NESTED RECTANGULAR ARRAYS

FOR MEASURES, ADDRESSES, AND PATHS

Trenchard More

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

Abstract

The left argument of the reshaping function in APL is an ordinal number or list of ordinals, called a shape, that measures the rectangularity of an array. Shapes are the simplest measures. A natural extension of the index-generator function shows how an array of any valence (rank), depth of nesting, or type can be used as a measure. The method of generalizing shapes to measures also generalizes addresses and paths to arbitrary arrays. Only the leaves of a measure, address, or path matter, the nested rectangular structure does not. These results of array theory for floating arrays are motivated by similar considerations for grounded arrays.

0. Introduction

A topic of current interest to persons concerned with the enhancement of APL is the extension of the language from arrays of numbers and characters to arrays of elements that have more elaborate structure than numbers or characters [1-4].

After attempting without success to find a coherent, computationally efficient way to combine ordered n-tuples with sets in a one-sorted theory, the author began considering set-like and tree-like extensions to APL in the summer of 1967 and proposed a year later [5] that the items of an array should again be arrays without exception just as the elements of a set are again sets in Zermelo set theory. The nesting of arrays was to terminate in arrays that had no axes and held only themselves just as the nesting of sets can terminate, according to Quine, in self-containing unit sets [6].

This proposal began the development of a theory that was intended to do for arrays what set theory does for sets. The basic operations of set theory are defined for all sets in terms of relatively few primitive operations, return only sets as values, combine coherently in a substantial algebra, and support an axiomatic theory. The belief and hope was that the same would be true for certain basic operations on arrays. This hope has now been largely realized [7].

Versions I and II (1969-70) of a partial theory contained errors in the structure of empty arrays. Version III (1971-72) removed the errors but treated empty arrays in too restrictive a manner. Version IV (1973-76) probably reached the correct intuition about empty arrays but contained a subtle error in the application to empty arrays of dyadic operations defined by abstraction. Version IV still falls short of the goal of defining all primitive operations for all arrays without introducing special faults or arrays indicating error.

The present paper reports some results in Version V that lead to a deeper understanding of the measurement and addressing of arrays. These results show how the operations for numerating addresses, reshaping arrays, and picking items by address can be defined for all arrays.

Total definition is incompatible with the design of APL. For example, the dyadic Boolean and arithmetic operations of array theory are defined even when the left and right arguments are nonconformable. APL prefers to recognize the addition of vectors of different lengths as an error. An abstract theory of arrays may help to clarify some of the choices available in the extension of APL to nested arrays.

There is a premium on eliminating exceptions in the definition of the basic operations. Complexity of proof increases exponentially with the number of exceptions. In Versions I-IV, the reshaping operation ρ is defined for all right arguments and only for left arguments equal to the shape or size of the array. For all other left arguments, ρ returns a fault, which is an

array of a special type. To define ρ in terms of other operations, special constructions are needed to specify when error is to occur and which fault to return.

A more general concept of the measure of an array now permits ρ to be defined for all left arguments. The fault-free definition of ρ provides the option of taking ρ as a primitive operation or defining ρ directly in terms of other operations. The equations defining the properties of ρ may be taken as axioms or derived as theorems. This flexibility increases the number of identities in the algebra of operations on arrays.

As currently defined in VS APL, the reshaping function ρ returns a rank error if the left argument is any array having two or more axes, a domain error if the left argument is a nonempty vector or scalar holding components other than the nonnegative integers, a length error if the left argument is the empty vector of numbers or characters and the right argument is empty, and a domain error if the right argument is empty and the left argument is a nonempty vector or scalar holding positive integers.

When an unsuitable argument results in an error message, no further calculation is possible. Interruptions of this sort help in the interactive debugging of defined functions but are unwanted in an application program that is written in APL for users unfamiliar with APL, provided that errors are detected in some other way.

The reshaping operation factors many problems involving arrays into simpler problems involving lists, which are arrays that have but one axis. The first equations that were considered in the theory came from the attempt to axiomatize the properties of reshaping. The question that was asked then and is asked now is this: how does one define the reshaping operation meaningfully for all left and right arguments?

The reshaping of an empty array to the shape of a nonempty array requires one or more items to be generated from an array that holds no items. This problem led to the opaque discussion of patterns and plans in Version III and the surprising results of types and prototypes in Version IV, which appear to be correct and are kept in Version V. What should happen if the left argument of reshaping is a deeply nested, multivalent (many axes) array in which the leaves (the most deeply nested objects) are integers, or characters, or real numbers?

A clue, which was not interpreted properly until it was exposed by equations, can be found in the way APL extends the left argument of reshaping to include vectors as well as scalars. The meaning of the clue was recognized by considering a similar question for the monadic index-generating or numerating operation ι. What should happen if the argument of numerating is a deeply nested, multivalent array?

