Empty arrays in extended APL

by D. L. Orth

During the past several years considerable work has been done on extending APL in three areas: operators, heterogeneous data, and nested data. In each area a proposed extension must treat empty arrays consistently. In this paper various possibilities for providing consistent behavior are presented. The new proposals possess at least one of two important qualities in which older proposals tend to be deficient: consistent behavior is independent of the structural properties of rank and nesting, and the user has control over the behavior when he wants it.

1. Introduction

This paper explores the effects on empty arrays of proposed extensions to APL in three general areas: extended operators, heterogeneous arrays, and general (i.e., nested) arrays. An important aspect of the usability of APL is consistent behavior for empty arrays, and this will certainly remain true in an extended APL language. We therefore examine the ways in which empty arrays presently interact with the APL primitive and derived functions, so as to maintain effective interactions in an extended language. It is not the intention of this paper to make detailed proposals for language extensions. The specific extensions presented here simply provide contexts in which to analyze empty arrays: others might have served just as well.

To understand the importance of empty arrays in present APL [1, 2], consider the expression $(P\ M) \neq M$, where M is a character matrix whose rows contain names, and P is a proposition whose value is a boolean vector with one element for each row of its argument, such that $(P\ M)[I]$ is 1 if and only if the criteria tested in P are true for $M[I;]$. The result of the expression is a matrix whose rows all satisfy the criteria tested in P. If no row of M satisfies these criteria, then the result is an empty matrix with no rows. Evidently, after such an expression is executed within an APL program, the program must continue to execute in a consistent manner for both empty and nonempty results. Otherwise, programmers would be required to make explicit provisions for all potential occurrences of empty arrays.

A nonempty array in present APL is uniquely determined by two independent characteristics, its *shape* (the lengths of its axes) and its *element list* (in some predetermined order). *Rank* is another fundamental characteristic but is not independent of shape, since the rank of an array A is the shape of the shape of A. Still another fundamental characteristic is *type* (numeric or character), but it can be derived from the element list. In most implementations there is also an associated storage type which is of finer resolution than the APL numeric type, typically boolean, integer, or real. Some implementations also have more than one storage type for character arrays. Even though it may be possible to ascertain that certain primitives consistently produce values of particular storage types, no primitive distinguishes among storage types for its arguments; primitives may require boolean or integer values of their arguments, but not boolean or integer storage types.

These two independent characteristics of nonempty arrays have obvious realizations as arrays: in APL notation, the shape of an array A is the vector ρA, and the element list is the vector $,A$. The rank of A is $\rho\rho A$. There are commonly used realizations of type that can be produced by various APL expressions, and almost all of them depend on so-called "fill" elements.

The notion of fill in APL is closely related to that of type. Certain APL structural primitives rearrange the elements of

an argument array in ways that sometimes leave gaps that must be filled; in present APL each location corresponding to a gap is filled with a scalar zero or a scalar blank, depending on whether the argument array consists of numbers or characters. The scalar used to fill the gaps is called the *fill element* of the argument array. Like type, the fill element of a nonempty array can be determined from the element list of the array. Unlike type, fill elements are also an important characteristic of empty arrays.

Empty arrays are not vacuous like the empty set in mathematics. They, as well as nonempty arrays, have shape and element lists; they are distinguished from nonempty arrays by having empty element lists and zeros in their shapes. An array with no elements must have the shape characteristic; the expression introduced above, i.e., $(P\,M)\neq M$, may apply to a ten-column matrix and produce a matrix with no rows, i.e., an array with shape zero-by-ten. Empty arrays must also have fill elements because the above-mentioned structural primitive functions can produce nonempty arrays from empty ones, which means that all the locations in the nonempty results are gaps that must be filled, and these primitives must behave as consistently for empty and nonempty arguments as the other primitives. In effect, empty arrays have type: the result of the above expression for a character matrix M is also a character matrix, whether that result is empty or nonempty.

Since empty arrays have empty element lists, their fill elements cannot be computed from their element lists, and therefore *fill* is a third independent characteristic of APL arrays. Of course, it would have been possible in the beginning to have defined the fill element of an empty array in terms of emptiness, so that the fill element would have always been computable from the element list. However, this was not done, and in retrospect the assignment of a single fill element to all empty arrays would have been unfortunate.

Thus, fill is a third characteristic of APL arrays that is independent of shape and element list (if only because of empty arrays) that (like the other two) can be realized as an array and that has been particularly useful in extending the definitions of primitive functions to empty arrays.

More observed all this in a context similar to APL [3, 4] and—from the viewpoint of APL—has adopted the fill element of an array as the realization of its type. Thus in APL the scalar zero would be taken as the type of numeric arrays and the scalar blank as the type of character arrays. Following More, *type* would be called the third independent characteristic of APL arrays, and in our Sections 3 and 4 the arrays from which fill is derived are in fact called *type arrays*, or simply *type*. More's work in this area has led to his concept of *array prototypes*, which are used to provide array theory primitives with consistent extensions to empty arrays.

Several years ago a series of papers appeared relating More's work in array theory to APL extensions, most notably, perhaps, [5]; a complete bibliography can be found in [6]. These papers are primarily concerned with general, or nested, arrays and the primitive functions and operators that apply to them. Some attention is given to the problem of empty arrays, principally in [7], and the general conclusion, at least for the so-called permissive general array systems, is that More's array theoretic concepts for empty arrays should be adopted for extended APL. However, despite the firm theoretical foundation for prototypes in array theory, in many practical situations within the context of APL, fill based on prototypes does not behave as well as some other possibilities [8]. There are two aspects of prototypes that make them too restrictive for extended APL: First, when adapted to APL, prototypes provide fill elements, whereas—as we will see—fill arrays are often more suitable; and second, the prototype of a nonempty array is derived from the element list and therefore cannot be independently specified.

In this paper, Section 2 is concerned with the effect of empty arrays on extensions to the primitive operators. There are essentially two different sets of definitions for the extended operators, instances of which can be found in [9, 10]. We use the definitions in [10], with some variations, because they do not require general arrays and therefore can be treated independently of the other extensions under consideration. A brief discussion of the other definitions appears in Section 4.

Section 3 is concerned with empty arrays and heterogeneous data. Again, in order to isolate this extension from the others, only flat heterogeneous arrays are discussed, i.e., APL-like arrays that are a mixture of numbers and characters. A definition of fill arrays is developed and its advantages over fill elements are discussed.

Section 4 concerns empty general arrays. A permissive general array system is discussed first; the name derives from the fact that the fundamental primitive used to create nested data has no effect on ordinary scalars. Such a system must also admit heterogeneous data if it is to be effective, and the results presented here extend those of Section 2. A brief discussion of strict general array systems follows. As in [7, 10], there is really only one useful general form that a fill definition can take. Next is a presentation leading to the conclusion that ordinary fill elements are often preferred to more complicated forms.

Section 5 deals with the conflict that arises because a simple form of fill is more effective in some applications and a more complex form in others. Essentially, the solution offered is to make fill a specifiable quantity, and that in turn rests on establishing fill as a concept independent of type. Curiously, it is also shown that specifiable fill provides an informal kind of abstract data typing. See [6, 11] for discussions of extensions to APL that permit more formal data abstractions.

The index origin is assumed to be 1 unless stated otherwise.

2. Extended operators

In present APL the primitive operators (reduction, scan, axis, outer product, and inner product) apply only to certain primitive functions. It is assumed in this section that these operators apply to all functions: primitive, derived, and defined. It is also assumed that only simple homogeneous arrays are permitted, as in present APL. The definitions adopted here for the extended primitive operators are as follows, where f and g are any functions and, unless stated otherwise, A and B are nonempty arrays:

Definition 1—reduction

$$f/A \leftrightarrow A[;\cdots;1]f\, f/[;\cdots;1\downarrow \iota N]$$

if $N > 1$, where N is a scalar whose value is $^{-}1\uparrow \rho A$. Otherwise,

$$f/A \leftrightarrow A[;\cdots;1]$$

or

$$f/A \leftrightarrow A$$

if $N = 1$ or if A is a scalar, respectively.

Definition 2—scan

$$(f\backslash A)[;\cdots;I] \leftrightarrow f/A[;\cdots;\iota I]$$

where the scalar I takes any valid index value. Otherwise, if $0 = {}^{-}1\uparrow \rho A$,

$$f\backslash A \leftrightarrow A$$

Definition 3—axis

If $C \leftarrow f[1]B$ then

$$C[I1;\cdots;IN;;\cdots;] \leftrightarrow fB[;I1;\cdots;IN]$$

where the integers $I1$ through IN take on all valid index values. The definition of $f[K]B$ is similar, except that individual subarrays of B to which f is applied lie along the Kth axis instead of the first. The length of the Kth axis may be zero. Analogously, if $C \leftarrow Af[1]B$ then

$$C[I1;\cdots;IN;;\cdots;] \leftrightarrow A[;I1;\cdots;IN]fB[;I1;\cdots;IN]$$

and $Af[K]B$ is similar. Presumably this operator would be extended to permit specification of more than one axis, as well as specification of different axes for the argument (or, in the dyadic case, both arguments) and the result. However, the additional generality adds nothing to the discussion here.

Definition 4—outer product

If $C \leftarrow A\circ .fB$ then

$$C[I1;\cdots;IN;J1;\cdots;JM;;\cdots;] \leftrightarrow A[I1;\cdots;IN]fB[J1;\cdots;JM]$$

where the integers $I1$ through JM take on all valid index values.

Definition 5—inner product

If $C \leftarrow Af.gB$ then

$$C[I1;\cdots;IN;J1;\cdots JM;;\cdots;] \leftrightarrow f/A[I1;\cdots;IN;]gB[;J1;\cdots;JM]$$

for $I1$ through JM as above.

