OPERATORS IN AN APL

CONTAINING NESTED ARRAYS

by M. A. Jenkins and Jean Michel

Abstract

Formal definitions of several functions and operators for manipulating nested arrays are presented. Three operators that control the application of a function to the items or groups of items of an array are defined. In addition, the definition of current APL operators is extended to nested arrays. A proposal is presented to accept some of the ideas for the present APL and others for an APL extended to include heterogeneous nested arrays.

Introduction

Adding nested arrays to APL* will require both the extension of some existing functions and operators ("functionals") of APL and the addition of new ones. Proposals for system 0 arrays appeared in Brown's paper [1] and in Ghandour and Mezei's article [2] -- the latter largely based on ideas of More [3]. Gull and Jenkins [4] present a smaller set of proposals for system 1 arrays, concentrating mainly on functions to manipulate nesting levels and selection. We present a systematic development of a unified collection of operators and functions for system 1, based on the concept of symmetry of definition with respect to the axes. In addition, we describe a restricted form of the proposal that is compatible with present APL. The discussion is for the universe $\cup_1$ of heterogeneous arrays without tags [4], leaving the extension to tagged generalizations (systems A and B in [6]) for later study.

This paper summarizes the results of a study of the structure of APL arrays and of the APL functions and operators carried out as part of a language design project that is still underway.

*Proposals for nested arrays in APL differ in their treatment of the relation between basic scalars and items of basic arrays. In system 1, the items of a basic array, including the item of a basic scalar array, are numbers or characters and are outside the universe of arrays. In system 0, a number or character is treated as an array that contains itself as its sole item. In set theoretic terms, the arrays of system 0 are equivalence classes of the arrays of system 1 under the smallest equivalence relation satisfying that

- a scalar array is equivalent to the array of rank zero that contains it as sole item, and
- two arrays are equivalent if they have the same shape and if corresponding items are equivalent.

The classification of the proposals for nested arrays is discussed by Gull and Jenkins [4].

1. Preliminary Definitions

First we present a definition of $\cup_1$, the universe of nested arrays with numbers and characters as their basic elements. The development follows that given in [4]. For any set S, let

$$S* = \cup_{k\geq 0} S^k$$

where S^k denotes the union of tuples of length k with components that are elements of S. Thus, $S*$ is the union of all such tuples of finite length. A $t \in S$ is a tuple belonging to some S^k, and we denote its length by $\underline{\text{len}}(t) = k$.

For any set D, we define the set of arrays with items in D by

$$A(D) = \{(s,d) \in \mathbb{N}* \times D* \mid \text{len}(d) = \prod_{i=1}^{\text{len}(s)} s_i\}$$

where $\mathbb{N}=\{0,1,2,\ldots\}$, the set of nonnegative numbers.

If x=(s,d) is an array, we define these terms:

the items of x = components of d
the shape of x = s
the list of x = d
the dimensions of x = components of s
the rank of x = len(s)
the count of x = len(d)

Let B be the set of basic (or simple) data, namely the set of numbers and characters of APL. Then $A(B)$ is a set of arrays that contains the arrays of current APL.

For example,

```
      ⍳5 = (1,(1,2,3,4,5))
2 2⍴'ABCD' = ((2,2,('A','B','C','D'))
       3 = (⍬,3)
```

where ⍬ denotes the empty set. This definition of arrays includes only one empty array of each shape. The array

```
(2,('A',5))
```

is a member of $A(B)$ but is not a valid APL object in current systems.

We may define arrays over other sets with the above definition; indeed, our purpose here is to define a universe of arrays with items that may themselves be arrays. For example, the set $A(B \cup A(B))$ is the set of arrays with items that are numbers, characters, or arrays of numbers or characters. Applying this process iteratively, we define the sequence of sets

$$L_0 = \phi$$

$$L_{i+1} = A(B \cup L_i) \text{ for } i=0,1,2,\ldots$$

Then $L_0 \subset L_1 \subset L_2 \subset \ldots$ and we define U_1 to be the least upper bound of this sequence:

$$U_1 = \bigcup_i L_i$$

Before proceeding to our development of new operators, we introduce some additional terminology. $L_1 = A(B)$ is the set of basic arrays. For any $x \in U_1$ we say that x is

a scalar if rank(x)=0
a vector if rank(x)=1
a matrix if rank(x)=2.

A scalar that is an element of L_1 is called a basic scalar, similarly basic vector and basic matrix. The set of basic scalars is in one-to-one correspondence with B by the mapping that sends a scalar to the item it holds. An array x is empty if count(x)=0. We denote the empty vector by ⍬.

If x has shape $s=(s_1,\ldots,s_r)$, where r=rank(x), then the sets of subscripts for x are

$$\{(i_1,\ldots,i_r) \in \mathbb{N}^r \mid 1 \le i_k \le s_k,\ k=1,\ldots,r\}$$

We define a map called subscripting from the subscripts for x to the set of items of x by listing all the subscripts in lexicographic order and pairing them with the corresponding components in the list of x. By lexicographic order we mean that $(i_1,\ldots,i_r)$ precedes $(j_1,\ldots,j_r)$ if there exists an n such that $i_k=j_k$ for every $k<n$ and $i_n<j_n$.

An index for x is a basic vector $(r,(i_1,\ldots,i_n))$, where r is the rank of x and $(i_1,\ldots,i_n)$ is a subscript for x. The array of all indices for x is the array of the same shape as x whose items are the indices of the corresponding items of x. Thus, the array of all indices for x is an element of L_2 and its list is the tuple of the indices of x in lexicographic order. We say that d_j is the item of x indexed by I if I is the index whose item is mapped to d_j by subscripting.

