APL 80, G.A. van der Linden (ed.)
North-Holland Publishing Company
© SIC, 1980

EXTENDING APL TO INFINITY

Eugene E. McDonnell and Jeffrey O. Shallit

I.P. Sharp Associates
220 California Avenue
Palo Alto, CA 94306, USA

Department of Mathematics
University of California
Berkeley, CA 94720, USA

A recent proposal suggested that a positive and negative infinity be added to APL. This paper discusses the effect on APL of adding infinities to APL. There are two principal topics: infinities as elements and arrays having infinitely long axes. The properties of infinite elements and infinitely large arrays are exhibited by describing the behavior of each primitive function with respect to them.

1. INTRODUCTION

In a recent paper [Ivl] Iverson proposed that the symbols _ and ¯ be used to denote infinity and negative infinity, respectively. He shows the use of these symbols to denote the limit of a function or its inverse, to separate ordinary numeric arguments of the axis operator, and to denote an alternative fill character for the expand function. In this paper, we explore further the consequences of admitting symbols for infinities. There are two main issues: infinite elements and infinite axes. We discuss these in terms of their effect on the behavior of each of the primitive functions.

1.1 Why have an infinity?

Having a representation for infinity would be advantageous in several ways. For example, it would provide a result for those cases involving division by zero, directly or indirectly. This would, among other things, simplify the definition of the residue function. Currently, in order to provide a definition which is valid for all numeric arguments, the residue function must be defined by *RES*:ω-α×⌊ω÷α+α=0. The phrase at the end ("+α=0") is included in order to avoid the domain error that would result in evaluating 0|*A*. Providing the result ¯ or _ for *A*÷0 would permit the definition for residue to be written more simply as *RES*:ω-α×⌊ω÷α.

1.2 Undefined versus indeterminate; pole

An undefined expression is one for which no numerical result is available. In elementary school we learn that 3÷0 is undefined. Later we learn that ⍟0 is also undefined. If we study analysis we learn that an argument which would cause the result of a function to be infinitely large is said to be a pole of the function; this is a more sophisticated way of dealing with the notion of "undefined".

On the other hand, an indeterminate is an expression for which the rules of mathematics are ambiguous or do not give consistent guidance. Thus, in dealing with the case 0÷0, one could argue that it should have the value 1, since *A*÷*A* has the value 1 for non-zero *A*, and as *A* approaches zero from any direction, the expression maintains the value 1. But one could also argue that 0÷0 should have the value 0, since 0÷*A* has the value zero, and as *A* approaches zero from any direction the expression maintains the value 0. Lastly, from

A=*B*×*C* ↔ *B*=*A*÷*C*

that, since 0=*B*×0, any number *B* could be used as the value of 0÷0. The point is, not what choice is made, but that a choice can be made and defended.

To sum up: undefined means "no choice", whereas "indeterminate" means "many choices".

2. ORTHOGRAPHY

The symbols chosen by Iverson for infinities, the underbar _ and overbar ¯, are different from the "lazy 8" symbol used in conventional mathematics. This is undoubtedly partly because of the present limitations of the APL character set, but there is also both some justice to the choice, and some opportune additional benefits resulting from it. In a tabulation of results, a dash is frequently used to indicate "not available". With the orthography proposed, we would have, for example:

```
      3 2 1÷2 1 0
1.5 2 _
```

where the underline signifies, in a sense, an "unavailable" result.

Since the proposed characters are numeric in type, they would be added to the list of characters available for spelling names in APL, and would follow the rules given for the use of the numeric characters -- they could appear within a name, but could not be used to begin a name. Thus, names of the form *RATE_OF_PAY* could be formed which would have greater clarity than, for example, either *RATEOFPAY* or *RATE∆OF∆PAY*. There is even some hope that _ and ¯ could replace the characters $\overline{\Delta}$ and $\underline{\Delta}$ as alphabetic extenders, thereby making ∆ available as a function symbol.