Some of the mixed functions of APL treat scalars and one-component vectors in the same way. For example, the left argument of reshaping is normally a vector holding zero or more nonnegative integers but may also be a scalar when there is only one such integer. Thus (,3)ρ*A* and 3ρ*A* give the same result. The index-generating function applies to nonnegative integer scalars and is extended to one-component vectors so that ι3 and ι,3 are the same.

Unlike the left argument of reshaping, the argument of monadic ι in APL may be any one-component array holding a nonnegative integer. Thus ι(1 1ρ3) and ι(1 1 1ρ3) also equal ι3. Otherwise ι returns a domain error. These design choices for APL recognize the convenience of disregarding the valence (number of axes) of a one-component array when used as an argument for certain functions. How can this convenience be given precise algebraic formulation?

In order to explain the results about the measurement and addressing of arrays, it will be necessary to discuss some of the properties of nested arrays. But here one encounters two different intuitions about how nesting terminates. Perhaps the majority of persons familiar with APL find the concept of an array that holds itself difficult to visualize or accept [1-3].

Many of the results of array theory appear to be alien from the viewpoint of APL because the equations of the theory begin with the general case, far from the boundary where nesting effectively terminates in self-containment. The theory is inherently recursive. Since the items of an array are always arrays, does the nesting of arrays ever stop? No. Arrays are infinitely deep in the sense that it is always possible to extract other arrays, even if the given arrays are empty. The theory is appropriately called a floating theory [2].

By contrast, APL is rooted firmly at the boundary where the components of an array are not arrays. Considered from the point of view of nested arrays, all functions of APL apply to a special subclass of arrays. When working at a boundary, it is difficult to determine which results can be generalized safely and which results are anomalous because of boundary conditions. Intuition gained from APL does not generalize reliably to a floating theory.

Indexing in APL corresponds to addressing in array theory but is too restrictive from the point of view of nested arrays. To bridge the differences between APL and array theory and to place the limitations of indexing in perspective, the paper begins with a discussion of grounded arrays. These are the primitive objects in a grounded theory of nested arrays.

The terminology of the grounded theory is drawn from APL and reflects a process of generalizing the basic arrays of APL to

nested arrays. Wherever usage permits, the terminology of the floating theory is kept distinct, not only from APL but also from linear algebra and set theory. A neutral terminology permits simultaneous discussion of array theory and set theory with linear algebra or with certain extensions of APL. Since this paper is written from the point of view of a floating theory, a grounded array is referred to as such while a floating array is called an array.

1. Grounded arrays

Each component of a grounded, rectangular, nested array is either another such array or an elementary object, such as a number or character, that is not a grounded array. A basic array is a grounded array in which all components are elementary objects. Basic arrays are visualized as occurring at ground level while elementary objects are thought to exist below ground because they are not available as arguments for operations on grounded arrays. APL, as currently implemented, represents data by basic arrays that can also be viewed as floating arrays of depths 0 and 1. [4, Sect. 2.1]

Matrices, vectors, and scalars are grounded arrays having two, one, and no axes, respectively. As a consequence of having no axes, a scalar must hold precisely one object, which always differs from the scalar. The components of a basic matrix, vector, or scalar are elementary objects and therefore not scalars. In a discussion of basic matrices, vectors, or scalars, the adjective "basic" is usually omitted when it is clear from context that every component is an elementary object.

2. Indexing by basic arrays

A vector *V* can be indexed (in 0-origin) by a basic array *A* of nonnegative integers to give a grounded array *V*[*A*] that has the same rectangularity (and hence shape) as *A* and holds certain components of *V*. For example, let *V* be the vector 5 6 7 that holds the integers 5, 6, and 7. Let 0 and 2 be the scalars that hold the integers 0 and 2. Then *V*[2] is the scalar 7 that holds the integer 7 and *V*[2 0 2] is the vector 7 5 7.

The point of the example is that functions in APL take basic arrays as arguments and return basic arrays as values. Numbers and characters help to explain the effect of APL functions but never occur as arguments or values. Because of the restriction to basic arrays, only vectors can be indexed in the manner described above.

A matrix *M* cannot be indexed by a basic array *A* to give a grounded array *M*[*A*] having the same rectangularity as *A*. Each component of *M* is located at the intersection of a row and a column, which have independent positions indexed by separate integers. The components of *A* cannot be pairs of integers because *A* is basic. Nor can *A* have twice as many components as *M*[*A*] by hypothesis.