In subsequent sections, we use mathematical notation as a metalanguage to define precisely the primitive functions of APL. We have adopted the following conventions: the roman letters x and y are used as variables over U_1 or subsets of U_1; the letter f is used to denote an arbitrary monadic or dyadic function; and the APL letters I, J, and K are used to denote arrays that are constrained to satisfy some specified relationship to a variable. Further, the symbol $\leftrightarrow$ is used to indicate that the entire expression to the left has the same meaning as the entire expression to the right. It is used both to illustrate examples and to

define equations for functions. We use APL syntax rules in such equations; however, we define monadic operators to be right monadic and use juxtaposition of operators to denote successive application of two operators. Thus, $\cdots f$ is analogous to $\div\div x$.

2. Some Primitive Functions

For $x \in U_1$ we define the shape function, $\rho: U_1 \to U_1$ by

$$\rho x \leftrightarrow (\text{rank}(x), \text{shape}(x))$$

from which it follows that

$$\rho\rho x \leftrightarrow (1, \text{rank}(x))$$

We also define the ravel function, $,: U_1 \to U_1$ by

$$,x \leftrightarrow (\text{count}(x), \text{list}(x))$$

Next we define the odometer function [7], denoted ι, to be the function that maps ρx into the array of all indices for x.

Let J be any array with items that are indices for $x \in U_1$. Then we define choose indexing, denoted $\circ$, such that $J \circ x$ is the array of the same shape as J whose items are the items of x indexed by the indices in J. The relationship among shape, odometer, and choose is expressed by the identity $x \leftrightarrow (\iota\rho x) \circ x$ for every $x \in U_1$.

In addition, we define for $x \in U_1$ the seal (or enclose or conceal) function, denoted <, by

$$<x \leftrightarrow (\phi, x)$$

That is, <x is a scalar array with the array x as its item. The unseal (or disclose or reveal) function, denoted >, is defined by

$$>(\phi, x) \leftrightarrow x$$

The definition of unseal imposes the constraint that >y is defined only if there exists an x such that y=<x. In general, we will not explicitly discuss all the domain constraints imposed by our definitions, but leave them to the reader to discern. Clearly, our definitions imply that unseal is the inverse of seal, namely

$$><x \leftrightarrow x \quad \text{for } x \in U_1$$

$$<>y \leftrightarrow y \quad \text{for } y \in \text{domain of unseal}$$

Some examples:

$$\iota\rho\text{'ABC'} \leftrightarrow (<,1),(<,2),<,3$$

$$\iota 2\ 2 \leftrightarrow 2\ 2\rho(<1\ 1),(<1\ 2),(<2\ 1),<2\ 2$$

$$((<2),<1)\circ(<3\ 4\ 5),4,\text{'X'},6 \leftrightarrow 4,<3\ 4\ 5$$

$$\iota\rho 3 \leftrightarrow <\theta$$

3. By-scalar

We define a new operator, by-scalar, denoted by $\top$, which applies to both monadic and dyadic functions. If f is monadic,

$$(<I)\circ\top f\ x \leftrightarrow f(<I)\circ x$$

for any index I for x. If f is dyadic, and provided shape(x)=shape(y),

$$(<I)\circ x \top f\ y \leftrightarrow ((<I)\circ x) f (<I)\circ y$$

for any index I for x (or y). We extend the definition in the dyadic case when x is scalar to

$$(<I)\circ x \top f\ y \leftrightarrow x\ f\ (<I)\circ y$$

for any index I for y, and similarly if y is scalar.

Notice that these definitions imply that f maps scalars to scalars in its domain of definition. By-scalar is closely related to the concept of scalar functions in APL. Indeed, we call a monadic function f a scalar function on domain $D \subseteq U_1$ if $\top f\ x \leftrightarrow f\ x$ for every $x \in D$. A dyadic function is scalar on $D \subseteq U_1^2$ if $x \top f\ y \leftrightarrow x\ f\ y$ for every $(x,y) \in D$.

By-scalar may be used to define the monadic raise and lower primitive functions given by Gull and Jenkins [4, Table 1].

Raise, denoted by $\uparrow$, is defined by

$$\uparrow \leftrightarrow \top<$$

and lower, denoted $\downarrow$, is defined by

$$\downarrow \leftrightarrow \top>$$

The domain of raise is U_1, while that of lower is the set of arrays with items that are scalars (rank-0 arrays). Lower is undefined on a basic array. (Lower is naturally generalized to arrays with items that are arrays all of the same shape; we defer the definition to Section 8.)

The identity

$$x \leftrightarrow \downarrow\uparrow x$$

for every $x \in U_1$, follows from the definition of $\top$ and the identity for < and >. It holds for empty arrays because there is only one empty array of each shape in U_1.

Examples:

Let F be the function $X\ F\ Y \leftrightarrow +/X|\iota Y$. Then

$$4\ 3\ 2\top F\ 9\ 8\ 7 \leftrightarrow 13\ 9\ 4$$

$$\uparrow\iota 3 \leftrightarrow (<1),(<2),<3$$

4. By-item

We may now use by-scalar to define the by-item operator, denoted ¨, of Gull and Jenkins [4].