For example, if $M \leftarrow 2\ 2\rho\iota 4$, then

$$3\uparrow[2]\, M \leftrightarrow 2\ 3\rho 1\ 2\ 0\ 3\ 4\ 0$$

$$,/M \leftrightarrow 1\ 3\ 2\ 4$$

$$M[;1];\circ .,M[;2] \leftrightarrow 2\ 2\ 2\rho 1\ 2\ 1\ 4\ 3\ 2\ 3\ 4$$

Note that the utility of both outer product and inner product would be enhanced significantly if the axis operator applied to these operators. Outer product applies its function argument to all pairs of scalars, one from the left argument array and the other from the right. The effect of axis would presumably be to apply the function argument to all pairs of subarrays along specified axes. The effect on inner product would be analogous. Similar behavior is obtained by Iverson [10] through the concept of function rank.

These definitions evidently give the same results as the present APL operators when applied to the same restricted class of functions and nonempty arrays. The problem to be considered in this section is to define the extended operators for empty arguments in such a way as to agree with the present operators whenever the latter are defined and to apply to as wide a range of functions and empty arrays as possible. There are two cases to the problem of empty arguments: Either the associated result in present APL is nonempty, as can occur in reduction and inner product, or the result is empty. The case of nonempty results is considered first.

• *Nonempty results*

In present APL only reduction and inner product can produce nonempty results from empty arguments. For example, $+/\iota 0$ is 0, $\times/2\ 0\rho 0$ is 1 1, and $(2\ 0\rho 0)+.\times(0\ 3\rho 0)$ is $2\ 3\rho 0$. It is enough to discuss reduction because nonempty results of an inner product on empty arguments are due to the reduction therein. The following definition is based on the one in the IBM APL Standard [12, Section 2.4.1]; f is assumed to be a primitive scalar function.

Definition 6

If $0 = {}^{-}1\uparrow \rho A$ (i.e., there are no subarrays along the last axis), then (1) if f has no left or right identity, then f/A evokes a domain error; (2) if f has a right identity, denoted here by the scalar RI, then f/A is $({}^{-}1\downarrow \rho A)\rho RI$; otherwise, (3) if f has a left identity LI, then f/A is $({}^{-}1\downarrow \rho A)\rho LI$.

Note: If f has both a left and right identity, they are identical, and their common value is called the identity of f.

This definition is motivated by the following relation for arrays A with $1 < {}^{-}1\uparrow \rho A$:

$f/A \leftrightarrow A[;\cdots;1])\ f\ f/A[;\cdots;1 \downarrow \iota^{-}1 \uparrow \rho A]$ (1)

If $1 = {}^{-}1 \uparrow \rho A$, this relation takes the form

$A[;\cdots;1] \leftrightarrow A[;\cdots;1]\ f\ f/A[;\cdots;\iota 0]$ (2)

and is true if f has a right identity and $f/A[;\cdots;\iota 0]$ is appropriately defined in terms of that identity.

The expressions $({}^{-}1 \downarrow \rho\omega)\rho LI$ and $({}^{-}1 \downarrow \rho\omega)\rho RI$ in the above definition are called *left* and *right identity-inducing expressions* by Brown and Jenkins [13]. In addition, it is convenient to refer to g: $({}^{-}1 \downarrow \rho\omega)\rho LI$ and h: $({}^{-}1 \downarrow \rho\omega)\rho RI$ as *left* and *right identity-inducing functions*. Brown and Jenkins discuss identity-inducing expressions for nonscalar primitives, as well as the generation of such expressions for derived functions, and even though their definition of reduction differs from the one above (see Section 4), their basic idea can be adapted to our purposes here.

Definition 7

The function f is said to have a *right identity-inducing function* g if for every A with $1 = {}^{-}1 \uparrow \rho A$,

$A[;\cdots;1] \leftrightarrow A[;\cdots;1]fgA[;\cdots;\iota 0]$ (3R)

or a *left identity-inducing function* g if for every such A,

$A[;\cdots;1] \leftrightarrow (gA[;\cdots;\iota 0])fA[;\cdots;1]$ (3L)

Extended reduction is then defined as follows:

If $0 = {}^{-}1 \uparrow \rho A$, then: (1) if f has no right or left identity-inducing function, then f/A evokes a domain error; (2) if f has a right identity-inducing function g, then f/A is defined to be gA; otherwise, (3) if f has a left identity-inducing function g, then f/A is defined to be gA.

For example, consider the catenate primitive α, ω. If $\rho A \leftrightarrow 2\ 2$, then

$A \leftrightarrow A, 2\ 0\rho A$ and $A \leftrightarrow (2\ 0\rho A), A$

More generally,

$A \leftrightarrow A,(({}^{-}1 \downarrow \rho A), 0)\rho A$ and

$A \leftrightarrow ((({}^{-}1 \downarrow \rho A), 0)\rho A), A$

for any array A. In particular, and following the form of (3R) and (3L), if $I \leftarrow A[;\cdots;\iota 0]$ then

$A[;\cdots;1] \leftrightarrow A[;\cdots;1],(({}^{-}2 \downarrow \rho I), 0)\rho I$

$A[;\cdots;1] \leftrightarrow ((({}^{-}2 \downarrow \rho I), 0)\rho I), A[;\cdots;1]$

Consequently g: $(({}^{-}2 \downarrow \rho\omega), 0)\rho\omega$ is both the right and left identity-inducing function of catenate, and therefore $,/A$ is $(({}^{-}2 \downarrow \rho A), 0)\rho A$ whenever $0 = {}^{-}1 \uparrow \rho A$.

To repeat, a scalar function with right or left identity RI or LI has the right or left identity-inducing expression $({}^{-}1 \downarrow \rho\omega)\rho RI$ or $({}^{-}1 \downarrow \rho\omega)\rho LI$. In addition, identity-inducing expressions for the nonscalar primitives are as shown in **Table 1**.

Table 1 Identity-inducing expressions for the primitive functions.

Dyadic function		*Identity-inducing expression*	*Left-right*
Catenate	,	$((^{-}2 \downarrow \rho\omega), 0)\rho\omega$ (Note 1)	L R
Reshape	ρ	${}^{-}1 \downarrow \rho\omega$	L
Take	$\uparrow$	${}^{-}1 \downarrow \rho\omega$	L
Drop	$\downarrow$	$(0\lceil{}^{-}1 + \rho\rho\omega)\rho 0$	L
Compress	/	$({}^{-}1 \uparrow {}^{-}1 \downarrow \rho\omega)\rho 1$ (Note 1)	L
Expand	\	$({}^{-}1 \uparrow {}^{-}1 \downarrow \rho\omega)\rho 1$ (Note 1)	L
Rotate	ϕ	$(^{-}2 \downarrow \rho\omega)\rho 0$ (Note 1)	L
Transpose	⍉	$\iota 0\lceil{}^{-}1 + \rho\rho\omega$	L
Membership	ϵ	$({}^{-}1 \downarrow \rho\omega)\rho 1$ (Note 2)	R
Index of	ι	$\iota\lfloor/\iota 0$ (Note 3)	L
Domino	⌹	$(\iota 1 \uparrow \rho\omega)\circ.=\iota 1 \uparrow \rho\omega$	R
Encode	⊤	$\lfloor/\iota 0$	L
Decode	⊥		None
Deal	?		None
Format	⍕		None

Note 1: Relation (3) holds only for A of rank at least 2.
Note 2: Relation (3) holds only if all elements of A have boolean values.
Note 3: Relation (3) holds only if all elements of A have natural number values.

As for nonprimitive functions, consider first the simple derived functions, i.e., primitive operators applied to primitive functions. There would be advantages to being able to formally generate identity-inducing functions of all the simple derived functions from those of the primitive functions, but unfortunately that is not possible. Some results in that direction are

A. Following [13], if f is a primitive function with a right (left) identity-inducing function g, then $\circ.f$ has a right (left) identity-inducing function h if and only if g commutes with indexing, in which case $h\omega \leftrightarrow g0\rho\omega$. (The function g *commutes with indexing* if $\rho gA \leftrightarrow {}^{-}1 \downarrow \rho A$ whenever $0 = {}^{-}1 \uparrow \rho A$, and

$(gA)[I1;\cdots;IN] \leftrightarrow gA[I1;\cdots;IN;\iota 0]$

for all valid indices $I1$ through IN.)

In particular, if f is a primitive scalar function, then g commutes with indexing and the right or left identity-inducing expression of $\circ.f$ is identical to RI or LI, respectively.

B. Analogously, $f[\rho\rho\omega]$ has a (right or left) identity-inducing function h if and only if g commutes with indexing along all axes other than the $\rho\rho\omega$th, in which case

$h\omega \leftrightarrow g[\rho\rho\omega]\omega$.

C. The results for inner product are not as general. Some specific results are: e.f has the left identity-inducing function $ID\ 1 \uparrow \rho\omega$ for ID: $(\iota\omega)\circ. = \iota\omega$ if f is compress,

and also has identity-inducing functions based on *ID* for certain combinations of primitive scalar functions (e.g., $+.\times$, $\times.*$, and $\neq.\wedge$).

Evidently it is also not possible to formally generate identity-inducing functions for defined functions, except in a few simple cases. A practical scheme is to permit defined identity-inducing functions that can somehow be associated with defined functions (see, e.g., [14]. As for nonsimple derived functions (primitive operators applied to nonprimitive functions), apparently the only practical treatment is to define a function for any member of this class when its use requires an identity-inducing function and then apply the mechanism for defined functions. In fact, in view of the rather sparse results in (C), it may be prudent to treat all functions derived from inner product, reduction, and scan in this manner as well.