2.1 Complex complications.

There are orthographic complications in describing infinity in the complex plane. In his summary paper [Pel] Penfield gives two proposals for the representation of a complex constant. In one, the real and imaginary parts are separated by the _ character, for example 3_4. Penfield prefers the use of the dieresis character ¨ , rather than the underbar, since using the underbar as separator would require writing the complex constant having real and imaginary parts equal to infinity as ___. On many printing or display devices the three underbars are not distinguished, but run together to form one line.

However, we propose that any complex infinity (one not on the real line) be represented in polar form. One such polar form might be m*P*a, where m is the magnitude of the number and a is the angle in degrees. Thus an infinity at an angle of 30 degrees would be represented by *_P*30. We propose that in a display of complex values any non-infinite value be represented in rectangular form (or as a real number), but that a complex infinity be represented in polar form.

3. REPRESENTATION IN MACHINE FORM

This discussion is specific to the machine architecture of the IBM System/360 and equivalent machines, such as those made by Amdahl Corporation. Representations for other machine architectures would have to be devised to accord with the possibilities of those machines.

In System/360, a long floating-point value has 64 bits. The first of these is the sign bit for the fraction, the next 7 are the (excess 64) exponent, and the remaining 56 are used, four at a time, for 14 hexadecimal fraction digits. A non-zero value is kept in a normalized form, in which the leading fraction digit is non-zero. This is the form in which APL implementations written for System/360 keep numbers with fractional parts as well as integers larger than can be accommodated in the 32-bit fixed-point form. System/360 also has a short floating-point form, not used by APL implementations, which uses only 32 bits, and gives only 6 fraction digits. One could use a long unnormalized form, in which the first six fraction digits were zero, and the last eight encoded to have various meanings, to represent the infinities we are discussing. Thus, if such a floating point number were to appear as an argument, a simple test could determine that an infinity is present:

```
      LTER  X,X    TEST UPPER 6 DIGITS
      BZ    MAYBE  BRANCH IF ZERO
      ...
      ...
MAYBE LTDR  X,X    TEST ALL 14 DIGITS
      BNZ   YES    BRANCH IF NON-ZERO
      ...
      ...
YES   (code for infinite argument)
```

The sign bit could be used to distinguish infinity from negative infinity. The low-order 32 bits provide enough encodings to represent not only the infinities we are discussing, but also other special values, such as indeterminates and infinitesimals. One could even distinguish "true" from "machine" infinities. These matters will not be further discussed in this paper.

4. INFINITE ELEMENTS AS RESULTS; AS ARGUMENTS

In this section we show which functions can produce arrays having infinite elements, and then discuss for each function the implications of allowing its argument(s) to have infinite elements.

Only certain of the primitive functions can give results which are infinite. Before discussing these, however, we have to distinguish two cases. In the first case, the argument is a pole of the function, and the result is properly infinite. For example, 0 is a pole for the reciprocal function, and so ÷0 is properly infinite. In the second case, the argument is not a pole of the function, but the result is not representable in the machine environment. For example, !1000 is a perfectly ordinary, finite number, but since it has 2568 decimal digits it can't be represented using System/360 architecture. Falkoff and Iverson point out [Fal]

> Problems such as overflow (i.e., a result outside the range of the representations available) were treated as domain errors, the term domain being understood as the domain of the machine function provided, rather than as the domain of the abstract mathematical function on which it was based.

We propose that this latter class of result be called machine infinity, and that it also be represented by the underbar symbol. We shall distinguish true infinity from machine infinity in the tabulations of the scalar functions given below by "T" and "M", for "true" and "machine".

4.1 Infinities as results of monadic scalar functions

Table 1 shows the monadic scalar functions which can produce an infinity as a result, indicates whether this is a true or machine infinity, whether this is a positive or negative infinity, or both, and, for a machine infinity, gives the range of values of the argument which produce the machine infinity, using the architecture of System/360, and the implementation of APL given by I.P. Sharp Associates.