If f is monadic,

$$\ddot{}f\ x \leftrightarrow \top f'\ x$$

where f' is the function $f'x \leftrightarrow <f>x$, and if f is dyadic,

$$x\ddot{}f\ y \leftrightarrow x\top\bar{f}\ y$$

where $\bar{f}$ is the function

$$x\ \bar{f}\ y \leftrightarrow <(>x)f>y$$

The fundamental distinction between by-item and by-scalar is that by-item applies its function to the items (which must be arrays) of the argument(s); whereas, by-scalar applies its function to the scalar arrays constructed to hold the items of the argument(s). This statement is made explicit by the identity

$$\top f\ x \leftrightarrow \downarrow\ddot{}f\uparrow x$$

for all x in the domain of $\top f$. Note that by-scalar is an idempotent operator; that is,

$$\top\top f \leftrightarrow \top f$$

However, by-item is not idempotent since $\ddot{}\ddot{}f\ x$ applies f to the items of the items of x.

Examples:

$$(\uparrow 2\ 3)\ddot{},\uparrow 4\ 5 \leftrightarrow ((<2),<3)\ddot{},(<4),<5$$
$$\leftrightarrow (<2\ 4),<3\ 5$$

$$\ddot{}\rho\ddot{}\iota\uparrow\iota 3 \leftrightarrow \ddot{}\rho\ddot{}\iota(<1),(<2),<3$$
$$\leftrightarrow \ddot{}\rho(<,1),(<1\ 2),<1\ 2\ 3$$
$$\leftrightarrow (<,1),(<,2),<,3$$

$$\ddot{}\ddot{}\rho<\uparrow\iota 3 \leftrightarrow \ddot{}\ddot{}\rho<(<1),(<2),<3$$
$$\leftrightarrow <(<0),(<0),<0$$

5. By-slice

We define now an operator that is an extension of by-scalar and of the present APL axis. It applies a function to slices of APL arrays, a concept we proceed to formalize.

For brevity in writing formulas and examples, we extend the domain of choose, odometer, and catenate (defined below) to accept scalars as arguments where one-item vectors would otherwise be required. In particular, scalars are permitted as the index for a vector in choose indexing. Then we may define indexing for vectors by

$$x[I] \leftrightarrow (\uparrow I)\circ x$$

where x is a vector and I is an array with items that are subscripts for x. For example:

```
5 3 8[2] ↔ (↑2)∘5 3 8
         ↔ (<2)∘5 3 8
         ↔ (<,2)∘5 3 8
         ↔ 3

5 3 8[3 1] ↔ (↑3 1)∘5 3 8
           ↔ ((<3),<1)∘5 3 8
           ↔ ((<,3),<,1)∘5 3 8
           ↔ 8 5
```

We define catenate for vectors by

$x,y \leftrightarrow (n,d)$

where n=count(x)+count(y) and d is the n-tuple with the items of x followed by the items of y, and we extend it to scalars as noted above.

A slice specification for x is a scalar or vector I with integer items designating axes of x without repetition. That is, the items of I are a subset of {1,2,...,rank(x)}. Let I' denote the vector with items that are the complement set of axes of x in ascending order, and let I'' be the permutation of ⍳⍴⍴x that under indexing reorders I',I to ⍳⍴⍴x ($I'' \leftrightarrow \Delta I',I$). Then, for a slice specification I of x, we define a dyadic generalization of raise called slice-raise (which differs from Gull and Jenkins' dyadic form of ↑ [4]) as follows: $I\underline{\uparrow}x$ is the array of shape $(\rho x)[I']$ whose items are arrays of shape $(\rho x)[I]$ such that if J is an item of $\underline{\iota}(\rho x)[I']$ and K is an item of $\underline{\iota}(\rho x)[I]$ then

$$(<K)\circ\rangle(<J)\circ I\underline{\uparrow}x \leftrightarrow (<(K,J)[I''])\circ x$$

One may verify that

$$\Theta\underline{\uparrow}x \leftrightarrow \underline{\uparrow}x \quad \text{and} \quad (\iota\rho\rho x)\underline{\uparrow}x \leftrightarrow <x$$

Examples of the use of slice-raise are

```
1↑2 3⍴⍳6 ↔ (<1 4),(<2 5),<3 6

2↑2 3⍴⍳6 ↔ (<1 2 3),<4 5 6

2↑0 2⍴1 ↔ ⍬

2 3↑0 3 2⍴'A' ↔ ⍬
```

Note that slice-raise may map empty arrays of different shapes to the same object.

We define a complementary function slice-lower to reverse (where possible) the effect of a slice-raise. Thus, $I\underline{\downarrow}x$ is the unique array y if it exists, such that $I\underline{\uparrow}y \leftrightarrow x$. Then $\Theta\underline{\downarrow}x \leftrightarrow \underline{\downarrow}x$ and $(\iota\rho\rho x)\underline{\downarrow}x \leftrightarrow >x$. For x nonempty, the identity $x \leftrightarrow I\underline{\downarrow}I\underline{\uparrow}x$ holds for every I that is a slice specification for x. This identity can be extended to the entire universe only if $\cup_1$ is extended to include more empty arrays [5].

We now define the operator by-slice, which is denoted by indexing the function with a list of two or three slice specifications. If f is monadic,

$$f[J;K]x \leftrightarrow K\underline{\downarrow}\ddot{}f\ J\underline{\uparrow}x$$

and if f is dyadic,

$$x\ f[I;J;K]y \leftrightarrow K\underline{\downarrow}(I\underline{\uparrow}x)\ddot{}f\ J\underline{\uparrow}y$$

By-slice applies the function to the slices of the argument(s) indicated by the first (two) slice specification(s) and packs the results in a single array using the final slice specification to indicate the positions to put the axes of the result values.