• *Empty results*

The other aspect of the problem of applying functions derived from the extended primitive operators to empty arguments occurs when the comparable results in present APL are empty. There is no problem for reduction and scan. If $0 = {}^{-}1 \uparrow \rho A$, then f/A is defined in terms of the identity-inducing function of f, whether or not other axes of A are of length zero, while $f\backslash A$ is A. If $0 \neq {}^{-}1 \uparrow \rho A$, then the definitions of f/A and $f\backslash A$ in Definitions 1 and 2 apply, whether or not A has axes of length zero. Difficulties occur in defining values for the following:

1. $A f[K] B$ and $f[K] B$ when an argument axis other than the Kth has length zero;
2. $A \circ .fB$ when an argument is empty; and
3. $A e.fB$ when an axis of A other than the last or an axis of B other than the first has length zero.

The difficulty in each case is due to the fact that there is nothing to which to apply the argument function f. [For example, what is the result of $(\iota 0) \circ .\rho 1\ 2\ 3$)?] In present APL these operators apply to a restricted class of functions for which the above results are derived formally from general shape and type rules, but there are no general shape and type rules for the extended operators. (Actually, the results of 1–3 above can be derived from formal identities for the nonscalar primitives $,\backslash\phi\epsilon\iota\boxplus\perp\top$, but that is not in general true of $\rho\uparrow\downarrow/\circledcirc?\tilde{\tau}$, where shapes of results can depend on values of the elements in element lists and, in some cases, the index origin.)

More [4, Sections 15, 16, 25, 29] has encountered this difficulty in a similar context, and has proposed a solution that constructs nonempty arguments to which the function arguments can be applied. More's scheme can be adapted to our purposes as follows:

Step 1 "Fill in" empty arguments by applying the function

$FILL\Delta IN{:}((\rho\omega)\lceil \sim (\iota\rho\rho\omega)\epsilon\alpha) \uparrow \omega$

to them, with left argument K for $f[K]$, and $\iota 0$ for $\circ .f$ and $e.f$;

Step 2 Apply the derived functions to the resulting nonempty arguments;

Step 3 Obtain the desired results by "emptying" or "vacating" the results along those axes where the arguments were filled. That is, if R is a result obtained by the first two steps and S is ρR, form T from S by replacing each 1 in S with 0 whenever the 1 is due to a "filled-in" axis of an argument; $T \uparrow R$ is then the result of the derived function.

For example, outer product for empty arguments would be defined in terms of outer product for nonempty arguments as follows (the first expression represents steps 1 and 2, the fourth step 3):

$FA \leftarrow (\iota 0) FILL\Delta IN\ A$

$FB \leftarrow (\iota 0) FILL\Delta IN\ B$

$IR \leftarrow FA \circ .fFB$

$A \circ .fB \leftrightarrow ((\rho IR \times 0 = (\rho\rho IR) \uparrow 0 = (\rho A), \rho B) \uparrow IR$

Thus in the case of $(\iota 0) \circ .\rho 1\ 2\ 3$, Step 1 yields $(,0) \circ .\rho 1\ 2\ 3$, Step 2 yields $1\ 3\ 0\rho 0$, and Step 3 yields $0\ 3\ 0\rho 0$.

As stated, More's scheme will not work in general for APL because there are functions that do not apply to arrays of zeros or arrays of blanks, such as the left argument of $\circledcirc$ in origin 1; see also "Numerical applications" in Section 4. (In array theory a result is always returned, and in those cases where the function does not apply to zeros or blanks the elements of the results are "faults.") This difficulty would be overcome in APL if it were possible to alter the contents of the "filled-in" arrays. Specifically, to each function f there could be an associated *fill-transforming* function h, just as there is an associated identity-inducing function, that would be applied to nonempty results of *FILLΔIN* whenever certain derived functions involving f are applied to empty arguments. For example, in the above definition of outer product for empty arguments, the intermediate result *IR* would be defined in terms of the fill-transforming function h as follows:

$IR \leftarrow (hFA) \circ .f(hFB)$

Thus the following would be inserted between Steps 1 and 2.

Step 1A Apply the fill-transforming function of f to the results of Step 1.

Note that the procedure is now consistent with the present evaluation of these operators for primitive scalar functions and empty array arguments. The fill-transforming function h for a primitive scalar function f can simply be defined by $h{:}(\rho\omega)\rho D$, where D is any scalar whose value is the domain of f. It is also worth noting that for many dyadic functions

such as the primitive structural functions it would be far more effective to have separate fill-transforming functions for the left and right arguments.

Further details of fill-transforming functions can be developed in similar ways to those of identity-inducing functions and are left to the reader.

3. Heterogeneous arrays

In this section heterogeneous arrays are assumed to be available in APL, i.e., arrays containing both numbers and characters as elements. This assumption has little effect on the considerations of the preceding section, and consequently the present discussion is mainly concerned with the primitives take and expand, and with fill.

There are several fairly obvious ways to define fill elements for heterogeneous arrays. One way is to distinguish the heterogeneous type in the same way that numeric and character types are distinguished, and to define the fill element to be a distinguished element that is neither numeric nor character. However, there does not appear to be an obviously useful class of scalars of heterogeneous type, as there is for both character and numeric types. (In fact, any analogy of heterogeneous arrays with numeric and character arrays will ultimately fail because heterogeneity is not actually a third distinct type, but is instead the lack of a distinctive type.) A second approach is to choose a fill element for heterogeneous arrays that is either a number or a character, but not necessarily zero or blank. Perhaps a special graphic could be created for that purpose. However, it would most likely be very cumbersome to account for such an exception in practical situations. A third approach is to use zero and blank as the fill elements for all arrays—homogeneous and heterogeneous—and to compute which is to be used from the arrays to be filled. For example, one possibility would be to use the blank as fill element for all heterogeneous arrays, thereby distinguishing between purely numeric arrays and arrays with at least one character element. Still another scheme, one that is suggested by array theory [3, 4], would be to use zero for fill in a nonempty array A if $1\uparrow,A$ is a number and blank if $1\uparrow,A$ is a character. (The use of the first element of $,A$ is suggested by the fact that every nonempty array A has a first element. One could just as well use the last element, or the middle, etc.)

Still another possibility, which is also suggested by array theory [3, 4], is that fill is not necessarily based on a single element. This section is primarily concerned with the notion of *fill arrays*, as opposed to fill elements.

To appreciate the appeal of more elaborate fill, consider a character matrix in present APL whose rows contain names. When expand is applied along the first axis of this matrix, new blank rows may appear, which consist of replications of the fill element (the scalar blank). In this case one could also conceive of the fill as a blank vector (or blank one-row matrix) and of the new blank rows as copies of that fill array. In present APL, where all arrays are homogeneous, these two viewpoints lead to the same result, but this is not necessarily true when heterogeneous arrays are permitted. Continuing the example, suppose that a numeric column is appended on the left of the character matrix of names, containing, perhaps, the ages of the people named in the corresponding rows of the character matrix. If the resulting matrix is expanded and a fill element is used, then each new row created by expand consists entirely of copies of that fill element. However, thinking in terms of fill arrays, the fill could typify every row of the matrix by having zero as its first element and blanks as its remaining elements, and each new row created by expand could be a copy of this fill array.

Applications such as this suggest the following definitions.

Definition 8

A nonempty array A is said to be *uniform with respect to the* Kth *axes* if all elements of

$$A[;\cdots;I1;\cdots;IN;\cdots;]$$

are either numbers or characters for each valid set of values of the scalars $I1$ through IN, where these scalars index the axes of A other than the Kth. For example, the matrix in the previous example is uniform with respect to the first axis because the first column consists entirely of numbers while all other columns consist entirely of characters.

In particular, A is said to be *uniform* if it is uniform with respect to all axes, i.e., if all elements of A are of the same type.

Definition 9

The *fill array of a nonempty array A with respect to the* Kth *axes* of A is defined to be

$$A[;\cdots;,1;\cdots;,1;\cdots;]$$

with every number replaced by zero and every character replaced by blank, where the 1s index the axes specified in K. (The use of ,1 instead of 1 has the useful effect that the rank of the fill array is the same as the rank of the array it fills.)

In terms of fill arrays, expand along the Kth axis of A would be defined just as it is now, except that the subarrays of the result corresponding to zeros in the left argument would be copies of the fill array with respect to axis K instead of replications of a fill element. For example, if

$$A \leftarrow 2\ 3\rho 1, \text{'}A\text{'}, 2, 3, \text{'}B\text{'}, 4$$

then

$$(1\ 1\ 0\backslash A)[3;] \leftrightarrow 0,\text{'}\ \ \text{'},0$$

$$(1\ 1\ 1\ 0\backslash A)[;4] \leftrightarrow 0\ 0$$

In general, if A is uniform with respect to the Kth axis, then so is $I\backslash[K]A$.

The only other primitive function that uses fill for nonempty arrays is take. Unlike expand, take does not apply

along an axis. (The extended axis operator, Definition 3, if available, would apply to take, and the definition of $N \uparrow [K]A$ would follow directly from the one for $N \uparrow A$ suggested below.) However, based on common usage of the take function, it is assumed that there are implied axes along which take applies, and that these axes depend on the arguments. Specifically, the implied axes for $N \uparrow A$ are $(N \neq \rho A)/\iota\rho\rho A$, i.e., those for which there is a change in length. Proceeding in the same manner as for expand, $N \uparrow A$ would be defined so that each subarray of the result that lies along axes $(N \neq \rho A)/\iota\rho\rho A$, and would presently be a replication of fill elements, is instead a copy of the fill array with respect to axes $(N \neq \rho A)/\iota\rho\rho A$. For example, if

$A \leftarrow 2\ 3\rho 1, \text{'}A\text{'}, 2, 3, \text{'}B\text{'}, 4$

then

$(3\ 3 \uparrow A)[3;] \leftrightarrow 0,\text{'}\ \ \text{'},0$

$(2\ 4 \uparrow A)[;4] \leftrightarrow 0\ 0$

and $3\ 4 \uparrow A$ is filled with copies of $1\ 1\rho 0$. As with expand, if A is uniform with respect to axes $(N \neq \rho A)/\iota\rho\rho A$, then so is $N \uparrow A$.