÷	*T*: *A*=0, _; *M*: (|*A*)<*A*1, ¯ _
*	*M*: *A*>*A*2, _¯
⍟	*T*: *A*=0,
!	*T*: (*A*<0)∧*A*=⌊*A*, ¯ _; *M*: *A*>*A*3, _
○	*M*: |*A*)>*A*4
3○	*T* and *M*: (*A*>*A*5)∧*A*<*A*6, ¯ _
5○	*M*: (|*A*)>*A*7, ¯ _
6○	*M*: (|*A*)>*A*7, _
¯5○	*M*: (|*A*)>*A*8, ¯ _
¯6○	*M*: *A*>*A*8, _
?	*M*: *A*>*A*9, _

legend
A=the argument
*A*1=1.381786988151113*E*¯76
*A*2=174.673080444335935
*A*3=56.5452012056031
*A*4=2.3036104216320906*E*75
*A*5=1.5707963267948951
*A*6=1.570796326794899
and in general there are arguments like *A*5 and *A*6 which give bounds around ○*N*+0.5 for any integer *N*
*A*7=175.36622619628905
*A*8=8.50705917302346*E*37
*A*9=3.369993336532381*E*66; however, this is a function of the value of ⎕*RL* as well; the smaller the value of ⎕*RL*, the larger *A*9 can be. The value shown is the smallest value which will give a machine infinity and corresponds to a value for ⎕*RL* of ¯2+2*31.

Table 1

The functions which give a true infinity are reciprocal, logarithm, factorial and tangent. Except for logarithm, which gives negative infinity at zero, the others are ambiguous in that either infinity or negative infinity could be given as the value at each pole. There does not appear to be a simple rule which could be uniformly applied to decide which infinity should be given. For the tangent function, the ambiguity is resolved because we cannot represent pi over two or the other members of its residue class, modulo two pi, exactly, and thus any infinite result for the tangent function will be a machine infinity. The argument will always be just below or just above the pole, and therefore we are able to give either infinity or negative infinity, respectively, as the result.

With reciprocal, for values of the argument which are sufficiently close to zero, we also can determine, using the sign of the argument, whether the result should be infinity or negative infinity. On the other hand, for the argument zero itself, there is no a priori reason for preferring either of the choices. The rule we propose is that the infinity chosen should have the same sign as the result of applying the function to any number for which the pole is the floor. For example.

```
      ×÷0.5
1
      ⌊.5
0
```

and therefore,

```
      ÷0
_
      ×_
1
```

The same rule can be applied to the poles of the factorial function (the negative integers). Thus,

```
      ×!¯2.3
1
      ⌊¯2.3
¯3
      !¯3
_
      ×_
1
```

but on the other hand

```
      ×!¯3.2
¯1
      ⌊¯3.2
¯4
      !¯4
¯
      ×¯
¯1
```

4.2 Infinities as arguments of monadic scalar functions

Once we admit the possibility of an infinity as the result of an expression, the question arises, what will be the result if this infinity is used in turn as the argument to a function. Table 3 shows, for each monadic scalar function, the result when it is applied to infinity or negative infinity. As will be seen, it seems useful to say that infinities, like zero, take on all the characterizations of real numbers. They are identically integers and non-integers, divisible by 2, multiples of two pi, and so forth.

Reciprocation The value shown for ÷_ and ÷¯ is 0. Knuth [Kn1] makes the point that if infinity is used to represent what we call machine infinity, as well as true infinity, it is incorrect to give 0 as the result of ÷_, "lest inaccurate results be regarded as true answers!" He suggests that an indeterminate value be reserved for such results. In this paper we do not discuss the possibility of an indeterminate result, and give zero as the result of reciprocating infinity.

Floor and ceiling These functions show how the infinities can be characterized as integers, since we see no alternative to giving as results the value of the arguments.

Roll Statistically, any finite number has zero probability of being chosen out of an infinitude of numbers, and so we give _ as the result of ?_.

Logarithm The result of ⍟¯ is shown as being a domain error. In fact, if complex numbers are admitted, it will be possible to give the logarithm of a negative number. In this paper we do not discuss further the implications of complex numbers on the question of infinities.