Examples:

Let $A \leftarrow$ 2 2 2 2⍴⍳16.
Then

```
2 3↑A ↔ 2 2⍴(<2 2⍴1 3 5 7),
         (<2 2⍴2 4 6 8),<2 2⍴9 11 13 15),
         <2 2⍴10 12 14 16

A×[2 3;1 2;2 3]2 2⍴0 1 0 1 ↔
  2 2 2 2⍴0 0 3 4 0 0 7 8,
  0 0 11 12 0 0 15 16
```

For any monadic function f and any array A in its domain,

$$f\ A \leftrightarrow >\ddot{}f<A \leftrightarrow f[\iota\rho\rho A;\Theta]A$$

Notice that the slice-lower in the definition of by-slice implies that the function to which by-slice is applied must yield an array of the same shape for each slice to which it is applied. Since

$$\top f\ x \leftrightarrow \underline{\downarrow}\ddot{}f\underline{\uparrow}x \leftrightarrow \Theta\underline{\downarrow}\ddot{}f\Theta\underline{\uparrow}x \leftrightarrow f[\Theta;\Theta]x$$

we see that by-scalar is just by-slice over slices specified by no axes, that is, over the set of scalars that hold the items of x.

The definitions of raise and lower,

$$\underline{\uparrow}x \leftrightarrow \overline{\top}{<}x$$

$$\underline{\downarrow}x \leftrightarrow \overline{\top}{>}x$$

generalize to the identities,

$$I\underline{\uparrow}x \leftrightarrow {<}[I;\theta]x$$

$$I\underline{\downarrow}x \leftrightarrow {>}[\theta;I]x$$

which can be easily verified. Note that this indicates that by-slice with seal makes raise and slice-raise redundant; by postulating by-slice directly they could be eliminated. It shows also that slice-raise and slice-lower are closely related to the indexed conceal and reveal functions defined by Brown [1]; that is, $<[I;\theta]x$ is equivalent to Brown's $\subset[I]x$ except for system 0 versus system 1 considerations.

6. Outer Product and Reduction

We now define the outer product operator, denoted $\circ$, by

$$x \circ f\ y \leftrightarrow (\iota\rho\rho x)\underline{\downarrow}(<x)\overline{\top}f'y$$

where f' is the function

$$x\ f'\ y \leftrightarrow (>x)\overline{\top}f\ y$$

Outer product applies f to the scalars that hold the items of each argument and stores the items of the function values as items of the result.

Since $x \circ f\ y$ is defined by an $\iota\rho\rho x$ slice-lower, we have

$$\begin{aligned}\rho x \circ f\ y &\leftrightarrow (\rho x),\rho(<x)\overline{\top}f'y\\ &\leftrightarrow (\rho x),\rho y\end{aligned}$$

Let I be an index for x and J be an index for y. Then I,J is an index for $x \circ f\ y$ and

$$\begin{aligned}(<I,J)\circ x \circ f\ y &\leftrightarrow (<I,J)\circ(\iota\rho\rho x)\underline{\downarrow}(<x)\overline{\top}f'\ y\\ &\leftrightarrow (<I)\circ{>}(<J)\circ(\overline{<x})\overline{\top}f'\ y\\ &\leftrightarrow (<I)\circ{>}(<x)f'\ (<J)\circ y\\ &\leftrightarrow (<I)\circ{><}({><}x)\ \overline{\top}f\ (<J)\circ y\\ &\leftrightarrow (<I)\circ x\overline{\top}f(<J)\circ y\\ &\leftrightarrow ((<I)\circ x)\ f\ (<J)\circ y\end{aligned}$$

The above definition of outer product is compatible with the outer product operator of present APL (except in syntax) when applied to a scalar function. Moreover, if $\overline{\top}f$ is defined for a function f, then

$$\circ\overline{\top}f \leftrightarrow \circ f$$

because both reduce to the same definition by the idempotency of $\overline{\top}$.

Outer product can be applied to the by-item of a more general function to operate on the items of the arguments. For example,

$$\begin{aligned}&((<1\ 1),<2\ 3)\ \circ\ddot{},<3\\ &\leftrightarrow ((<1\ 1)\ddot{},<3),(<2\ 3)\ddot{},<3\\ &\leftrightarrow (<1\ 1\ 3),<2\ 3\ 3\end{aligned}$$

Thus, with this definition of outer product, we control explicitly the level at which we apply a function by combining outer product with by-item. Since this combination occurs frequently, we might wish to give it a special notation such as $\ddot{\circ}$.