The next question concerns empty arrays and is suggested, for instance, by the previous example concerning matrices whose first column is numeric and whose remaining columns are character. In such an application it would not be unreasonable to expect that all such matrices have the same fill arrays with respect to the first axis, whether the matrices are empty or nonempty. Evidently, such a requirement of empty arrays can be met only if fill arrays and the axes with respect to which they apply are somehow associated with empty arrays. (For example, one can think of an empty array in present APL as having associated with it a one-element fill array that is either a zero or a blank, and that applies to all axes.) Thus primitive functions that produce empty arrays would be responsible to establish such associations for nonempty arguments and to transform existing associations for empty arguments. (In present APL the primitives are responsible for establishing the types of empty results, and follow the type rule stated in [12, Section 2.4].)

Before we deal with the establishment and transformation of the suggested associations, it may be worthwhile to give more thought to the underlying idea. Specifically, there is the following question: What is to be done when an empty array is associated with a fill array and one particular set of axes, but a primitive is applied that requires a fill array with respect to another set of axes? One answer is to associate all possible fill arrays and their axes with each empty array, but a more practical answer, perhaps, is to associate one array, from which fill arrays with respect to all sets of axes can be constructed. We follow the latter suggestion, which means that with each empty array there is associated a second array but not a set of axes, and that the associated array is not necessarily a fill array with respect to any set of axes, but is the source from which those fill arrays can be constructed. Moreover, in order to simplify the definitions by removing unnecessary distinctions between empty and nonempty arrays, an array will also be associated with every nonempty array, one from which fill arrays with respect to all sets of axes can also be constructed. If all fill arrays of a nonempty array A can be constructed from the array associated with A, then a reasonable choice for that associated array is the one obtained from A by replacing every number in A with a zero and every character with a blank. As was pointed out in the introduction, the array associated with A can be called the *type array* of A, or simply the *type* of A.

The situation, then, is as follows.

Definition 10

The type TA of the nonempty array A is defined to be A with every number replaced by a zero and every character replaced by a blank.

The type TA of an empty array A is nonempty, and is to be determined from the types of the nonempty arrays from which A is constructed, as well as the primitive functions and derived functions used in the construction. Using the concept of fill arrays with respect to certain axes as a guide, it will be required that $\rho TA \leftrightarrow 1\lceil\rho A$.

Definition 11

The fill array with respect to the Kth axes of any array A (empty or nonempty) whose type is TA is defined to be

$TA[;\cdots;,1;\cdots;,1;\cdots;]$

where the ,1s index the axes specified in K. The following is a detailed proposal for nonscalar types of empty arrays.

There are five primitives that can produce empty arrays from nonempty ones: compress, take and drop, reshape, and indexing. The following definitions apply for both empty and nonempty arguments, and for empty results, but not necessarily for nonempty results.

Compress $(I/[K]A)$ is defined for empty results so that if TA is the type of A, then the type of an empty result is the fill array of TA with respect to the Kth axes.

Analogously, the primitives take $(N \uparrow A)$ and drop $(N \downarrow A)$ are defined so that the type of an empty result is the fill array of TA with respect to axes $(N \neq \rho A)/\iota\rho\rho A$ and $(N \neq 0)/\iota\rho\rho A$, respectively.

As for reshape $(S\rho A)$, and again following common usage of the function, the type of an empty result is

$(1\lceil S)\rho TA[,1;,1;\cdots;,1]$

unless $(\sim S \in 0\ 1)/S$ equals $(\sim(\rho A) \in 0\ 1)\ \rho A$, in which case the type of empty result is $(1\lceil S)\rho TA$.

And finally, for indexing $(A[I1;\cdots;IJ;\cdots;IN])$, the implied axes of application are those axes J for which either

$IJ \leftrightarrow \iota(\rho A)[J]$ or IJ is elided, and the type of an empty result is $TA[K1;\cdots;KJ;\cdots;KN]$, where KJ is ,1 if J is an implied axis of application, and is IJ otherwise.

As for expand $(I\backslash[K]A)$ of an empty array A, if the result is also empty, then its type is TA if I contains no 1s, and $I\backslash[K]TA$ otherwise. If the result is nonempty, it is defined in the same way as for nonempty A, except that the fill array is that of TA with respect to the Kth axis. The definition of $N \uparrow A$ is analogous. If the result is empty, then its type is $(1 \lceil N) \uparrow TA$, while if the result is nonempty it is defined in the same way as for nonempty A, except that the fill array is that of TA with respect to axes $(N \neq \rho A)/\iota\rho\rho A$.

Rules to produce the types of empty results of the other nonscalar primitives can be derived in a fairly straightforward manner, and we only illustrate them here; for example, if TA is the type of the empty array A, then ϕTA is the type of ϕA, while $(\phi\rho A)\ \rho 0$ is the type of $\boxplus A$ (which is defined for matrices A with $\geq/\rho A$), and TA is the type of B,A if B is also empty. In addition, for a primitive scalar function f, $(\rho TA)\ \rho 0$ is the type of the empty results fA, AfB, and BfA, where B is either a scalar or is empty and of the same shape as A. There are also straightforward rules to define the types of empty results of derived functions that are valid in present APL.

As for the extended operators, identity-inducing and fill-transforming functions apply just as well in the presence of fill arrays as arrays of fill elements. In fact, their use might be enhanced—particularly for defined functions—because they would apply to a richer class of arrays. In particular, $K\ FILL\Delta IN\ A$ would be identical to the fill of A with respect to the Kth axes.

Each of the fill element definitions at the beginning of this section can be analyzed in the same way as the fill arrays definition. For example, the above three-point scheme for type and fill arrays would appear as follows for the last of the suggested fill element definitions.

Definition 12

The type TA of the nonempty array A is defined to be $((\rho\rho A)\ \rho 1)\rho 0$ if $1\uparrow,A$ is a number and $((\rho\rho A)\rho 1)\rho' '$ if $1\uparrow,A$ is a character.

The type TA of an empty array A is to be determined by analyzing the appropriate primitives, as with fill arrays. Since it is formally convenient for the rank of TA to be the same as the rank of A, it would be required that $(\rho TA) \leftrightarrow (\rho\rho A)\rho 1$.

Definition 13

The fill with respect to the Kth axes of the array A (empty or nonempty) whose type is TA is defined to be

$$((\rho A)f \sim (\iota\rho\rho A)\epsilon K)\rho TA$$

The rest of the analysis is left to the reader, since it only involves defining certain primitives in ways very similar to present APL.

4. General arrays

It appears (so far, at least) that APL general array implementations fall into two general categories, which are usually referred to as permissive systems and strict systems. Permissive systems are discussed first, since they tend to contain more extensive general array features than strict systems, and as far as the topics of this paper are concerned, are more closely related to the preceding section.

• *Permissive general arrays*

In this section it is assumed that APL has been extended to include extended operators, heterogeneous arrays, and (permissive) general arrays. Our purpose is to examine the concepts of identity-inducing functions, fill-transforming functions, and fill arrays in this context. For convenience we begin by introducing some of the basic functions that apply to general arrays.

First of all, there are new primitive functions denoted by $\subset\omega$ and $\supset\omega$ and called *enclose* and *disclose*, such that for any array A, $\subset A$ is a scalar *holding* A, and $\supset\subset A$ is A. An array that contains no enclosed elements is said to be *flat*. Both the enclose and disclose of a flat scalar A are assumed to be identical to A. The function called *LIST* produces a flat vector of the numbers and characters in its argument. That is, $LIST\ A$ is $,A$ if A is flat, and otherwise $LIST\ A$ is $(LIST \supset (,A)[1])$, followed by $LIST \supset (,A)[2]$, etc.

With regard to operators, the existence is assumed of the primitive scalar extension operator called *each* and denoted by ¨. That is, for every monadic function f the shape of $f¨A$ is the same as the shape of A, and every element of $f¨A$ is obtained by disclosing the corresponding element of A, applying f to the disclosed contents, and enclosing the result. The description for dyadic f is similar. Moreover, f¨ is applied to empty arrays by using f's fill-transforming function h and the technique in Section 2. In the monadic case,

$$f¨B \leftrightarrow (\rho f¨B) \uparrow f¨h(\iota 0)FILL\Delta IN\ B$$

and the dyadic case is similar. In addition, if g is the identity-inducing function of the dyadic function f, then g¨ is the identity-inducing function of f¨.

The axis operator is assumed to apply to both enclose and disclose. The elements of $\subset[K]A$ hold the subarrays of A along the Kth axes, while the subarrays of $\supset[K]B$ along the Kth axes are the elements held by B. Consequently, for nonempty A and B

$$A \leftrightarrow \supset[K] \subset [K]A \tag{4a}$$

$$B \leftrightarrow \subset[K] \supset [K]B \tag{4b}$$

These identities are important because the functions involved in them provide fundamental transitions between simple and general arrays. More generally

$$f[K]A \leftrightarrow \supset[L]f¨ \subset [K]A \tag{5a}$$

$$f\ddot{}B \leftrightarrow \subset[L]f[K] \supset [K]B \qquad (5b)$$

for any appropriate function f; L is the set of axes along which the individual results of f[K] lie. Disclose of a nonscalar and nonempty array is defined to be disclose along those axes which position the individual disclosed elements along the last axes of the result. Note that $\subset A \leftrightarrow \subset[\iota\rho\rho A]A$.