Circular functions The circular functions 1 2 3○ are shown as giving zero as the result. Ball, in [Bal], says that these results should be undefined. We give zero as the result for each, since we claim that the infinities are multiples of two pi (for sine and tangent) and also pi over two and the members of its residue class, modulo two pi, for the cosine. In indeterminate cases like this, there is also some justification in claiming that zero is an unbiased choice, using the arguments given in [Mc1].

Factorial We show ¯ as the result of !¯. Here we consider negative infinity not only to be an integer, but an even integer, since, as we saw in the discussion of Table 1 the factorial of negative even integers is negative infinity.

4.3 Infinities as results of dyadic scalar functions

Table 2 shows the dyadic scalar functions which can produce infinity or negative infinity as a result. The only functions which give true infinities are those derived from the corresponding monadic scalar functions, namely divide and dyadic logarithm (from reciprocal and logarithm).

+	*M*
	M
×	*M*
÷	*T*: *A*10÷0 ; *M*
*	*M*
⍟	*T*: *A*10⍟0
!	*M*
legend *A*10≠0	

Table 2

4.4 Infinities as arguments of dyadic scalar functions

Table 4 shows the result of using each dyadic scalar function with negative infinity as one argument, and several kinds of other argument. For the commutative functions, such as addition, only one row is given, which may be read either with negative infinity as the left argument, and each element at the head of the columns as the right argument, or vice-versa. For the non-commutative functions, such as subtraction, there are two entries, the first showing negative infinity as left argument, with the elements at the head of the columns as right argument, and the second with negative infinity as right argument, and the elements at the head of the columns as left argument. Table 5 provides the same information for infinity as left and right argument.

For many scalar dyadic functions we show domain error when both arguments are infinite. Although some authorities [Bal Bul Knl] give results in some of these cases, we feel that there is enough disagreement among competent judges that at present we ought to beg the question. It is easier to remove a domain error than to put one in.

F	¯	_
+	¯	_
-	_	¯
×	¯1	1
÷	0	0
|	_	_
⌊	¯	_
⌈	¯	_
?	*D*	_
*	0	_
⍟	*D*	_
○	¯	
¯7○	*D*	*D̄*
¯6○	_	_
¯5○	¯	_
¯4○		
¯3○	-*V̄*	*V̄*
¯2○	*D*	*D*
¯1○	*D*	*D*
0○	*D*	*D*
1○	0	0
2○	0	0
3○	0	0
4○	_	_
5○	¯	_
6○	_	_
7○	¯1	1
!	¯	_

Table 3

LF	¯	*N*	0	*P*	*S*	1	*G*	_	*FR*
¯+	¯	¯	¯	¯				*D*	+¯
¯-	*D*	¯	¯	¯				¯	
	D	_	_	_				_	-¯
¯×		¯	0̄	¯				¯	×¯
¯÷	*D̄*							*D*	
	D	0̄	0	0				*D*	÷¯
¯|	*D*	*N*	0	*P*				*D*	
	0	0	¯	0				*D*	|¯
¯⌊	¯	¯	¯	¯				¯	⌊¯
¯⌈	¯	*N*	0	*P*					⌈¯
¯*				*D*				_	
	D	*D*	_		_	1	0	0	*¯
¯⍟				*D*					
				D					⍟¯
¯○				*D*					
									○¯
¯!	1	0	0	0				0	
	1	0	0	0					!¯
¯<	0	1	1	1				1̄	
	0	0	0	0				0	<¯
¯≤	1	1	1	1				1	
	1	0	0	0				0	≤¯
¯=	1	0	0	0				0	=¯
¯>	0	0	0	0				0	
	0	1	1	1				1	>¯
¯≥	1	0	0	0				0	
	1	1	1	1				1	≥¯
¯≠	0	1	1	1				1	≠¯