We define reduction in a similar way. We use red to denote the reduction operator to avoid confusion with the left monadic notation of APL. For a vector x we define

$$\underline{\text{red}}\ f\ x = \begin{cases} x[1]f\ \underline{\text{red}}\ f\ x[1+\iota^{-}1+\rho\rho x] & \text{if } 1<\rho x\\ x[1] & \text{if } 1=\rho x\\ \text{neutral}(f) & \text{if } 0=\rho x\end{cases}$$

where neutral(f) is the neutral for f if one exists. If f has a neutral α, then we define α to be the neutral of $\circ f$ and $\overline{\top}f$ also, and $<\alpha$ to be the neutral of $\ddot{}f$. For example,

```
red+1 2 3 ↔ 1+red+2 3
          ↔ 1+2+red+,3
          ↔ 1+2+3
          ↔ 1+5
          ↔ 6

red¨,(<3),(<1),<4 ↔ (<3)¨,red¨,(<1),<4
                  ↔ (<3)¨,(<1)¨,red¨,,<4
                  ↔ (<3)¨,(<1)¨,<4
                  ↔ (<3)¨,<1 4
                  ↔ <3 1 4
```

With this extension of reduction, we explicitly control the level of application of the argument function by using by-item, just as in outer product. We might consider the notation ⌿f to denote red¨f. The effect of red⍤1 f is the same as red f. For example, if ⍴x ↔ 3,

```
red⍤1 f x ↔ x[1]⍤1 f red⍤1 f x[2 3]
          ↔ x[1]⍤1 f x[2]red⍤1 f,x[3]
          ↔ x[1]⍤1 f x[2]⍤1 f x[3]
          ↔ x[1]⍤1 f (x[2]f x[3])
          ↔ x[1]f x[2]f x[3]
          ↔ red f x
```

The definitions for scan and inner product follow directly from the definitions of reduction and outer product, so we omit the details. As an example of the definitional power of these operators, we apply them in the following definition of slice indexing, denoted ⍧:

$$I⍧x \leftrightarrow (<⌿\ddot{\circ}, \ddot{\uparrow} I)\circ x$$

Let us explain the defining identity for slice indexing. Its purpose is to ensure that if

$$I=(k,(I_1,\ldots,I_k))$$

where I_j, $j=1,\ldots,k$ is an array of subscripts along the jth axis, then

$$I⍧x \leftrightarrow x[I_1;\ldots;I_k]$$

The function $\ddot{\circ}$, builds all the possible pairwise catenations of the items of its arguments. It is applied successively to the $\uparrow I_j$, obtained through $\ddot{\uparrow} I$, by doing the reduction of its by-item. Thus the result of $>⌿\ddot{\circ},\ddot{\uparrow} I$ is an array whose items are vectors $i_1,\ldots,i_r$ where i_j is an item of I_j; that is, they are the indices for x that choose the desired items. This definition is closely related to the definition for slice indexing described by Haegi [8].

Let us show that the definition satisfies the identity

$$⍬⍧x \leftrightarrow (<⍬)\circ x$$

when x is a scalar.

```
⍬⍧ ↔ (>⌿¨∘,¨↑⍬)∘x
   ↔ (>⌿¨∘,⍬)∘x  since ¨f ⍬ ↔ ⍬
   ↔ (>red¨⍴¨,⍬)∘x
   ↔ (>neutral(¨⍴¨,))∘x
   ↔ (>(<<⍬))∘x since neutral(,)=⍬
                 and the above rules
   ↔ (<⍬)∘x
```

7. Symmetry and Compatibility with APL

The definitions of several primitive functions of APL, as well as the definitions of the operators scan and reduction, are not symmetric with respect to the axes; this limits the number of useful identities between them. (For example, the APL catenate, reduction, reverse, and so on, use the last axis by default if no other axis is given.)

It is possible to achieve the semantics of the nonsymmetric APL functions by defining symmetric versions and using by-slice. We proceed to give a possible set of such symmetric functions together with the appropriate by-slice applications which yield the corresponding APL semantics. We use APL expressions on the right of the following identities.

- Let red f x ↔ f/,x.

  Then (red f)[I;⍬]x ↔ f/[I]x

  and we have identities such as:

  (red f)[I,J;⍬]x ↔ f/[I]f/[J]x

  if f is associative and $I<J$.

- Let scan f x ↔ f\x but limit the domain to vectors.

  Then (scan f)[I;I)x ↔ f\[I]x.

- Let rev x ↔ (ρx)ρϕ,x.

  Then rev[*I*;*I*]x ↔ ϕ[*I*]x.

- Let x rot y ↔ xϕy but limit the domain to x scalar, y vector.

  Then x rot[⍬;*I*;*I*]y ↔ xϕ[*I*]y.

- Let x cat y ↔ (,x),(,y).

  If ρρx ↔ ρρy,

  then x cat[*I*;*I*;*I*]y ↔ x,[*I*]y.

  If ρρx ↔ (ρρ*Y*)-1 or ρρx ↔ 0,

  then x cat[⍬;*I*;*I*]y ↔ x,[*I*]y and

  x cat[⍬;⍬;*I*]y ↔ x,[*I*-.5]y.

- Let x comp y ↔ (,x)/,y.

  Then x comp[1;*I*;*I*]y ↔ x/[*I*]y.

- Let x exp y ↔ (,x)\,y.

  Then x exp[1;*I*;*I*]y ↔ x\[*I*]y.

Note that red, rev, cat, comp, and exp, which are variations of reduction, reverse, catenate, compress, and expand, respectively, ravel their argument(s) to remove the asymmetric treatment of the axes.

Brown [1] and Ghandour and Mezei [2] give generalizations of axis, applied to the above functions (and others), which use a vector of axes and apply the function to the indicated slices; however, the semantics are function-dependent in that separate semantics have to be specified for each function or operator. Our definition of by-slice is an attempt to provide an operator that has the semantic power of a generalized axis with a function-independent meaning (which allows it to be applied to user-defined functions).