This lack of similarity to the indexing of vectors is partly circumvented by indexing a matrix *M* with basic arrays *A* and *B* of row indices and column indices to give a grounded array *M*[*A*;*B*] holding components of *M* and having the same rectangularity as the Cartesian product of *A* with *B*. *M*[*A*;*B*] has as many axes as there are axes of *A* plus axes of *B*. The Cartesian product of *A* with *B* is not basic because each component is a two-component vector holding an integer from *A* followed by an integer from *B*.

It is also the case that a scalar *S* cannot be indexed by a basic array *A* to give a grounded array *S*[*A*] having the same rectangularity as *A*. The sole component in *S* is located relative to no axes. An appropriate address for a scalar corresponds to the empty vector ⍳0 of numbers. Again, the components of *A* cannot be empty vectors because *A* is basic. Nor can *A* be empty because the intent is to allow the construction of a nonempty grounded array *S*[*A*] of components selected from *S*. The restriction to basic arrays prevents APL from indexing scalars in a manner similar to the indexing of vectors. The current solution in APL is not to index scalars at all.

Indexing illustrates the monadic operation ⍪ of carting for array-theoretic Cartesian products. Let *Q* be a quadrivalent basic array and let *A*, *B*, *C*, and *D* be basic arrays of indices appropriate for axes 0, 1, 2, and 3 of *Q*, respectively. This means that if axis 0 of *Q* has length 3, then every component of *A* is an ordinal less than 3. In particular, let *A* be the vector 2 0 1 1 2 of length 5. Let *B* be the scalar 4. Let *C* be a 2-by-3 matrix in which 6 is the first component. Let *D* be a 7-by-1-by-9 basic array in which 8 is the first component.

Finally, let *S* be the cart ⍪*A B C D* of the vector *A B C D* of length 4. Then *S* is a grounded array having six axes of lengths 5, 2, 3, 7, 1, and 9. The array *Q*[*A*;*B*;*C*;*D*] got by indexing *Q* has the same six axes in the same order. Each component of *S* is a vector of length 4 in which the first, second, third, and fourth components come from *A*, *B*, *C*, and *D*, respectively. The first component of *S* is the vector 2 4 6 8. Vector *A* contributes the first axis of *S*. Scalar *B* has no axes and so contributes no axes to *S*. The number of axes in *S* is the total number of axes in *A*, *B*, *C*, and *D*.

At each location in *Q*[*A*;*B*;*C*;*D*] there occurs the component of *Q* addressed by the vector at the same location in *S*. The array *Q*[*A*;*B*;*C*;*D*] is never a component of *Q* but rather a grounded array that holds components of *Q*. Versions I-IV extend the indexing operation to a locating operation that defines *Q*[*W*] for every array *W* in which all items are legitimate addresses for *Q*. As a special case, *Q*[⍪*A B C D*] is the same as *Q*[*A*;*B*;*C*;*D*]. In Version V, the array *Q*[*W*] is defined for all arrays *Q* and *W*.

Monadic Cartesian products are important because they reflect the essential nature of

rectangular arrangement. The cart of a vector of two or more nonempty basic arrays is relatively easy to visualize. But what is the cart of a basic array in which the components are not grounded arrays but elementary objects? These boundary conditions are compared in Section 8 with those for arrays.

3. Representation of tensors

The distinctions between a tensor, the array of numbers that represents the tensor, and a number help to explain the difference between floating and grounded arrays. The concept of a basic array is drawn primarily from the representation of a vector by a list of components, a linear transformation by a matrix or table of coefficients, and a tensor of valency N by an array of valency N with components arranged on N axes. The components and coefficients are numbers that belong to a field of coefficients. A tensor of valency N is a multilinear form or function that is linear in each of N vector arguments.

An invariant tensor of valency 0 is called a scalar and is given by a rule that sends every frame of reference to the same number, which is the unique component of the scalar. The difference between a scalar and its component is the difference between a function and a number. This distinction is blurred by the usual practice of referring to coefficients as "scalars."

A scalar in the tensor sense is represented by an array that has no axes and holds precisely one component. A tensor, such as a scalar, differs from its components and also differs from its representation as an array. But these two differences do not require the component of the scalar to differ from the one-component, nilvalent array that represents the scalar!

There are two choices. The first identifies the component of a tensor of valency 0 with the array that represents the tensor, treats elementary objects as nilvalent arrays that hold only themselves, and is consistent with the proposal that the items of an array should again be arrays without exception. The second choice distinguishes the component of a tensor of valency 0 from the nilvalent array that represents the tensor, separates a universe of elementary objects from the universe of basic arrays, and is consistent with the concept that a grounded array may hold as components other grounded arrays as well as elementary objects that are not grounded arrays.