Table 4

LF	¯	*N*	0	*P*	*S*	1	*G*	_	*FR*
_+	*D*	_	_	_					+_
_-								*D̄*	
	▬	▬	▬	▬				*D*	-_
_×	¯	¯	0	_					×_
_÷	*D*	¯		_				*D̄*	
	D	0	0̄	0̄				*D*	÷_
_|	*D*	*N*	0	*P*				*D*	
	0	0		0				0	|_
_⌊	¯	*N*	0̄	*P*				_	⌊_
_⌈	_	_	_	_				_	⌈_
_*	0̄	0̄	1̄	_				_	
	D	*D*	0	_	0	1	_	_	*_
_⍟	*D*	*D*	¯1	0	¯			1̄	
	D	*D*	¯1			_	_	1	⍟_
_○				*D*					
									○_
_!	_	0	0	0				1	
	0̄	0	0	0				1	!_
_<	0	0	0	0				0	
	1	1	1	1				0	<_
_≤	0	0	0	0				1	
	1	1	1	1				1	≤_
_=	0	0	0	0				1	=_
_>	1	1	1	1				0	
	0	0	0	0				0	>_
_≥	1	1	1	1				1	
	0	0	0	0				1	≥_
_≠	1	1	1	1				0	≠_

Table 5

legend	
D	domain error
F	function
G	1<*G*, *G*<_
L	left argument
N	negative number
P	positive number
R	right argument
S	0<*S*, *S*<1
V	○0.5

4.5 Infinities as results of mixed functions

The two mixed functions that may give results having infinite elements are decode (⊥) and the two forms of ⌹: matrix divide and matrix inverse. For decode, the infinities are machine infinities; this follows from the definition of decode in terms of addition and multiplication. Matrix inverse may result in a machine infinity if the matrix is close to singular; however it will also give a true infinity if its argument is a singular matrix, one having a zero determinant. To see why this is so, recall that the inverse of a square matrix is given by

```
      ⌹A ←→ (⍉ADJOINT A)÷DET A
```

and if the determinant of A is zero, then for each non-zero element of the transposed adjoint, the corresponding element of the inverse will be infinite.

An application where this would be useful is as follows: currently, it is difficult to determine (under program control) whether a given matrix is singular without error trapping facilities. If we permit division of non-zero by zero to give an infinite result, a test for singularity is given by _∊|⌹A. Similar results exist for dyadic ⌹.

4.6 Infinities as arguments of mixed functions

If A and B are arrays having one or more infinite elements, then expressions of the form

```
 ⍴A   ⌽A   ⍉A  K↑A   ,A K/A A⍳B   ⍋A   ⍒A A[K]
K⍴A  K⌽A  K⍉A  K↓A  A  B K\A A∊B   ⍋A  K⍒A
```

have obvious definitions in terms of current APL.

The expressions

```
A⍴B    A↑B    A↓B    ⍳A
```

can produce infinite arrays as discussed in the next section. For example, ⍳_ is an array such that (⍳_)[K] ←→ K.

We see no way by which the deal function ? can be meaningfully defined for infinite arguments.

The functions ⊥, ⊤, and ⌹ are defined in terms of more primitive functions, and we may retain these definitions with respect to infinite arguments. For example, 10⊥_ 2 3 ←→ _.

5. INFINITE ARRAYS

Expressions such as

```
      ⍳_
```

and

```
      _ 2⍴X
```

suggest a generalization of APL to infinite arrays

We say that A is an infinite array if _∊⍴A. The concept of infinite arrays adds significant new capabilities to APL.

Consider the problem of evaluating the series for the constant e. This series is infinitely long, and practically speaking, one uses only a finite prefix of it. Suppose we wish to evaluate it until the n-th term is smaller than 1E¯8. If we know that we don't have to sum more than 25 terms, we can write

```
      +/÷!⍳(1E¯8>÷!0,⍳25)⍳1.
```

If, on the other hand, 1E¯8 is replaced by an arbitrarily small positive number EPS, an accurate a priori bound may be difficult to compute. In this case, we might replace 25 by 100 or some larger number. But _ is greater than any number; hence we should be able to write

```
      +/÷!⍳(EPS>÷!0,⍳_)⍳1,
```

which gives a solution using the infinite vector ⍳_.