It would be ideal if default values for the fields of by-slice could be determined in all the above cases so that axis could be treated as just a special case of by-slice. Indeed, for monadic functions, we may use the default rule:

(a) When f is monadic, f[*I*] is interpreted as f[*I*;*I*] (resp. f[*I*;⍬]) if the rank of the result of applying f to a slice along *I* is ρ*I* (resp. 0).

Similarly, for dyadic functions, we want a default rule such that

x f[*I*]y ↔ x f[*I'*;*J'*;*K'*]y

will be satisfied for rotate, catenate, compress, and expand. Unfortunately, these are conflicting requirements. Catenate (with a non-fractional index) and rotate suggest the rule:

(b) If the arguments are of the same rank, then take *I'*←*J'*←*K'*←*I*. Otherwise, if (ρρx)> ρρy (resp. <), take *I'*←*I* and *J'*←⍬ (resp. *I'*←⍬ and *J'*←*I*) and *K'*←*I*.

This rule gives an interesting interpretation for axis on scalar functions as the examples below illustrate.

```
2 3ρ⍳6 +[1]0 5 10 ↔ 2 3ρ1 7 13 4 10 16

2 3ρ⍳6 +[2]0 5 ↔ 2 3ρ1 2 3 9 10 11
```

Alas, rule (b) fails on compress and expand, and while other defaults might be chosen, no function-independent default can satisfy the requirements to make present axis a special case of by-slice for both compress and rotate. The basic reason is that, for *V* a vector and *M* a matrix, the interpretations in present APL of *V*/[1]*M* and *V*ϕ[1]*M* are inconsistent, being *V* comp[1;1;1]*M* and *V* rot[⍬;1;1]*M*, respectively.

8. Proposals

Which (if any) of the ideas in the previous section can or should be accepted for present APL or for an APL with nested heterogeneous arrays? First we suggest that by-scalar and by-slice should be included in present APL and be applicable

to any primitive or user-defined function that produces values of the same shape for the slices to which it is applied. Despite the discussion of the last section, it is possible to include by-slice in a compatible extension in the sense of using the axis notation to have its current meaning in all cases where it is defined, but to systematically extend by by-slice. The reason is that axis, as used in present APL, needs the definitions of the functions to which it is applied only for vectors. These coincide with the symmetric versions for that restricted domain, and hence there is no conflict between axis and by-slice.

For example, the present interpretation of +/[*I*]x is compatible with the definition of by-slice applied to the present function "+/". To extend axis with by-slice without changing the semantics of current APL, the following conventions can be followed:

- Compress, expand, and lamination (catenate with a fractional index) must be treated as special cases.
- Otherwise, follow default rule (a) for monadic functions and default rule (b) for dyadic functions.
- Use the current definitions of reduction, reverse, and so on when no axis is specified, and use the symmetric versions with by-slice otherwise.

Then the by-slice notation could be used, and it would be consistent with axis wherever the semantics of axis have been extended. This approach (admittedly a "kludge") is compatible with current APL, yet yields the benefit of the symmetric definitions.

Extending axis and adding by-slice in this manner and allowing their use on user-defined functions would greatly enhance the expressive power of the language, perhaps leading to entirely new approaches to solving many problems. Extending the remaining APL operators (reduction, scan, outer product, and inner product) to handle any functions that map scalars to scalars would also be beneficial.

It is also possible to add a version of odometer and choose to APL without nested arrays. Since, for an array, all indices are vectors of the same length, an array of indices can be represented by an array of integers of one rank higher. Thus, ⍳ could be extended to generate the array of all indices for an arbitrary array such that

$$\rho\iota\rho A \leftrightarrow (\rho A),\rho\rho A$$

and choose could be defined to accept arrays of indices represented in this way. Hence, we propose that ⍳ be defined so that

$$(\iota\rho A) \leftrightarrow (1+\rho\rho A)\underline{\uparrow}\iota\rho A$$

and a "choose" be defined such that

$$I \text{ "choose" } A \leftrightarrow ((1+\rho\rho A)\underline{\uparrow}I)\circ A$$

If heterogeneous nested system 1 arrays are added to APL, we suggest extending the above proposal to include (in addition to by-scalar and by-slice) the functions seal, unseal, raise, lower, choose, slice, and odometer, as well as the operator by-item. Moreover, reduction, scan, outer product, and inner product should be defined with the semantics we have presented. All the operators should be applicable to the primitive or user-defined functions that meet the domain constraints, and function expressions should be allowed. Neutral elements should be defined for those primitive functions that have them; then reduction on an empty array should be defined only for those primitive functions (or for functions derived from them using operators).

Other functions or extensions of functions could be added for convenience. For example, slice-raise and slice-choose, and lower (monadic $\underline{\downarrow}$) could be extended to arrays whose items are all arrays of the same shape:

$$\underline{\downarrow}x \leftrightarrow ((\rho x)+\iota\rho\rho>(<1)\circ,x)\underline{\downarrow}x$$

In the appendices, we propose definitions for the format of a nested array (⍕), a mesh operator, and index-finder (dyadic $\underline{\iota}$), which may also be of interest as possible extensions.