4. Indexing versus picking

There is no operation in APL that can return as value a particular component of a basic array because all components of basic arrays are elementary objects outside the universe of discourse. The equivalent effect of such an operation is achieved in APL by the indexing function, which can return as value the basic scalar that holds the desired component. In array theory, the picking operation uses its left argument as an address to pick a particular component from the right argument. The value returned is the component itself, not a grounded array holding the component.

Let the depth of a grounded array be one greater than the depth of the component having the greatest depth. If elementary objects are given depth 0, then all basic arrays have depth 1. The arguments and values of every APL function have depth 1. The extension from basic arrays to grounded arrays permits the definition of operations that change depth. There is no problem with an operation that increases depth because the value returned is always a grounded array. But the picking operation, by decreasing depth, may return as value either a grounded array or an elementary object.

A theory is more tractable as an algebra when the values returned by any operation are suitable arguments for all operations. If the operations of the grounded theory are defined only for grounded arrays, then some sort of error must result when the operations are applied to elementary objects. To achieve such tractability in the grounded theory, either the definition of all operations must be extended to the domain of elementary objects or the picking operation must be modified so as to return certain kinds of grounded arrays instead of elementary objects.

Definition of operations on two distinct universes results in a two-sorted theory that has greater complexity than a one-sorted theory. It is preferable but not desirable to modify the picking operation. The way out of this dilemma is to take a different approach. The simplicity of a theory depends more on the simplicity of the operations and equations than on that of the primitive objects. For example, Maxwell's equations are simpler than the electromagnetic field of an atom. Instead of modifying operations to match the boundary conditions imposed by primitive objects, why not modify the primitive objects to match the boundary conditions imposed by the operations? The latter choice leads to self-containing arrays.

5. Intrinsic properties of arrays

The change from a grounded to a floating theory -- from the modification of operations to the modification of primitive objects -- is indicated by a change in terminology: grounded array to array, component to item, matrix to table, vector to list, scalar to single, elementary object to mote, and basic array to simple array. The items of an array are arrays. Tables, lists, and singles are arrays having two, one, and no axes. By virtue of having no axes, a single must hold just one item. A single that holds itself is called a mote. An array is simple iff (if and only if) all its items are motes. A mote is simple

because its item is the mote itself.

The parts of an array are itself, its items, its pieces (items of items), etc., which occur on successively deeper levels of the array. The array itself occurs at a unique site on level zero. Items occur on level one at sites called locations. A location, like a point in a multidimensional space, is determined by an ordered collection of positions, like coordinates -- one position on each axis of the array. Pieces occur at sites on level two. A site on level two is equivalent to a location within a location. Items of pieces occur at sites on level three, etc. Every part of an array is again an array.

Any part of an array may be empty. An empty part contributes no parts other than itself to the array because the empty part has no items or pieces and cannot be a mote. However, an empty part does have a prototype, which may be any typical array. The parts of the prototype contribute to the structure of the empty part in a way that is explained below. The prototype of an empty array amounts to a typical item and is visualized as if it occurred on level one even though it is not an item.

An array is typical iff it equals its type. An empty array is always typical because it holds no items and has only a prototype, which is typical. The type of a nonempty array is determined by replacing each of its items by the corresponding typical item without changing the rectangularity of the array. The determination of type is therefore recursive and has the effect of replacing every part of an array by the corresponding typical part without changing the structure of the array.

Emptiness of a part *D* of an array *A* shades the parts of the prototype *P* of *D* from inclusion as parts of *A*. Yet the parts of *P*, which are all typical arrays, contribute to the structure and type of *A*. Hence the parts of *P* are called shade parts of *A*. For example, *P* itself is a shade part of *A*. All parts and shade parts of any shade part of *A* are shade parts of *A*. Every shade part of an array is again an array.

The nesting of parts and shade parts within an array effectively terminates at various levels wherever motes are first encountered. Such motes are the most deeply nested parts and shade parts of the array and are called leaves and shade leaves. A leaf of an array is not part of any empty part of the array; a shade leaf is part of at least one empty part. Every array has at least one leaf or shade leaf. Empty arrays have no leaves, only shade leaves.

The level of the deepest leaf or shade leaf is the deepest level of the array. Since a mote is its own leaf, the deepest level of a mote is level zero. The deepest level of a simple array other than a mote is level one. In general, the deepest level of an array is one greater than that of the prototype or item having the deepest level.

Integers, Boolean numbers, and characters are three of the many different types of motes that may occur as leaves within an array. The integer zero, Boolean zero, and blank are the corresponding typical motes, which may occur as leaves or shade leaves. The type of an array is determined by replacing every leaf by the corresponding typical leaf without changing the structure of the array. For example, the type of a list of five integers is the unique list of five integer zeros.