5.1 Implementing infinite arrays

Typically, an APL implementation stores an array with a header containing the number of the array's axes and the length of each axis. For an infinite array, the length of an infinite axis can be given as a negative integer, for example ¯1. Furthermore, an infinite array may be stored as a function of its indices. For example, to store the infinite vector V←2+3×⍳_ we need only store the function VF:2+3×⍵. Then it is easy to see that V[K] ←→ VF K. Any particular element of V may be obtained by using the associated function VF. Since the user can never examine all the elements of V, it does not matter that the entire infinite array is not in fact stored: any portion of it may be computed as needed. A request such as

```
      ⎕←V
```

may be interpreted as continued evaluation and display of the results of *VF* until the user interrupts the display. Thus, to the user, it appears as if an infinite array is present. This is a generalization of the J-vector introduced by Abrams [Abl] and implemented in several versions of APL.

If *V* is an infinite vector and *A* is a scalar, then *V*ι*A* should return the location of the first occurrence of *A* in *V*; if *A* is not in *V* the search for it will go on forever. Hence the introduction of infinite arrays leads to simple expressions whose execution never terminates.

Note that expressing an infinite array as a function of its coordinates implies that the number of axes of the array is finite; otherwise it is conceivable that we could never compute the value of a given array position. In particular, all infinite arrays have a countable number of elements, meaning that we can pair each element of *V* with a non-negative integer in a unique fashion.

Before discussing in detail which mixed functions may be implemented on infinite arrays, it will be useful to make a few informal definitions concerning a monadic function f.

f is totally locally computable (TLC) if there is a well-defined, terminating procedure to compute (f *A*)[*K*] for any *K*, which requires that the procedure need only look at a finite subset of *A*, and that the procedure need not have access to any global information about *A*;

f is partially locally computable (PLC) if such a procedure exists for certain *A*;

f is locally uncomputable (LUC) if such a procedure never exists.

For example, f:2*α is TLC since f *V*)[*K*] ↔ 2**V*[*K*].

The function g:αι1 is PLC, since if 1 is in *V* the computation *V*ι1 will terminate, while if 1 is not in *V* it won't.

Finally, +/*V* is LUC since we can never compute the sum of an infinite number of elements by looking only at a finite subset. To compute +/*V*, "global" information about *V* is needed.

Those functions that are TLC may be conveniently implemented. Those functions which are PLC present more formidable difficulties.

We can now discuss the implementation of primitive functions and operators with respect to infinite arrays.

5.2 Scalar functions of infinite arrays

The scalar functions are easily implemented using the following identities. If *A* and *B* are infinite vectors then

```
    (f A)[K] ↔ f A[K]
  (A f B)[K] ↔ A[K] f B[K].
```

5.3 Mixed functions of infinite arrays

For most of the mixed functions it suffices to examine their behavior for infinite vectors. Extension to higher-order arrays is based on their action on infinite vectors.

Functions along finite axes of an infinite array are easily implemented. For example, if

```
A←_ 2ρι_
```

then

```
(+/A)[K] ↔ +/A[K;] ↔ (4×K)-1.
```

Since every infinite array is represented as a function of its indices, it suffices for implementation purposes to exhibit such a function. For example, the identity (ι_)[*K*] ↔ *K* indicates how ι_ may be implemented.

In the discussion that follows, *A* and *B* are infinite arrays, *V* and *W* are infinite vectors, *K* is a finite scalar, *J* is either a finite scalar or _ or ¯, and *C* is a finite vector.

shape: If *A* is an infinite array, then _ερ*A*.

reshape: Here we must have ~_ε1↓ρ*A*. To see this, consider the following: an expression like 3 _ρι_ says to fill an array in row-major order with three rows and an infinite number of columns with the numbers from one to infinity. This cannot be done, since it takes an infinite amount of numbers to fill the first row. Thus we have the unpleasant result that there are certain arrays that cannot be created with reshape alone. In particular, the identity

```
A ↔ (ρA)ρA
```

no longer holds in general, since the expression on the right may not even be defined. As another example, let

A←(⍳3)∘.×⍳_. Then _ρA effectively takes only the first row of A.