Finally, every primitive scalar function f is assumed to be *pervasive*, in that $f\ddot{}A$ is identical to fA for every appropriate array A. Consequently, the primitive scalar functions are referred to instead as the primitive pervasive functions.

The concepts of type and fill arrays introduced in Section 3 can be extended to APL systems with both general and heterogeneous arrays. In this section the following four definitions of type and fill for a nonempty array A are considered.

Definition 14—flat fill elements
The type and fill element of A is $((\rho\rho A)\rho 1)\rho 0$ if $1 \uparrow LIST\ A$ is a number and $((\rho\rho A)\rho 1)\rho' '$ if $1 \uparrow LIST\ A$ is a character.

Definition 15—nested fill elements
The type and fill element of A is obtained from $A[,1;,1;\cdots;,1]$ by replacing each number in this scalar with a zero and each character with a blank.

Note: This definition, which introduces general arrays as fill elements, is based on the array theory concept of prototype [4]; if TA is the type of A by Definition 15, then $\supset TA$ is the prototype of A.

Definition 16—flat fill arrays
The type TA of A is obtained from A by replacing each element E with a scalar blank if $1 \uparrow LIST\ E$ is a character and with a scalar zero if $1 \uparrow LIST\ E$ is a number. The fill array of A with respect to the Kth axes is

$$TA[;\cdots;,1;\cdots;,1;\cdots;]$$

where the 1s index the axes specified in K.

Definition 17—nested fill arrays
The type TA of A is obtained from A by replacing each character in A with a blank and each number with a zero. The fill array of A with respect to the Kth axes is

$$TA[;\cdots;,1;\cdots;,1;\cdots;]$$

where the 1s index the axes specified in K.

Note that Definitions 14 and 15 are identical for flat arrays, as are Definitions 16 and 17. Moreover, Definitions 16 and 17 for flat arrays provide the same fill arrays as those in Definition 9.

The definitions of uniformity and uniformity with respect to specific axes (Definition 8) carry over to general arrays simply by requiring that the word "type" in those definitions refer to one of Definitions 14 to 17. Enclose and disclose along axes provide transformations between nonempty uniform arrays and nonempty arrays that are uniform with respect to specific axes. That is, if A is uniform with respect to the Kth axes, then $\subset[K]A$ is uniform, while if B is uniform, then $\supset[K]B$ is uniform with respect to the Kth axes.

Each of the above definitions can be extended to empty arrays by the technique used in Section 3, but care must be taken in accounting for the behavior of enclose and disclose along axes. For example, the expression $\subset[K]0\ 0\ 1\rho 0$ indicates the potential for forming empty arrays that hold empty arrays (K equals 2 3), empty arrays that hold nonempty arrays (K equals 3), and nonempty arrays that hold empty arrays (K equals 1 2). Whether or not empty arrays holding other arrays can actually be formed depends on which type definition is adopted, for in such cases only the type can provide information about the arrays to be held.

The first problem to be discussed is that of extending the definitions of $\subset A$ and $\supset B$ to empty A and B, as well as those of $\subset[K]A$ and $\supset[K]B$, so as to maintain identities (4a) and (4b). The behavior of functions derived from the each operator on empty arguments can subsequently be derived from (5b) by assuming that that identity holds for empty B and applying the evaluation procedure in "Empty results."

For each of the above types, proposed definitions of $\subset A$, $\supset B$, $\subset[K]A$, and $\supset[K]B$ for empty arguments A and B, and the validity of relations (4a) and (4b), are as follows, where TA and TB are the types of A and B, and ZB is the two-element vector $0,' '$.

1. Based on Definition 14, $\subset A$ is a scalar with type TA and $\supset B$ is identical to B. More generally, $\subset[K]A$ has shape $(\sim(\iota\rho\rho A)\epsilon K)/\rho A$ and type $(\rho\rho \subset [K]A)\rho 1)\rho TA$, while $\supset[K]B$ has shape $S[\not\Downarrow \not\Downarrow(\iota\rho S)\epsilon K]$, where $S \leftrightarrow (\rho B)$, $(\rho, K)\rho 1$, and type $((\rho\rho \supset [K]B)\rho 1)\rho TB$. For example,

   $$\subset[2\ 3]0\ 0\ 2\rho ZB \leftrightarrow 0\rho ZB[1]$$

   and

   $$\supset[3]0\ 0\rho \subset ZB \leftrightarrow 0\ 0\ 1\rho ZB[1]$$

   In general, (4b) holds but (4A) fails, e.g.,

   $$\supset[3] \subset [3]0\ 0\ 2\rho ZB \leftrightarrow 0\ 0\ \rho ZB[1]$$

2. Definition 15 provides a richer class of types for nested arrays than flat arrays. Based on this definition, $\subset A$ is a scalar with type $\subset(\rho A)\rho TA$, while $\supset B$ has shape $(\rho B),\rho \supset (\iota 0)\rho TB$ and type $((\rho\rho \supset B)\rho 1)\rho \supset TB$. More generally, $\subset[K]A$ has shape as in (1) and type $((\rho\rho \subset [K]A)\rho 1)\rho \subset (\rho A)[K]\rho TA$, while $\supset[K]B$ has shape as in (1) but with $S \leftrightarrow (\rho B)$, $\rho \supset TB$, and type $((\rho\rho \supset [K]B)\rho 1)\rho \supset TB$. For example,

   $$\subset[2\ 3]0\ 0\ 2\rho ZB \leftrightarrow 0\rho \subset 0\ 2\rho ZB[1]$$

   where $0\rho \subset 0\ 2\rho ZB[1]$ is the empty vector with type

$\subset 0\ 2\rho ZB[1]$ (whose content $0\ 2\rho ZB[1]$ has type $ZB[1]$), and

$\supset[3]0\ 0\rho \subset ZB \leftrightarrow 0\ 0\ 2\rho ZB[1]$

where $0\ 0\ 2\rho ZB[1]$ has type $ZB[1]$. In general, (4a) holds but (4b) fails, e.g.,

$\subset[3] \supset [3]0\ 0\rho \subset ZB \leftrightarrow 0\ 0\rho \subset (\rho ZB)\rho ZB[1]$

3. Definition 16 provides a richer class of types for flat arrays than nested arrays. Based on this definition, $\subset A$ is a scalar with type $(\iota 0)\rho TA$ and $\supset B$ is identical to B. More generally, the shapes of $\subset[K]A$ and $\supset[K]B$ are as in (1), the type of $\subset[K]A$ is $TA[;\cdots;,1;\cdots;,1;\cdots;]$ (where the 1s index the Kth axes), and the type of $\supset[K]B$ is $(1 \lceil \rho \supset [K]B)\rho TB$. The examples for Definition 14 are the same here. In general, (4b) holds but (4a) fails.
4. Based on Definition 17, $\subset A$ is a scalar with type $\subset(\rho A) \uparrow TA$, while $\supset B$ has shape (ρB), $\rho \supset (\iota 0)\rho TB$ and type $\supset D\ddot{}\,TB$, where D: $(1 \lceil \rho\omega) \uparrow \omega$. More generally, $\subset[K]A$ has shape as in (1), $\supset[K]B$ has shape as in (1) but with $S \leftrightarrow (\rho B)$, $\rho \supset (\iota 0)\rho TB$, the type of $\subset[K]A$ is $E\ddot{} \subset [K]TA$, where E: $(\rho A)[K] \uparrow \omega$, and the type of $\supset[K]B$ is $\supset[K]D\ddot{}\,TB$. For example,

$\subset[2\ 3]0\ 0\ 2\rho ZB \leftrightarrow 0\rho \subset 0\ 2\rho ZB$

where $0\rho \subset 0\ 2\rho ZB$ is the empty vector with type $\subset 0\ 2\rho ZB$ (whose content $0\ 2\rho ZB$ has type $1\ 2\rho ZB$), and

$\supset[3]0\ 0\rho \subset ZB \leftrightarrow 0\ 0\ 2\rho ZB$

where $0\ 0\ 2\rho ZB$ has type $1\ 1\ 2\rho ZB$. Both (4a) and (4b) hold in general.

Note that each definition of $\subset[K]\omega$ is consistent with the three-step evaluation procedure in "Empty results."

We turn now to the identity-inducing and fill-transforming functions of the primitive pervasive functions. Recall that for a primitive scalar function f with a (right or left) identity element I, and for a scalar D whose value is the domain of f, the identity-inducing function g and the fill-transforming function h can be defined as follows:

$$\begin{aligned} &\text{g: } (\bar{}1 \downarrow \rho\omega)\rho I \\ &\text{h: } (\rho\omega)\rho D \end{aligned} \tag{6a}$$

The question to be considered is whether or not these definitions should be modified to reflect nested types for the primitive pervasive functions. Presumably, the new definitions would be as follows:

$$\begin{aligned} &\text{g: } I + (\bar{}1 \downarrow \rho\omega)\rho(TYPE\omega) \neq TYPE\omega \\ &\text{h: } D + \omega \neq \omega \end{aligned} \tag{6b}$$

where $TYPE\ A$ is the type of A for all A. (The expression $X \neq X$ has the effect of replacing each number and each character in X with a zero.)