Shade parts prevent a discontinuity of structure at the boundary of emptiness. For example, let *A* be a list that represents a file of sales records. Each record or item of *A* is a list of lists in which level two is deepest. Imagine that *A* is updated by inserting and removing such records. Level three of *A* is deepest as long as *A* holds at least one item. Must this deepest level change when the one remaining item in *A* is removed before another is inserted? No. The type of the first item of nonempty *A* is taken to be the prototype of the empty updated version of *A*. Since level two is deepest in the prototype, level three of *A* remains deepest even when *A* is empty.

The arrangement of items for an array is rectangular in the sense that every row of items parallel to one of the axes for the array has the same extent, which is taken to be the extent of the axis. Different axes may have different extents. The ordering of the axes and the extent of each axis together constitute the intrinsic property of rectangularity.

The following are intrinsic properties of an array: the zero or more axes, the zero or more positions on each axis, the extent of each axis, the valency or total number of axes, the rectangularity, the total extent of the listing in main order of all items of the array, the zero or more locations that project to a position on each axis, the one or more levels of the array, the deepest level, and the one or more sites that occur on successively deeper levels.

6. Representation of intrinsic properties

Intrinsic properties are not themselves arrays. Nor is the concept of counting on which the properties are based. Such properties and concepts cannot be manipulated in a calculus of arrays until they are represented explicitly as arrays. The representation depends on the (finite) ordinals 0, 1, 2, ..., which are introduced into the universe of arrays as motes. An ordinal is therefore a nilvalent array that holds itself. From this purely structural point of view, an ordinal amounts to a unique symbol given by a mote. How do the ordinals represent the concept of counting?

Ordinals are well ordered in the sense

that one ordinal precedes all others in any nonempty collection of ordinals. For example, given 5, 0, and 2, zero comes first. Given 5 and 2, two comes first, etc. The primitive numerating operation ⍳ in array theory, which corresponds to the 0-origin index generating function in APL, sends any given finite ordinal to the list of all preceding ordinals in ascending order. Thus ⍳4 equals 0 1 2 3.

The significance of the numerating operation is that it represents the concept of counting by changing, in effect, a linguistic token for an ordinal, such as the mote 4, into a geometric representation of the ordinal, such as the list 0 1 2 3. No ordinals precede zero. Consequently ⍳0 is the null ⍬, which is the unique empty list of ordinals. Emptiness represents zero.

Representation of intrinsic properties is illustrated by the following example. An array with four axes has valency four. This valency can be represented by any array that is sent to the ordinal 4 by a mapping of the universe of arrays onto a subuniverse that includes the ordinals. The mapping is given by a composition of operations. The representation is adequate because the numerating operation sends 4 to the list 0 1 2 3. Of all the arrays that represent the valency four, the ordinal 4 is the simplest in structure because it is a mote. Motes have the fewest axes (none) and are unique in having the fewest levels (one).

Axis numbers, indices, lengths, counts, valences, level numbers, and depths are ordinals and therefore arrays. The simplest intrinsic properties of an array are represented as follows: each axis is represented by an axis number, each position by an index, each extent by a length, the total extent by the count, the valency by the valence, each level by a level number, and the deepest level by the depth. Since all manipulations and descriptions of arrays are constructed in the universe of arrays, it is customary to refer to the intrinsic properties of an array by means of the arrays that represent the properties.

7. Primary arrays, measures, addresses

Because of the greater simplicity of motes and singles relative to lists, and of lists and tables relative to arrays with many axes, there is a premium on minimizing the depth and valence of an array. "Apovalence" means "tending away from valence." An array is apovalent iff its items are arranged in a way that requires the fewest number of axes. An array is univalent iff its items are arranged on one axis. The suit ⍪*A* and list ,*A* of an array *A* are, respectively, the apovalent and univalent arrays that hold the items of *A* in main order. An array *A* is a suit or list according as *A* equals (is the same array as) ⍪*A* or ,*A*.

An array is said to be empty or singular according as it holds no items or precisely one item. Suits are singles if singular or lists if nonsingular. An array that uses fewest axes and levels to encompass zero or more ordinals as leaves is called primary. A primary array is therefore a suit of ordinals. All nonsingular primary arrays, such as ⍬ and 2 3 and 0 5 0, have depth 1 and valence 1. A discontinuity in depth and valence occurs in the case of singular primary arrays, which are ordinals of depth 0 and valence 0. When self-holding, apovalent arrays also tend away from depth.

The concept of simplest structure has conflicting criteria: lack of discontinuity versus fewest levels and axes. Both criteria are useful. Array-theoretic operations are usually easier to apply when there are fewest discontinuities. Intrinsic properties of arrays are usually easier to express and represent as arrays when there are fewest levels and axes. The operations of listing and suiting provide ways to represent both criteria as arrays.