Reshape may be implemented using the fact that

```
(_ρC)[K] ←→ C[1+(ρC)|K-1]
```

ravel: This may be implemented using the identity ,A ←→ (×/ρA)ρA. Once again we obtain results that may look peculiar for infinite arrays of rank two or greater, that is

```
,(⍳3)∘.×⍳_ ←→ ⍳_
```

reverse: ⌽V is not defined, since the "last" element of V is not defined.

rotate: For non-negative, finite K, K⌽V is K↓V. K⌽V is not defined if K is negative or infinite.

catenation:

```
V,W ←→ V
V,C ←→ V
(C,V)[K] ←→ C[K] if K∊⍳ρC
            V[K-ρC] otherwise
```

transpose (monadic and dyadic): Transposition of infinite arrays (represented as functions of their indices) is facilitated by the use of Iverson's "from" function [Ivl, p.17]:

```
I⌷A ←→ (,A)[1+⍉(ρA)⊥⍉I-1]
```

(this definition in terms of ⊥ works correctly for ρA finite; it must be modified for infinite arrays, but the extension is clear). Then

```
K⌷J⍉A ←→ K[J]⌷A.
```

take:

```
_↑V ←→ V
(_↑C)[K] ←→ C[K] if K≤ρC
            fill element otherwise
¯↑V is not defined.
```

drop:

```
(K↓V)[J] ←→ V[K+J]     (K≥0)
K↓V ←→ V     (K<0)
```

_↓V and ¯↓V are not defined.

compress:

```
(B/V)[K] ←→ V[(+\B)⍳K]
```

expand:

```
(B\V)[K] ←→ 0 if B[K]=0
            V[+/B[⍳K]] if B[K]=1
```

indexing: (V[W])[K] ←→ V[W[K]]

index generator: (⍳_)[K] ←→ K

index of:

V⍳C: This function is PLC. It may be implemented as a parallel look up for each element of C.

C⍳V: We have (C⍳V)[K] ←→ C⍳V[K]

V⍳W: This function is also PLC, and may be implemented as shown for V⍳C.

membership:

(V∊C)[K] ←→ V[K]∊C.

C∊V: This function is PLC and may be implemented in a fashion similar to that for V⍳C. Note that we have the unusual fact that if the computation for C∊V terminates, the result is just (ρC)ρ1.

V∊W: As with V⍳C, this is PLC.

upgrade, downgrade: These functions are not in general well-defined. For example, no meaningful result can be assigned to

```
⍋2*-⍳_
```

monadic and dyadic format: These are both TLC functions, but no easy algorithm appears possible.

decode, encode, matrix inverse and divide, and execute: are all LUC and hence we propose that they not be implemented.

5.4 Derived functions of infinite arrays

inner product, reduction: These generally create LUC functions, and hence cannot be easily implemented [Mol].

scan: (f\V)[K] ←→ f/V[⍳K]

outer product:

```
    I⌷A∘.f B ←→
(((ρρA)↑I)⌷A)f((-ρρB)↑I)⌷B
```

where ρI ←→ (ρρA)+ρρB

5.5 Display of infinite arrays

Since APL prints arrays in row-major order, we encounter problems when trying to display arrays A with _∊1↓ρA. For example, since it takes an infinite amount of time to print the first row of (ι4)∘.×ι_, we never get to see the other rows. However, Breed [Brl] has shown how APL display can be modified so that, instead of displaying a matrix by giving each row in its entirety before starting the next row, one could instead display as much of each row as would fit in the print width before going on with the remainder of each row, continuing in this fashion until the entire matrix has been displayed. To illustrate the effect, compare the way an APL system currently displays the matrix *A*←2 20ρι40, with print width set at 40:

```
 1  2  3  4  5  6  7  8  9 10 11 12 13
      14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33
      34 35 36 37 38 39 40
```

with the way it would be displayed using Breed's modification:

```
 1  2  3  4  5  6  7  8  9 10 11 12 13
21 22 23 24 25 26 27 28 29 30 31 32 33

      14 15 16 17 18 19 20
      34 35 36 37 38 39 40
```

This suggestion applies also to higher rank arrays having only one infinite axis, where the infinite axis is the last axis: successive infixes of all the planes, etc. are displayed to the extent that they can fill the given display width, before the remainder of the planes, etc. are displayed. It cannot be used for displaying an array having more than one infinite axis.

