arrays for measures, addresses and paths." ACM-STAPL/Sigplan Proceedings APL79 Conference, May, 1979.

22. Berke, P. "Tables, files and relations in array theory." Rep. G320-2122, IBM Scientific Ctr., Cambridge, Mass., 1978.

23. Berke, P. "Data design with array theory." Rep. G320-2123, IBM Scientific Ctr., Cambridge, Mass., 1978.

24. Hassitt, A. and Lyon, L.E. "Array theory in an APL environment." ACM-STAPL/Sigplan Proceedings APL79 Conference, May, 1979.

25. Heisenberg, W. "The meaning of beauty in the exact sciences." Munich, 1970. Translated by P. Heath. Reprinted in Heisenberg, W., Across the Frontiers, Harper & Row, Publishers, New York, 1975, p. 172.

26. Cohen, P.J. Set Theory and the Continuum Hypothesis. W. A. Benjamin, Inc., New York, 1966.

27. Kneale, W. and Kneale, M. The development of logic. Oxford, at the Clarendon Press, 1962, p. 439.

28. Fraenkel, A.A., Bar-Hillel, Y. and Levy, A. Foundations of Set Theory, 2nd Revised Ed. North-Holland Publishing Company, Amsterdam, 1973.

29. Iverson, K.E. A Programming Language. John Wiley and Sons, Inc., New York, 1962.

30. Falkoff, A.D., Iverson, K.E., and Sussenguth, E.H. "A formal description of System/360." IBM Syst. J. 3 (1964), 198-263.

31. Falkoff, A.D. and Iverson, K.E. APL/360 User's Manual. Thomas J. Watson Research Center, IBM Corp., Yorktown Heights, NY, July, 1968.

32. Falkoff, A.D. and Iverson, K.E. "The design of APL." IBM J. Res. Develop. 17, 4 (1973), 324-334.

33. Iverson, K.E. "The role of operators in APL." ACM-STAPL/Sigplan Proceedings APL79 Conference, May, 1979.

34. McCarthy, J. "Recursive functions of symbolic expressions and their computations by machine, Part I." Comm. ACM 3 (1960), 184-195. Reprinted in Saul Rosen, ed., Programming Systems and Languages. McGraw-Hill Book Company, Inc., New York, 1967.

35. Greenberg, B. Notes on the Programming Language LISP. M.I.T. Student Information Processing Board, 1976.

36. Pratt, V.R. "LISP." M.I.T. Laboratory for Computer Science. July, 1977.

37. Schwartz, J.T. "Abstract and concrete problems in the theory of files." Data Base Systems, Courant Computer Science Symposium 6, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972, pp. 1-21.

38. Kennedy, K. and Schwartz, J. "An introduction to the set theoretical language SETL." Comp. & Maths. with Appls. 1, 1 (1975), 97-119.

39. Warren, H.S., Jr. "Definition of the concept of 'vector' in set theoretic programming languages." Comp. & Maths. with Appls. 2 (1976), 73-83.

40. Childs, D.L. "Feasibility of a set-theoretic data structure: a general structure based on a reconstituted definition of relation." Proceedings of IFIP Congress 1968. North-Holland Publishing Company, Amsterdam, 1969, pp. 420-430.

41. Childs, D.L. "Description of a set-theoretic data structure." AFIPS Proceedings of FJCC 1968, Vol. 33, Part 1. The Thompson Book Company, Washington, D.C., 1968, pp. 557-564.

42. Childs, D.L. "Extended set theory: a general model for very large, distributed, backend information systems." ACM Proceedings of the Third International Conference on Very Large Databases, Tokyo, Oct. 1977.

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

44. Codd, E.F. "Relational completeness of data base sublanguages." Data Base Systems. Courant Computer Science Symposium 6, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972, pp. 65-98.

45. Codd, E.F. "A data base sublanguage founded on the relational calculus." ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, Nov. 1971, pp. 35-68.

46. Chamberlin, D.D. et al. "Sequel 2: A unified approach to data definition, manipulation, and control." IBM J. Res. Develop. 20, 6 (Nov. 1976), 560-575.

47. The American Heritage Dictionary of the English Language. Morris, W., ed. Houghton Mifflin Company, Boston, Mass., 1969.

48. Webster's New International Dictionary of the English Language, 2nd. ed. 1934 unabridged, G. & C. Merriam Company, Publishers, Springfield, Mass. 1958.

49. Gerretsen, J.C.H. Lectures on Tensor Calculus and Differential Geometry. P. Noordhoff, Groningen, The Netherlands, 1962.

50. Loeb, A.L. Space Structures: Their Harmony and Counterpoint. Addison-Wesley Publishing Company, Reading, Mass., 1976.

51. Misner, C.W., Thorne, K.S., and Wheeler, J.A. Gravitation. W.H. Freeman and Company, San Francisco, CA, 1973, p. 19.

52. Kleene, S.C. Mathematical Logic. John Wiley and Sons, Inc., New York, 1967.

53. Quine, W.V. "Unification of universes in set theory." Journal of Symbolic Logic 21, 3 (Sept. 1956), 267-279.

54. Quine, W.V. "New foundations for mathematical logic." American Mathematical Monthly 44, 70-80 (1937).

55. Quine, W.V. Mathematical Logic, Revised ed. Harvard University Press, Cambridge, Mass., 1951.

56. See References 15, p.3; 16, p.8; 17, p.136; 18, p.26; 19, p.11; and 20, p.4.

57. Church, A. Introduction to Mathematical Logic, Vol. 1. Princeton University Press, Princeton, NJ, 1956, p. 40.

58. Halmos, P.R. Naive Set Theory. D. van Nostrand Company, Inc., Princeton, NJ, 1960.

59. Berge, C. Topological Spaces. The Macmillan Company, NY, 1963.

60. I am indebted to E. Kleban, P. Feit, and M. Jenkins for technical discussions on recursive definitions in array theory and LISP.

61. Gull, W.E. and Jenkins, M.A. "Decisions for 'type' in APL." Sixth Annual ACM Symposium on Principles of Programming Languages, (Jan. 1979), 190-196.

62. Brown, J.A. "Evaulating extensions to APL." ACM-STAPL/Sigplan Proceedings APL79 Conference, May, 1979.

63. I am indebted to M. Jenkins and J. Michel for discussions on alternatives to types and prototypes.

64. Weizenbaum, J. Computer Power and Human Reason. W.H. Freeman and Company, 1976, P. 125.

65. Gleason, A. Fundamental of Abstract Analysis. Addison-Wesley Publishing Company, Reading, MA, 1966, p. 55.

66. Skolem, Th. "Two remarks on set theory." Math. Scand. 5 (1957), 40-46.

67. Fraenkel, A.A. Abstract Set Theory, 1st ed. North-Holland Publishing Company, Amsterdam, 1953, p. 172.

68. Brower, K. The Starship and the Canoe, Holt, Rinehart and Winston, New York, 1978, p. 21.

69. See Reference 17, p. 146. The end of Section 10 on page 146 states incorrectly that the replacement operator is an endomorphism. The error and repair are noted in Reference 19, p. 23.

70. Zermelo, E. "Untersuchungen uber die Grundlagen der Mengenlehre I." Math. Annalen 65 (1908), 261-281. English translation in van Heijenoort, J. From Frege to Godel. A Source Book in Mathematical Logic, 1879-1931. Harvard University Press, Cambridge, Mass. 1967, pp. 199-215.

71. Schmandt-Besserat, D. "The earliest precursor of writing." Scientific American, (June 1978) 50-59.