Of all the arrays that represent the rectangularity of *B*, let ⍨*B* be the primary array that measures *B*. This array, which is the counterpart of the size of *B* in APL, is called the shape or primary measure of *B*. Each item of ⍨*B* is an ordinal, called a length, that represents the extent of the corresponding axis for *B*. An array *P* is the primary measure of *B* iff *P* equals the shape ⍨*B* of *B*. How does one determine whether or not an array *M* is a measure of *B*?

The composition ⍨⍳ of shaping ⍨ with numerating ⍳ is a monadic operation called gaging. Gaging sends every ordinal to itself. For example, ⍨⍳4 equals ⍨0 1 2 3 equals 4. Versions I-IV define ⍳ so that gaging sends every primary array to itself. Thus ⍨⍳⍬ equals ⍬ and ⍨⍳2 3 equals 2 3. The problem is to extend ⍳ to all arrays so that gaging maps the universe of arrays onto the primary arrays in a way that is useful and algebraically tractable. Some observations can be made about the desired properties of gaging before the definition of ⍳ is known.

Primary arrays are the only arrays that are fixed under gaging. Thus an array *A* is primary iff *A* equals ⍨⍳*A*. This is a criterion for fewest levels and axes. The gaging operation is idempotent because it results in a primary array. Thus ⍨⍳*A* equals ⍨⍳⍨⍳*A* for all *A*. It is understood that "for all *A*" means "for every array *A*" because there is only one universe of discourse.

The gage of an arbitrary array *M* results in a primary array ⍨⍳*M* that can be used as a primary measure to represent the rectangularity of an array *B*. The reshaping of *B* to an array measured by ⍨⍳*M* results in the array (⍨⍳*M*)⍴*B* of shape ⍨⍳*M*. Now extend the reshaping operation to all arrays by defining *M*⍴*B* to equal (⍨⍳*M*)⍴*B* for all *M* and *B*. Then the shape ⍨(*M*⍴*B*) of the array *M*⍴*B* equals the gage ⍨⍳*M* of *M* for all *M* and *B*.

The test for determining when one array measures another is the following: an array M is a measure of an array B iff the gage $\sim\iota M$ of M equals the shape $\sim B$ of B. Gaging sends a measure to the corresponding primary measure. It follows for all M and B that M measures B iff $M\rho B$ equals B. Proof: If M measures B, then $\sim\iota M$ equals $\sim B$ and $M\rho B$ equals $(\sim\iota M)\rho B$ equals $(\sim B)\rho B$ equals B. Conversely, if $M\rho B$ equals B, then $\sim B$ equals $\sim(M\rho B)$ equals $\sim\iota M$.

Reference to an array J as an item indicates that J bears the array-theoretic itemhood relation ϵ to another array, which is determined by context to be an array B. Thus $J\epsilon B$ is true or false according as J does or does not occur as an item of B. Similarly, reference to A as an address indicates that A represents a location for another array, which is determined by context to be an array B. How can this relation be expressed within the theory?

Addresses have the same intrinsic properties as measures. The gage of an address is a primary address just as the gage of a measure is a primary measure. An array P is a primary address for an array B iff P is a primary array in which each item, called an index, is an ordinal less that the corresponding item or length in the shape $\sim B$ of B. Each index in P represents a position on an axis of B. A primary address for a single is a list of no indices just as the primary measure for a single is a list of no lengths. Both of these empty lists equal the null, which is the unique empty list of ordinals. A single or nilvalent array holds precisely one item because it has the null as unique primary address.

The numerating operation sends a primary array, which is the shape $\sim B$ of some array B, to the array $\iota\sim B$, called the grid of B, that has the same shape as B and holds every primary address for B. Thus $\sim\iota\sim B$ equals $\sim B$ for all B. Each item of $\iota\sim B$ addresses itself. For example, $\iota\sim R\ S\ T$ equals $\iota 3$ equals 0 1 2. An array P is a primary address for B iff $P\epsilon\iota\sim B$. An array A is an address for B iff the gage $\sim\iota A$ of A is an item in the grid $\iota\sim B$ of B.

If A is an address for B, then the array $A\supset B$, called the A pick of B, is the item of B that occurs at primary address $\sim\iota A$. Otherwise, $A\supset B$ equals the prototype of B. The left argument of picking extends from primary address to address in the same way that the left argument of reshaping extends from primary measure to measure. Thus $A\supset B$ equals $(\sim\iota A)\supset B$ for all A and B.