The proposed definition for the fill-transforming functions is reasonable because it is consistent with the description of pervasiveness in terms of the each operator and with the definition of the each operator for empty array arguments. The proposed definition for the identity-inducing functions, however, is not acceptable. For example, consider the array

$A \leftarrow 2\ 1\rho(\subset 1\ 2), \subset 2\ 2\rho 3\ 4\ 5\ 6$

and the relation (2) of Section 2 for +. If types are defined by Definition 15, then

$A[;\iota 0] \leftrightarrow 2\ 0\rho \subset 0\ 0$

and therefore, according to the identity-inducing function (6b),

$+/A[;\iota 0] \leftrightarrow 2\rho \subset 0\ 0$

Therefore relation (2) fails for A because it requires evaluating $(2\ 2\rho 3\ 4\ 5\ 6)+0\ 0$, which evokes a rank error. Relation (2) does not fail for A and Definition 17 because

$A[;\iota 0] \leftrightarrow 2\ 0\rho(\subset 0\ 0), \subset 2\ 2\rho 0$

and

$+/A[;\iota 0] \leftrightarrow (\subset 0\ 0), \subset 2\ 2\rho 0$

In this case, however, the identity-inducing expression for $\circ.+$, which is simply 0, cannot be produced by applying the method described in (A), "Nonempty results," to the identity-inducing expression for +. If we restrict our attention to uniform arrays, then Definitions 15 and 17 have the same effects with respect to the proposal at hand, relation (2) holds, and (A) in "Nonempty results" will produce an identity-inducing expression of $''.+$. However, there is no point in accepting such a restriction because the identity-inducing expressions for the primitive scalar functions given in (6a) can also be used for the primitive pervasive functions, and apply for nonuniform general arrays as well as uniform ones. These identity-inducing expressions are evidently the most generally useful ones.

With regard to operators, there is another set of straightforward definitions for the extended primitive operators. This set, which depends on the presence of general arrays, can be defined simply by changing the appropriate lines of Definitions 1–5.

1. Reduction

$f/ \leftrightarrow A[;\cdots;\ 1]f\ddot{}\ f/A[;\cdots;\ 1 \downarrow \iota N]$

2. Scan

Same definition, but in terms of the new reduction definition.

3. Axis

Unchanged.

4. *Outer product*

$$(A \circ .\mathrm{f} B)[I1;\cdots;IN;J1;\cdots;JM] \leftrightarrow A[I1;\cdots;IN]\mathrm{f}\ddot{}B[J1;\cdots;JM]$$

5. *Inner product* (in terms of the new definition of reduction)

$$(A\mathrm{f}.\mathrm{g}B)[I1;\cdots;IN;J1;\cdots;JM] \leftrightarrow \mathrm{f}/A[I1;\cdots;IN;]\mathrm{g}\ddot{}B[;J1;\cdots;JM]$$

The development of identity-inducing functions in Section 2 can be carried out for this definition of reduction as well; in fact, this is the definition used in [13]. It should be clear that g is the (right or left) identity-inducing function of f with respect to reduction as in Definition 1 if and only if g¨ is the (right or left) identity-inducing function of f with respect to this reduction definition. The above discussion of nested types and identity-inducing functions for the primitive pervasive functions applies equally well to this definition, with the same conclusion. Unlike Definitions 1–5, the shape rules for the present operators also apply to these extensions. However, the type rules do not, so that fill-transforming functions and the three-step evaluation procedure in "Empty results" are just as necessary here for producing types of empty results in a consistent way.

• *Strict general arrays*

For the purposes of this paper, the main differences between a strict general array system and a permissive system are that, in a strict system,

a. Enclose of a simple scalar is not identical to that simple scalar, e.g., $\subset 3 \leftarrow/\rightarrow 3$.
b. The primitive scalar functions are not pervasive.
c. General arrays are uniform in the sense that if at least one element of an array is nested, then all elements must be nested.

In this section it is assumed that APL has been extended to include extended operators (Definitions 1–5) and (strict) general arrays.

The basic point to be made here is that permissive general arrays are heterogeneous, while strict general arrays are homogeneous and therefore can be considered a third APL type, along with simple numeric arrays and simple character arrays. Consequently, a nested scalar (say *GF*) could be chosen for fill in general arrays that would be the counterpart of 0 for simple numeric arrays and ' ' for simple character arrays. In analyzing this definition we follow a parallel course to the section on permissive general arrays so as to make comparisons easy: A strict general array system may in fact not contain some of the features discussed here.

The definitions of the each operator, as well as enclose and disclose along axes, can be included in a strict system and identities (4a) and (4b) hold for nonempty A and B. It should be noted, however, that not all arrays B are in the domain of $\supset[K]\omega$ because the result must be homogeneous.

Following the form established for the permissive system alternatives, we have the following definition of type and fill for a nonempty array A.

Definition 18—strict fill elements

The type and fill element of A is the $((\rho\rho A)\rho 1)\rho 0$ if $1 \uparrow A$ is a number, the $((\rho\rho A)\rho 1)\rho' '$ if $1 \uparrow A$ is a character, or $((\rho\rho A)\rho 1)\rho GF$ if $1 \uparrow A$ is nested.

This definition can be extended to empty arrays by the technique in Section 3. Based on Definition 18, the proposed definitions of $\subset A$, $\supset B$, $\subset[K]A$, and $\supset[K]A$ for empty arguments A and B, and the validity of identities (4a) and (4b), are as follows, where TA and TB are the types of A and B.

5. Based on Definition 18, $\subset A$ is a scalar with type GF and $\supset B$ is identical to B. More generally, $\subset[K]A$ and $\subset[K]B$ have the same shapes as described for Definition 14. The type of $\subset[K]A$ is GF, while the type of $\supset[K]B$ is $((\rho\rho \supset [K]B)\rho 1)\rho GF$. As with Definitions 14 and 16, (4b) holds but (4a) fails in general.

A recommendation for the value of the fill element GF is presented in the next section.

• *Numerical applications*

We turn now to a less formal question, namely, the utility of the more elaborate fill definitions in actual applications. In this section nested types and fill are examined in the context of numerical applications, where it is not uncommon for fill to become involved in computations. The effects of Definitions 15 and 17 are examined in terms of a class of numerical applications that can be handled particularly well in APL with extended operators and general arrays, which we call alternate arithmetic applications. This class of applications also provides some insight into identity-inducing and fill-transforming functions, and we begin there.

To illustrate these applications, we first point out it is not uncommon for an algorithm using floating-point arithmetic to behave unacceptably because that arithmetic is only an approximation to real number arithmetic. In such cases it would be very helpful to execute the algorithm with a more accurate arithmetic such as rational number arithmetic or higher-precision floating-point arithmetic. Either of these alternate arithmetics can be employed simply by defining the desired arithmetic functions in an appropriate way and substituting the names of these functions for the corresponding primitive arithmetic symbols in the algorithm's definition. Evidently this can lead to derived functions defined in terms of primitive operators and defined functions, and thereby to the problems discussed in Section 2.

The first point to be discussed is actually a continuation of the discussion of permissive general arrays and concerns the definition of the identity-inducing functions of the primitive

pervasive functions (6b) when types are based on either Definition 15 or 17. It can happen that an alternate arithmetic function is simply the corresponding pervasive primitive, and in such cases the function in (6b) produces the desired effect. Examples include vectorspace addition (on which is based one of the principal supporting arguments for that definition) and rational number multiplication. In vectorspace arithmetic all arrays A under consideration are uniform with all elements holding vectors of a given length, and this would be true as well for values of $+/A[;\iota 0]$ if those values are based on (6b), but would not be true of (6a). Rational number arithmetic and $\times/A[;\iota 0]$ are similar. In most cases, however, an alternate arithmetic function is not simply a primitive, and therefore the identity-inducing functions for the pervasive primitives are not generally relevant to alternate arithmetic applications; the required general solution to the problem of providing identity-inducing functions for alternate arithmetics will involve a mechanism for controlling the behavior of reduction for defined functions, such as the one proposed in Section 2. Moreover, even when the alternate arithmetic function can be expressed as a pervasive primitive, that primitive is being used in a restricted context. It therefore seems appropriate that a defined function such as *PLUS*: $\alpha + \omega$ or *TIMES*: $\alpha \times \omega$ should be used instead and the desired reduction behavior obtained through whatever mechanism is available for defined functions.

Rational arithmetic provides a significant example of the problem that can occur when fill-transforming functions are not used in the evaluation procedure in "Empty results" for producing results of derived functions in empty arguments. Arrays of rational numbers are conveniently represented by uniform general arrays whose elements all hold two-element vectors, the first of which represents the numerator, and the second, the denominator. The fill for such an array is, according to Definition 15 or 17, an array whose elements all hold the two-element vector 0 0. But 0 0 is not a valid representation of a rational number because the denominator is 0, and therefore the fill is not a valid array of rational numbers. Consequently, functions that perform rational number arithmetic will evoke a domain error when applied to the fill. Fill-transforming functions are required here: an effective one is $(\rho\alpha)\rho \subset 0\ 1$.