9. Conclusions

The major new approach we have taken is to separate the concepts of by-scalar, by-item, and by-leaf (defined below). We feel this separation clarifies many confusions surrounding earlier attempts to define the role of "scalar functions" in APL with nested arrays. In our terminology, a function f is scalar if $\uparrow f \leftrightarrow f$. New scalar functions can be created by $\uparrow$ since $\uparrow\uparrow f \leftrightarrow \uparrow f$. The by-item of a function in our definition can never be identical to the function itself ($\ddot{}f \leftrightarrow f$), because by-item constrains the argument of the function it produces to have items that are not basic, and $\ddot{}f \leftrightarrow f$ would imply an infinite descent. Indeed, $\ddot{}f$ is always a function that descends exactly one level. In order to descend many levels, Gull and Jenkins [4] introduced the operator by-leaf, which may be defined recursively by

$$\underline{\ddot{}}f\ x \leftrightarrow \begin{cases} \underline{\ddot{}}f\ x & \text{if all items of x are arrays (x is a tree array)} \\ f\ x & \text{otherwise} \end{cases}$$

(Here we have denoted by-leaf by $\underline{\ddot{}}$ for reasons stated below.) In the homogeneous subset $H \subset U_1$, this is an interesting operator since descent continues until the leaves of the trees are encountered, the leaves being homogeneous arrays of numbers or characters. If by-leaf is applied after by-scalar ($\underline{\ddot{}}\uparrow f$), then the function f is applied by-scalar to the leaves.

Note that for the heterogeneous universe it is perhaps more natural to use $\underline{\ddot{}}f \leftrightarrow \uparrow f'x$ where

$$f'y = \begin{cases} f\ y & \text{if y is a basic scalar} \\ \underline{\ddot{}}f{>}y & \text{otherwise.} \end{cases}$$

Following More [3], we define a function f to be pervasive in a domain D if

$$\underline{\ddot{}}f\ x \leftrightarrow f\ x \quad \text{for every } x \in D \subset U_1.$$

In a system 0 context, the distinction between $\uparrow$ and $\ddot{}$ is blurred. If x is a basic array, then $\ddot{}f\ x \leftrightarrow \uparrow f\ x$ provided f maps basic scalars to basic scalars; but they are different if x is not basic, even if f maps generalized scalars to generalized scalars. Attempting to characterize a scalar function by $\ddot{}f \leftrightarrow f$ is possible in system 0 but also yields $\underline{\ddot{}}f \leftrightarrow f$; that is, a scalar function is always pervasive. (Here we are inferring properties of system 1 definitions in system 0, using the mapping of U_1 onto U_0 defined by Gull and Jenkins [4].)

The definitions of operators we have given for system 1 can be used in system 0. They achieve compatible semantics (except for telescoping scalars) provided they are interpreted in terms of the mapping from U_1 onto U_0. The conceptual separation achieved by defining by-item to be distinct from by-scalar and by-leaf can be used as well in system 0. Moreover, our definitions of outer product, reduction, and by-slice may also be used for system 0 arrays and are compatible with present usage. Note, however, that because the map of system 1 onto system 0 is not one-to-one, definitions given for system 0 cannot always be lifted to system 1. (For instance, Brown's definition of reduction for a vector of length 1 or greater:

$$\underline{\text{red}}\ f\ x \leftrightarrow \begin{cases} <(>x[1])f>\underline{\text{red}}\ f\ 1\downarrow x & \text{if } 1<\rho x \\ x[1] & \text{otherwise} \end{cases}$$

is incompatible with system 1.)

We have chosen notations for by-item, by-scalar, and by-leaf to correspond to the notations for choose, slice, and reach, respectively, to which they are somewhat related.

choose	$\circ$	$\ddot{}$ by-item
slice	$\downarrow$	$\uparrow$ by-scalar
reach	$\underline{\circ}$	$\underline{\ddot{}}$ by-leaf

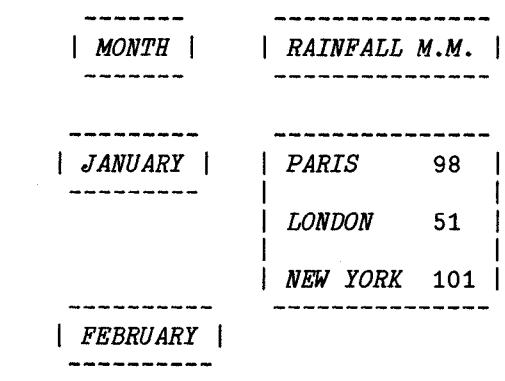
By-leaf is not included in the proposal of Section 8 since the definition which is natural for H (homogeneous arrays) does not allow the function to reach all the basic data in the heterogeneous case. Changing the definition as noted does achieve complete descent but is not very interesting. In a system of tagged arrays with tags on the items (system B in [6]), the tagging structure could control the descent and an interesting recursive control mechanism generated.

Finally we note that we have presented here a rigorous definition of APL arrays, functions, and operators using terminology and a formal framework that makes it possible to discuss APL objects precisely and without confusion. What we have presented is the beginning of a generating set of functions and operators for defining APL. We believe that a formal description of "the common APL subset" can be defined in this formalism and it could be an important aid to the task of language standardization.

Appendix 1 -- Format for Nested Arrays

We propose a definition of format (monadic ⍕) for nested arrays, as an example of the application of the operators and functions defined in the paper.

Often, while building the output of a program, we encounter the problem of placing in a rectangular display, a collection of "boxes" containing data, titles, or decorations. The difficulty is that all the shapes must be adjusted before printing. For example:



Note that the same problem was encountered while constructing the second box in the second column and hence the problem is essentially recursive. This and other related problems can be solved automatically if format is extended to nested arrays properly: Format on an array is defined to compute the format of its items and then adjust the shapes so that they fit together. Then, the problem is solved by applying format to the nested array containing the boxes. The scheme works only if $2\geq\rho\rho\blacktriangledown A$ for all A, since the result of format must be adjustable to a character matrix.