The each operator, which is designated by a colon, transforms a monadic operation Δ to the monadic operation $:\Delta$ that applies Δ to each item of an array. $\sim:\Delta\ A$ equals $\sim(:\Delta)A$ equals $\sim A$ for all A. The each-left operator $\dot{:}$ transforms a dyadic operation Δ to the dyadic operation $\dot{:}\Delta$ that combines each item of the left argument by Δ with the right argument. For example, $\dot{:}\supset$ is the locating operation. $P\ Q\dot{:}\supset B$ equals the pair $(P\supset B)(Q\supset B)$. The array $B[A]$, called B sub A, equals $A\dot{:}\supset B$ by definition.

8. Cartesian products

Given arrays A and B, the single $\circ A$ is the nilvalent array that holds A alone while the pair $A;B$ is the list $A\ B$ that holds A before B. Singles and pairs are apovalent. $:\Delta\circ A$ equals $\circ\Delta\ A$ and $:\Delta(A;B)$ equals $(\Delta\ A);(\Delta\ B)$ equals $(\Delta\ A)(\Delta\ B)$ for all A and B and every monadic operation Δ. An array A is mote or simple according as A equals $\circ A$ or $:\circ A$. A mote M is simple because M equals $\circ M$ equals $\circ\circ M$ equals $:\circ\circ M$ equals $:\circ M$.

The uniting operation $\cup$ lists the pieces (items of items) of an array. For example, let D be an array of any valence that holds the arrays W, X, Y, and Z in main order. The valences of the items of D do not matter. Let W hold P, Q, and R. Let X hold S alone. Let Y be empty. Let Z hold T and R. Then $\cup D$ equals the list $P\ Q\ R\ S\ T\ R$. The union $\cup\circ A$ of the single of any array A equals the list $,A$ of the pieces of $\circ A$.

As argued in Section 2, the shape $\sim\underline{\times}A$ of the Cartesian product of any array A equals the apovalent union $;\cup:\sim A$ of the shapes of the items of A. Since shapes are suits and the suit of a list is a suit, $\sim\underline{\times}\circ A$ equals $;\cup:\sim\circ A$ equals $;\cup\circ\sim A$ equals $;,\sim A$ equals $;\sim A$ equals $\sim A$ for all A. The cart of a pair is an array of pairs. The cart $\underline{\times}\circ A$ of the single of an array A equals the array $:\circ A$ holding singles and having the same shape as A. Thus the cart of a mote M equals the mote itself because $\underline{\times}M$ equals $\underline{\times}\circ M$ equals $:\circ M$ equals M.

Every item in $\underline{\times}A$ has the same shape as A itself and holds one item from each item of A. Since an array $:\circ A$ of singles contributes no axes to a Cartesian product, $\underline{\times}:\circ A$ equals $\circ A$ for all A because $\circ A$ is the nilvalent array that holds the item A having the same shape as $:\circ A$ and holding one item from each item of $:\circ A$. The cart of a simple array S equals the single of S because $\underline{\times}S$ equals $\underline{\times}:\circ S$ equals $\circ S$.

It is reasonable to assume in the grounded theory that every component in the cart $\underline{\times}A$ of a grounded array A has the same shape as A and holds one component from each component of A. A basic scalar 2 holds the elementary object $\underline{2}$ just as the basic pair 2 3 holds $\underline{2}$ and $\underline{3}$. Elementary objects are presumed to contribute no axes to a Cartesian product. $\underline{\times}2$ is the scalar that holds an object B having no axes (like 2) and holding, fictitiously, one component from $\underline{2}$. If B is identified with $\underline{2}$, then $\underline{\times}2$ equals 2 in the grounded theory as in the floating theory.

$\underline{\times}2\ 3$ is the scalar that holds an object P having the shape of a basic pair (like 2 3) and holding, fictitiously, one component

from 2 and one component from 3. If P is assumed to be 2 3, then x2 3 equals ∘2 3 in the grounded theory as in the floating theory. But then B above should be identified with 2, which requires 2 to hold itself and x2 to equal ∘2 instead of 2. This distinction is not necessary in the floating theory because 2 equals ∘2. Self-holding arrays handle the termination of nesting automatically in the floating theory.

9. Numerating

The grid ⍳~B of an array B equals the cart x:⍳~B of the result :⍳~B of numerating each length in the shape ~B of B. For example, ⍳2 3 equals x:⍳2 3 equals x(⍳2)(⍳3) equals x(0 1)(0 1 2) equals the 2-by-3 array that holds the six primary addresses in the list (0 0)(0 1)(0 2)(1 0)(1 1)(1 2). It is clear in Versions I-IV that ⍳A should equal x:⍳A for every primary array A. How does numerating apply to every array?