The fill provided by Definitions 15 and 17 can be pleasing in appearance for applications of uniform arrays, but can have detrimental effects when the arrays are nonuniform. In general, Definitions 14 and 16 provide more effective fill than Definitions 15 and 17 for numerical applications. To illustrate this point, consider polynomial arithmetic, which is another instance of alternate arithmetic. Polynomials can be represented by *coefficient vectors* C whose corresponding polynomials are $+/C \times \omega^{*}\iota\rho C$ (in zero origin). The terms of a polynomial are

$$C[0] \qquad C[1] \times \omega \qquad C[2] \times \omega^{*}2$$

etc., and the terms are said to be in increasing order (with respect to powers of ω). In many applications it is preferable for the terms to be in decreasing order:

$$D[0] \times \omega^{*}2 \qquad D[1] \times \omega \qquad D[2]$$

in which case the polynomials are expressed as $+/D \times \omega^{*}\phi\iota\rho D$. General arrays permit analogous representations of polynomials of several variables, i.e., when ω stands for a vector instead of a scalar. The coefficient vectors of such polynomials, arranged in increasing order, would be general arrays C for which $\rho\rho \supset C[I] \leftrightarrow I$ for every scalar index I, and the terms would be

$$C[0] \qquad (\supset C[1]) \times \omega \qquad (\supset C[2]) \times \omega \circ . \times \omega$$

etc. The polynomial itself could be expressed as

$$+/LIST\ C \times ((\circ .\times)\ddot{})\backslash 1, (^{-}1 + \rho C)\rho \subset \omega$$

As in the case of a single variable, decreasing order of the multiple-variable terms is preferred in many applications:

$$(\supset D[0]) \times \omega \circ . \times \omega \qquad (\supset D[1]) \times \omega \qquad D[2]$$

If A and C are coefficient vectors of two polynomials of a single variable whose terms are in increasing order, the sum of the two polynomials has coefficient vector

$$(M \uparrow A) + (M \leftarrow (\rho A)\lceil\rho C) \uparrow C \tag{7a}$$

The fill provided in this expression by all four definitions has a single element, which is 0. For example, the sum of the polynomials with coefficient vectors 2 5 and $^{-}1\ 0\ 3$ has coefficient vector $2\ 5\ 0 + {}^{-}1\ 0\ 3$, or $1\ 5\ 3$. If the terms were organized in decreasing order, the sum would have coefficient vector

$$(M \uparrow B) + (M \leftarrow -(\rho B)\lceil\rho D) \uparrow D \tag{7b}$$

Expression (7a) also applies when A and C are coefficient vectors of two polynomials of several variables with terms in increasing order. Once again, the fill provided by all four definitions has a single element 0 because the first element of a coefficient vector is a simple scalar. For example, the sum of the polynomials with coefficient vectors $A \leftarrow 2, \subset 5\ 6$ and $B \leftarrow {}^{-}1, (\subset 0\ 1), \subset 2\ 2\rho 3\ 5\ 0\ {}^{-}6$ has coefficient vector $(A, 0) + B$, or $1, (\subset 5\ 7), \subset 2\ 2\rho 3\ 5\ 0\ {}^{-}6$. However, if the terms are in decreasing order, only Definitions 14 and 16 consistently provide a single-element fill. Using Expression (7b) and either Definition 15 or 17, the sum of the coefficient vectors $A \leftarrow (\subset 5\ 6), 2$ and $B \leftarrow (\subset 2\ 2\rho 3\ 5\ 0\ {}^{-}6), (\subset 0\ 1), {}^{-}1$ would be produced by evaluating the expression

$$((\subset 0\ 0), A) + B$$

($\subset 0\ 0$ is the fill) which evokes a rank error when evaluating $0\ 0 + 2\ 2\rho 3\ 5\ 0\ {}^{-}6$.

In this application the nested fill could be avoided by replacing Expression (7b) with

$$\phi(M \uparrow \phi B) + (M \leftarrow (\rho B)\lceil\rho D) \uparrow \phi D$$

but simple scalar elements are not always so readily available. The primitive scalar (or pervasive) functions are the building blocks for all numerical applications and, with respect to these functions, only flat scalars are *universal arguments*, i.e., compatible with all (numeric) arrays. As a consequence, the most generally useful fill arrays for numerical primitives are flat.

This completes the analysis of the alternative arithmetic applications and permissive general arrays. A similar analysis for a strict general system leads to the recommendation that the fill element *GF* be given the value ⊂0.

• *Nonnumerical applications*

Fill plays a more static role in nonnumerical applications than in numerical ones, but even there the definitions of fill arrays and nested fill elements in Section 4 can cause difficulties in applications of nonuniform arrays. For example, employee information may be kept in a general array of rank 1 with one element for each employee. As is common in data base applications, there may also be special information, pertaining, perhaps, to the organization of the data, or containing a separate representation of employee names for efficient sorting, or any of a number of things. If this special information is stored in reserved elements at the front of the general array, as would be natural, then the fill will be the type of some portion of the special information and will most likely bear little resemblance to the organization of the employee data, which would probably be quite regular. In addition, it could be quite expensive to detect the fill and replace it, should that be necessary. Evidently a simple fill element would be best here, since it is economical as far as space is concerned, and it is also easily detected in case it is to be replaced.

5. A fill primitive

Iverson [10, Section F] points out in his discussion of function rank that in the application of a general nonscalar APL function, the axes of an argument ω will be split at some point K such that the function is applied to each subarray along axes $K \downarrow \iota\rho\rho\omega$. In the monadic case the *rank* of the function is defined to be the nonnegative integer *MF* for which $(\rho\rho\omega) - MF$ equals K, and the dyadic case is similar. We call these subarrays the *units* to which the function applies, and we say that the complementary axes $K \uparrow \iota\rho\rho\omega$ represent a *collection* of units. Applications often deal with collections of *logical units*, i.e., subarrays which the user views as units, and Iverson goes on to describe a *rank operator* by which the ranks of functions can be controlled so that the resulting units to which functions apply coincide with the logical units of the applications.

As we have seen, a potential use of fill is to provide a means of associating "typical" units with empty collections so that they will be processed in a consistent manner with nonempty collections, and fill derived from type arrays guarantees this consistency for sequences of primitives in which each function applies to a collection of units and produces a collection of units. In particular, the units may be transformed as the processing proceeds. Array theory provides fill elements, and thereby a way of associating "typical" elements with empty general arrays (hence the name prototype). Thus, in the context of extended APL with fill elements, consistent behavior for empty collections of nonscalar units cannot be expected.

Unfortunately, not all uses of fill have to do with representing typical units. In most applications not only are collections of units processed, but the units themselves as well, and it is very often the case that the units have no logical units of their own. For example, an employee data base may keep a pair of arrays as information on each person, such as name (a character vector) and salary history (a numeric matrix). Consequently a character vector, numeric matrix pair may be viewed as a logical unit of the data base, but neither a character vector nor a numeric matrix would be considered as the logical unit of a pair. Another example is provided by the coefficient vectors described in "Numerical applications," where the enclosed elements are all of different ranks. As illustrated in Section 4, in processing the units themselves, or when arrays do not consist entirely of logical units, the most useful type definitions are those that provide the most universally applicable, elementary fill quantities.

The goal of this section is to reconcile these two disparate goals, i.e., to permit "typical" units of empty collections to be represented by fill, and at the same time to provide elementary fill for irregular arrays. The key is to remove the dependency of fill on type arrays and to require the primitive functions to propagate fill, which is consistent with present APL (see the "type" rules in [12, Section 2.4]). We begin again with present APL, but describe things so as to prepare the way for the admission of heterogeneous and general arrays, and consequently the descriptions at first contain redundancies for present APL.

A useful classification of APL arrays is as follows. The class NN of APL arrays consists of all arrays whose fill is a scalar zero and, if nonempty, whose elements are all numbers, while the class CC of APL arrays consists of all arrays whose fill is a scalar blank and, if nonempty, whose elements are all characters. The class NA of APL arrays consists of all empty arrays together with all nonempty arrays whose elements are all numbers, i.e., there are no restrictions on fill; the class CA of APL arrays consists of all empty arrays together with all nonempty arrays whose elements are all characters. Class AA arrays are all APL arrays. Using this classification, the following is a complete description of fill for the results of the primitive functions in present APL.

1. The scalar primitives other than = and ≠ apply to class

NA arrays and produce class NN arrays; the scalar primitives $=$ and $\neq$ apply to class AA arrays and produce class NN arrays.

2. The nonscalar primitives $\boxplus\perp\top\Delta\nabla$? and monadic ι apply to class NA arrays and produce class NN arrays; the nonscalar primitives ϵ, monadic ρ, and dyadic ι apply to class AA arrays and produce class NN arrays.
3. Monadic $\bar{\ast}$ applies to class AA arrays and produces class CC arrays; dyadic $\bar{\ast}$ applies to class NA left and right arguments and produces class CC arrays; $\ddot{\ast}$ applies to class CA arrays and produces class AA arrays.
4. The nonscalar dyadic primitives $\uparrow\downarrow/\backslash\phi\phi$ and dyadic ρ apply to class NA left arguments and class AA right arguments, and a result array has the same fill as that of the right argument; the nonscalar monadic primitives $\phi\phi$, apply to class AA arrays and a result array has the same fill as that of the argument.
5. If both arguments of catenate (,) are nonempty, then they are both class NA or they are both class CA, and the result has their common fill; if one argument is empty, then both arguments are of class AA and the fill of the result is the same as the fill of the nonempty argument; if both arguments are empty, they are of class AA and the fill of the result is the same as the fill of the right argument.
6. The fill of a result of indexing is the same as the fill of the array being indexed; the fill of an array altered by index-specification is the same as the fill of the array before alteration unless the entire array is replaced, in which case the fill is the same as the fill of the replacement.
7. The fill arrays for take and expand are defined as follows (see Section 3): Let

   $FS{:}\omega + (\sim\omega) \times \rho\alpha$

   Then the fill array for $/\backslash[K]A$ is $(A\ FS\ K \neq \iota\rho\rho A)\rho TA$, where TA is the fill of A, and the fill array for $N \uparrow A$ is $(A\ FS\ (N \neq \rho A)/\iota\rho\rho A)\rho TA$.
8. The fill for numeric constants is the scalar zero and the fill for character constants is the scalar blank.

In order to admit heterogeneous and general arrays, this eight-part description is simply accepted as the *definition* of fill for the results of the primitive functions, where the only required change is to the definition of catenate for two nonempty arguments. Thus (5) should now read as follows:

5′. The arguments of catenate are of class AA, and if both arguments are nonempty or if both are empty, then the fill of the result is the same as the fill of the right argument, while if one argument is empty, then the fill of the result is the same as the fill of the nonempty argument.