We now give a precise definition of the ⍕ function.

Let A be an array to output. We assume that A is a matrix; for higher rank arrays the print image would be given with inserted blank lines.

Let $V \leftarrow {}^{\cdot\cdot}MAT^{\cdot\cdot}\blacktriangledown A$

where $MAT\ X \leftrightarrow (^{-}2\uparrow 1\ 1,\rho X)\rho X$ is applied to ensure that the collection of boxes obtained from format are all matrices, and ⍕ denotes the present format primitive. Now we define

$$F\ I \leftrightarrow \lceil/[I]\underline{\downarrow}(<<I)^{\cdot\cdot}\circ^{\cdot\cdot}\rho V$$

which computes for I=1 the maximum width of boxes in each column and for I=2 the maximum height of boxes in each line. Then

$$S\leftarrow\overset{\cdot\cdot}{/}\overset{\cdot\cdot}{\circ},^{\cdot\cdot}F\underline{\uparrow}1\ 2$$

is the matrix of adjusted shapes for each item of V, and $S^{\cdot\cdot}\uparrow V$ gives the collection of adjusted boxes. We now have to join the boxes together; it can be done in two steps using the function

$$I\ GLUE\ V \leftrightarrow \overset{\cdot}{/},[I]\ V$$

which "glues" a vector of arrays along axis I. Then

$$\blacktriangledown A \leftrightarrow >1\ GLUE\ 2\ GLUE\ S^{\cdot\cdot}\uparrow V$$

For example, the box in row 2, column 2 above would be displayed by

```
⍕((<'PARIS'),(<'LONDON'),
  <'NEW YORK '),[2]↓98 51 101
```

Appendix 2 -- Expressions on the Left of Assignment and Mesh

It has been suggested for years that assignment in APL be extended to include arbitrary selection expressions on the left. For example,

```
(3↑A)←⍳3
(B/A)←⍳+/B
```

We will show how the desired semantics can be obtained by a pure-value expression. The idea is that a form:

```
(f A)←B
```

where f is an expression amounting to a selection function from *A*, be interpreted as *A*←*A*⌸f *B* where ⌸ is the operator mesh.

We proceed to define mesh, which, when applied to a monadic selection function, produces a dyadic function which "meshes" the right argument into the left argument in the pattern defined by the selection function.

To give the definition, we first need to introduce index finder ⍳. For every index *J* of x,

```
(⊂J)∘x⍳y
```

is the first (in lexicographic order) index *I* of y such that

```
(⊂I)∘y ↔ (⊂J)∘x
```

We may then define g ↔ ⌸f by:

```
    ∇ Z←A g B;M
[1]   M←f⍳⍴A
[2]   'INDEX' ERRORIF(⍴M)≠⍴B
[3]   Z←⍎ SELECT ⍳⍴A
    ∇
```

where ≠ denotes the not-identical function of Gull and Jenkins [4], and where *SELECT* is

```
    ∇ Z←SELECT X
[1]   →(X∈M)/SELECTFROMB
[2]   Z←X∘A
[3]   →0
[4]   SELECTFROMB:  Z←(X⍳M)∘B
    ∇
```

For example, if *F* x ↔ 3↑x and *G* x ↔ *B*/x, then *A*←*A*⌸*F*⍳3 and *A*←*A*⌸*G*⍳+/*B* give us the semantics we wish for (3↑*A*)←⍳3 and (*B*/*A*)←⍳+/*B*.

Indeed, the transformation

(*F A*)←*B* gives *A*←*A*⌸*F B*

is straightforward and purely mechanical; it could be used to extend the syntax of present APL to accept simple selection expressions on the left of assignment.

References

1. Brown, J. A. A generalization of APL. Ph.D. Thesis, Syracuse University, New York, 1971.

2. Ghandour, Z., and Mezei, J. Generalized arrays, operators, and functions. IBM Journal of Research and Development 17, 4 (1973), 335-372.

3. More, T. Types and prototypes in a theory of arrays. Report 320-2113, IBM Scientific Center, Cambridge, Massachusetts, 1976. For a complete bibliography of More's array theory, see APL Quote Quad 7, 4 (Winter 1977), 11-13, and APL Quote Quad 8, 2 (December 1978), 12-13.

4. Gull, W. E., and Jenkins, M. A. Recursive Data Structures in APL. Comm. ACM 22, 2 (Feb. 1979), 79-96.

5. Gull, W. E., and Jenkins, M. A. Decisions for "type" in APL. Proc. of the Sixth ACM Symposium on the Principles of Programming Languages, San Antonio, Texas, January 1979.

6. Jenkins, M. A., and Michel, J. On types in recursive data structures: A study from the APL literature. Proc. of the Third Jerusalem Conference on Information Technology, August 1978, 523-529.

7. Abrams, P. S. An APL machine. Ph.D. Thesis, TR-CS-70-158, Computer Science Department, Stanford University, Calif., 1970.

8. Haegi, H. R. The extension of APL to tree-like data structures. APL Quote Quad 7, 2 (Summer 1976), 8-18.

(Note: This work was supported in part by a grant A7892 from the National Research Council of Canada. A preliminary version of this paper was presented at the Minnowbrook APL Workshop, September 1977.)

M. A. Jenkins
Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6
Canada

Jean Michel
Mathematique
Universite Paris XI (Orsay)
Orsay
France