The insight that is needed to reach Version V is the recognition that ⍳A can equal x:⍳A for all A. What is the shape of ⍳A? In other words, what is the gage of A? The answer follows from the law that :∪x:xA equals x∪A for all A, which is the general associative law for carting.

The gage ~⍳A of A equals ~x:⍳A equals ~x:(x:⍳)A equals ~x:x::⍳A because the each operator distributes over compositions involving structural operations like carting. The last result equals ~:∪x:x::⍳A because ~B equals ~:Δ B for all B. The gage of A then equals ~x∪::⍳A by associativity. But ∪::Δ B equals :Δ∪B for all B and every monadic operation Δ. Thus the gage of A equals ~x:⍳∪A equals ~⍳∪A for all A. Let the content ⊍A of A be the result ∪∪∪...∪∪A of uniting enough times to flatten A to depth 1. The content of A is the list ⊍⍳A that holds only the leaves of A. Thus ~⍳A equals ~⍳∪∪A equals ~⍳⊍A for all A.

The gage of A equals the apovalent union of the list holding the gages of the leaves of A because ~⍳⊍A equals ~x:⍳⊍A equals ⍪∪:~:⍳⊍A equals ⍪∪:(~⍳)⊍A for all A. Leaves are motes, such as ordinals and characters. The equality of ⍳A with x:⍳A applies ⍳ recursively down to the leaves of A. What happens when ⍳ applies to a mote? ⍳∘A equals x:⍳∘A equals x∘⍳A equals :∘⍳A for all A. If M is a mote, then ⍳M is a simple array because ⍳M equals ⍳∘M equals :∘⍳M. For example, ⍳3 equals 0 1 2 equals (∘0)(∘1)(∘2) equals :∘0 1 2 equals :∘⍳3.

The numerate of an ordinal is a list in which every item is also an ordinal and the prototype or first item is the typical ordinal. Let the same principle be true for all motes. Then the prototype and every item in ⍳M has the same type as M, the first item in ⍳M equals the type of M, and ⍳M equals ,⍳M for every mote M. It follows for all A that the prototype and every item in ⍳A has the same type as A and the first item of ⍳A equals the type of A. Unlike APL, ⍳,3 equals (,0)(,1)(,2) because ⍳,M equals x:⍳,∘M equals x,∘⍳M equals :(,∘)⍳M for every mote M. Similarly, ⍳(1 1⍴3) equals (1 1⍴0)(1 1⍴1)(1 1⍴2).

Although the Boolean numbers o and ı differ in type from the ordinals 0 and 1, arithmetic operations apply to Boolean numbers as in APL because gaging sends every mote to an ordinal. Thus ~⍳o equals 0 because ⍳o is the empty list of Boolean numbers just as ⍳0 is the null. Similarly, ~⍳ı equals 1 because ⍳ı equals ,o just as ⍳1 equals ,0. If M is any mote other than an ordinal or Boolean number, then the gage ~⍳M of M equals 0 because ⍳M equals the empty list having the type of M as prototype. ⍳E equals ∘E for every leafless array E.

An array A is a path to a site in an array B iff the epigage ⍪:(⍳~)A of A is a suit of primary addresses to successively deeper locations in B. If A is empty, then A is a path to B itself. For example, if B is the triple C D E and E is the triple P(∘Q)R, then ⍬ and 2 and 2 1 and 2 1 ⍬ are paths to A and E and ∘Q and Q, respectively.

Acknowledgments

M. Schatzoff, R. MacKinnon and L. Robinson have encouraged this work. I am indebted to P. Berke, R. Creasy, R. Frank, A. Hassitt, M. Jenkins, E. Kleban, L. Lyon, J. Rubin, N. Sorensen, and particularly P. Penfield for technical discussions.

References

[1] G. Foster, Summary of the session on general arrays at the May, 1976 Queen's University APL Workshop, APL Quote Quad 7, 4 (Winter 1977), 11-13.

[2] M.A. Jenkins and T. More, Summary of the session on general arrays at the Sept., 1977 Syracuse University APL Workshop, APL Quote Quad 8, 2 (Dec. 1977), 12-13.

[3] K.E. Iverson, "Operators and functions," Research Report RC 7091, IBM Research Center, Yorktown Heights, NY, Apr. 26, 1978.

[4] W.E. Gull and M.A. Jenkins, "Recursive data structures in APL," Comm. ACM 22, 2 (Feb. 1979), 79-96.

[5] T. More, "Notes on the development of a theory of arrays," Tech. Rep. 320-3016, IBM Scientific Ctr., Philadelphia, PA (1973), 6.

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

[7] T. More, "The nested rectangular array as a model of data," invited address, APL79 Conference, Rochester, NY, May 30, 1979.