5.6 The functions DIAG and IDIAG

As we noted before, if *A* is an infinite array, then ,*A* may not include all the elements of *A*. However, a function can be defined which converts an infinite array to an infinite vector without "losing" information.

For example, suppose

```
      A←(ι_)∘.×ι_
      5 5↑A
 1  2  3  4  5
 2  4  6  8 10
 3  6  9 12 15
 4  8 12 16 20
 5 10 15 20 25
```

We can convert *A* to a vector by selecting elements from successive diagonals of the array:

```
      DIAG A
1 2 2 3 4 3 4 6 6 4 5 8 9 8 5 ...
```

DIAG for finite arrays may be defined as follows

```
      DIAG:(,ω)[⍋,+/IOTAρω]
      IOTA:1+ωτωρ(ι×/ω)-1
```

DIAG causes a diagonal transformation of its array right argument. The result is a vector.

DIAG allows the user to examine arrays that otherwise could not be printed. For example,

```
      DIAG (ι2)∘.+ι_
2 3 3 4 4 5 5 6 6 7 7 8 ...
```

DIAG may be used with finite arrays as well, of course:

```
      DIAG 3 5ρι15
1 2 6 3 7 11 4 8 12 5 9 13 10 14 15
```

The inverse to the function *DIAG* is the function *IDIAG*. For finite arrays we have the following definition:

```
      IDIAG:αρω[⍋⍋,+/IOTAα]
```

Note that *IDIAG* is dyadic; the left argument specifies the shape of the result. We have

```
      A ←→ (ρA)IDIAG DIAG A
```

This identity holds for all arrays *A*, while the related identity

```
      A ←→ (ρA)ρ,A
```

holds only for finite arrays.

Example:

```
      A←_ _IDIAGι_
      5 5↑A
  1  2  4  7 11
  3  5  8 12 17
  6  9 13 18 24
 10 14 19 25 32
 15 20 26 33 41
```

IDIAG (unlike reshape) allows the creation of infinite arrays of any shape.

ACKNOWLEDGMENTS

We should like to thank the following people, who read an early draft of this paper and gave us many valuable comments: Larry Breed of IBM, Paul Penfield of MIT, Steve Smoliar of General Research, and Arlene Azzarello, Bob Bernecky Leigh Clayton, Doug Forkes, Ken Iverson, Roland Pesch, Joey Tuttle, and Ed Wilts, of I.P. Sharp Associates.

REFERENCES

[Abl] Abrams, P.S., An APL Machine, SLAC Report No. 114, Stanford University, Stanford, CA, 1970

[Bal] Ball, J.A., Algorithms for RPN Calculators, Wiley-Interscience, New York, 1978

[Brl] Breed, L.M., personal communication

[Bul] Bucholz, W. (ed), Planning a Computer System: Project Stretch, McGraw-Hill, New York, 1962

[Fal] Falkoff, A.D. and K.E. Iverson, "The Evolution of APL", SIGPLAN Notices, 13, 8, ACM, New York, 1978

[Ivl] Iverson, K.E. "Operators and Functions", RC 7091, IBM Corp., Yorktown Heights, NY, 1978

[Knl] Knuth, D.E. Seminumerical Algorithms, Addison-Wesley, Reading, MA, 1969

[Mcl] McDonnell, E.E., "Zero divided by zero", APL76, ACM, New York, 1976

[Mol] More, T., Jr., "Axioms and theorems for a theory of arrays", IBM Journal of research and development, 17, 2, 1973

[Pel] Penfield, P., Jr., "Proposal for a complex APL", APL Quote Quad, 9, 4, 1979