In addition to this change, general arrays require definitions for enclose ($\subset\omega$) and disclose ($\supset\omega$) analogous to those for the other primitives, which are taken to be the following.

9. Enclose ($\subset$) applies to arrays of class AA, and if TA is the fill of an argument, then $\subset TA$ is the fill of the result; $\supset$ applies to arrays of class AA, and if TA is the fill of an argument, then $\supset TA$ is the fill of the result.

Thus a permissive general array extension admits no new fill elements, while a strict one admits $\subset 0$, $\subset' '$, $\subset\subset 0$, $\subset\subset' '$, etc. Note that enclose and disclose are the only primitives that actually transform the fill: the others simply pass it along or replace it.

In the presence of heterogeneous arrays it is possible to create some interesting effects that may at first seem counterintuitive. For example, if

$A \leftarrow 2\ 3$

then the fill of A is the scalar zero (8). According to (6), the result of

$A[1] \leftarrow 'A'$

$A[2] \leftarrow 'B'$

still has fill zero even though A is $'AB'$, so that in particular

$1\ 0\ 1/\backslash A \leftrightarrow 'A', 0, 'B'$

Once fill becomes a truly independent characteristic of APL arrays, one must expect to encounter pairs of sequences of primitives that yield arrays with the same shape and element list but with different fill.

$\subset[K]\omega$ can be defined for empty arguments by the procedure in "Empty results." $\supset[K]\omega$ is not a conventional application of the axis operator, but is simply $\supset\omega$ followed by a dyadic transpose, so its definition for empty arguments is straightforward. As with other fill element definitions (see "Permissive general arrays"), identity (4a) holds but identity (4b) fails in general for empty arrays. Of course this brings us to the topic of consistent behavior for empty collections, and we have seen that any fill element scheme will be deficient in this area. What one should be able to do here is specify fill arrays that replace the default fill elements. Iverson [10, Section H] has defined variant forms of take and expand that provide this capability, and that may well be sufficient if these forms are extended to permit fill arrays as well as fill elements, and if an alternative to the evaluation procedure in "Empty results" is provided. Another approach is suggested here, which is more in keeping with the developments in this paper, and which we believe will be more general and more convenient to use.

Definition 19

A new primitive function called *FILL* is defined which is analogous to ρ and for which *FILL A* is the fill of A, while

F FILL A is the array whose shape and element list are the same as those of *A* but whose fill is *F*. Thus for all arrays *A*,

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

Given such a primitive, and assuming that specified fill is propagated by the same rules as default fill, consistent behavior of empty collections of units can be obtained. If *TU* is a typical unit of an application and the units lie along certain axes within their collections, thon all that must be done is to specify *TU* as the fill of a collection when processing begins; if the units are enclosed scalars, then $\subset TU$ should be specified as fill. When an individual unit is removed from a collection for processing, then respecify the fill if necessary.

The properties of specifiable fill are examined by briefly surveying the individual aspects of empty arrays discussed in the preceding sections. Fill plays a minimal role in identity-inducing functions because these functions tend to be dependent only on shapes and the function argument of reduction. However, it is probably not difficult to construct examples where fill arrays, and in particular specifiable fill, would be of use. On the other hand, specifiable fill eliminates the need for fill-transforming functions associated with defined functions because specifiable fill need not only consist of zeros and blanks. For example, it was pointed out in "Numerical applications" that $(\rho\alpha)\rho \subset 0\ 1$ is an effective fill-transforming function for rational arithmetic functions, but the same effect can be obtained by assigning $\subset 0\ 1$ as the fill for arrays of rational numbers.

The type arrays of Sections 3 and 4 maintain the same ranks as the arrays to which they are associated in order that transformations on the associated arrays can be mirrored on the types themselves. This manipulation of types is required so as to maintain constant relations between the axes of type arrays and the axes of the arrays to which they are associated. These constant relations guarantee to the greatest extent possible that any logical decomposition of arrays into units and collections of units is mirrored in their types, and therefore when a computation applied to an empty collection requires fill, that the fill will have the form of a "typical" logical unit.

In contrast, specifiable fill requires typical units to be provided explicitly, but in so doing permits a greater variety of possibilities, if only because it need not consist only of zeros and blanks. The above descriptions of the primitives are designed to pass along these typical units whenever it is reasonable to do so, and to use them in the gaps created by take and expand in the expected manner. In particular, identities (4a) and (4b) hold for empty arrays when the shapes of the empty arrays and the shapes of their specified fill are correctly related. Since fill is specifiable, it has no required relation to shape and element list, and therefore the primitives cannot be expected to apply to fill in the same way that they applied to type arrays. Consequently, when the form of the logical units of an application changes, the form of the fill must be explicitly changed as well.

The ability to specify the fill used by take and expand for nonempty arrays is obviously useful. Other applications to nonempty arrays arise from the fact that specifiable fill is an independent quantity and therefore represents additional information. For instance, it is sometimes difficult to know whether or not an array represents a logical unit or a collection of logical units. As an example, consider the alternate arithmetic application of replacing the real arithmetic carried out by the primitive functions with rational arithmetic carried out by defined functions. If the name *PLUS* everywhere replaces the symbol + (and similarly for other arithmetic primitives) in an algorithm, then the function *PLUS* must serve as an extension of + that does ordinary real addition when it should as well as rational number addition when it should. Suppose that fill is not specifiable and that rational numbers are represented by two-element vectors, while arrays of rational numbers are represented by general arrays of two-element vectors. Then the global replacement of + by *PLUS* can be guaranteed to have the desired behavior only for operator extensions such as Definitions 1–5 and only for algorithms that do not themselves use general arrays. For if the algorithm uses general arrays, then how is *PLUS* supposed to differentiate between general arrays of pairs of real numbers and general arrays representing arrays of rational numbers? And if the operators apply disclose before function arguments are applied, then how is *PLUS* supposed to determine whether an ordinary two-element vector holds two real numbers or one rational number? Both of these situations can be clarified by allowing specifiable fill, and by specifying the fill for any two-element vector representing a rational number to be a two-element vector that represents a valid rational number in the common domain of the arithmetic functions, and that does not coincide with any other specified fill in the algorithm.

Thus the following general conclusions can be made: Fill arrays based on types provide the expected behavior for empty collections of logical units, but do not generally provide useful fill in other circumstances; just the opposite is true of simple fill elements based on type; and specifiable fill permits the user to control the behavior of empty collections while providing a default fill that is generally useful in other circumstances. In addition, specifiable fill can replace fill-transforming functions and is of use in dealing with nonempty arrays.

References and note

1. The original version of this paper was produced several years ago as an informal working document. At the time no production APL implementation had any of the extended language features discussed here. Thus the reference throughout the paper to "present APL" was clear: it meant a language level on the order of [2]. What that phrase means today is not so

clear, but it has been left alone, and the reader now knows to what it refers.
2. *APL Language,* Order No. GC26-3847; available through IBM branch offices.
3. Trenchard More, "Axioms and Theorems for a Theory of Arrays," *IBM J. Res. Develop.* **17,** No. 2, 135-175 (1973).
4. T. More, "Types and Prototypes in a Theory of Arrays," *Technical Report No. G320-2112,* IBM Cambridge Scientific Center, Cambridge, MA, May 1976.
5. W. E. Gull and M. A. Jenkins, "Recursive Data Structures in APL," *Commun. ACM* **22,** No. 2, 79–96 (1979).
6. K. F. Ruehr, "A Survey of Extensions to APL," (APL '82 Conference Proceedings), *APL Quote Quad* **13,** No. 1, 277-314 (1982).
7. W. E. Gull and M. A. Jenkins, "Decisions for 'Type' in APL," *Sixth ACM POPL Symposium Proceedings,* 1979, pp. 190–196.
8. D. L. Orth, "A Comparison of the IPSA and STSC Implementations of Operators and General Arrays Extensions of APL," (APL '81 Conference Proceedings), *APL Quote Quad* **12,** No. 2, 11–21 (1981).
9. Ziad Ghandour and Jorge Mezei, "Generalized Arrays, Operators and Functions," *IBM J. Res. Develop.* **17,** No. 4, 335–352 (1973).
10. K. E. Iverson, "Rationalized APL," *Research Report No. 1,* I. P. Sharp Associates, Ontario, Canada, January 1983.
11. M. A. Jenkins and J. Michel, "ALICE: An Extensible Language Based on APL Concepts," *Technical Report No. 80–104,* Queens University at Kingston, Ontario, 1980.
12. A. D. Falkoff and D. L. Orth, "Development of an APL Standard," (APL '79 Conference Proceedings), *APL Quote Quad* **9,** No. 4, Part 2, 409–453 (June 1979).
13. J. A. Brown and M. A. Jenkins, "The APL Identity Crisis," (APL '81 Conference Proceedings), *APL Quote Quad* **12,** No. 1, 62–66 (September 1981).
14. D. A. Rabenhorst, "APL Function Variants and System Labels," (APL '83 Conference Proceedings), *APL Quote Quad* **13,** No. 3, 281–284 (1983).

Received November 16, 1983; revised March 2, 1984

Donald L. Orth *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Orth is a Research staff member in the Computer Sciences Department. He is currently manager of the APL compiler project. He joined IBM in 1974 at the IBM Scientific Center in Philadelphia, Pennsylvania. He received his B.S. in 1961 and his M.S. in 1963, both in mathematics, from the University of Notre Dame, Indiana. He received his Ph.D. in mathematics in 1967 from the University of California at San Diego. From 1967 to 1969, he was a National Science Foundation Fellow at Princeton University, New Jersey. Dr. Orth is the author of *Calculus in a New Key.*