A Comparison of Array Theory and a Mathematics of Arrays

Article · January 1991

DOI: 10.1007/978-1-4615-4002-1_14

M. A. Jenkins
Queen's University

Lenore Mullin
University at Albany, The State University of New York

unary operations f. V indicates how many axes at the end of A are to be partitioned. The operation is applied to the partitions and the results gathered. The constraint on f is that the shape of its result depends only on the shape of its argument, and hence the result on each partition will be of the same size.

```
OMEGA IS TRANSFORMER f
   OPERATION V A {
     mix1 EACH f ( V takeright axes A split1 A ) }
```

[Va,Vb] BOMEGA f [A,B] implements the version of "omega" in MOA for binary operations f. Va indicates how many axes at the end of A are to be partitioned, and Vb does the same for B. The operation is applied to pairs of partitions from A and B and the results gathered. The constraint on f is that the shape of its result depends only on the shape of its arguments, and hence the result on each partition will be of the same size.

```
BOMEGA IS TRANSFORMER f
   OPERATION Vab AB {
     Va Vb := Vab ;
     A B := AB ;
     mix1 ( (Va takeright axes A split1 A) EACHBOTH f
       (Vb takeright axes B split1 B)) }
```

X MOA_OUTER f Y computes the outer product using a construction involving reshapes and a generalized transpose. All the f combinations are computed using scalar extension.

```
MOA_OUTER IS TRANSFORMER f
   OPERATION X Y {
     Newx := shape Y link shape X reshape X;
     Newy := shape X link shape Y reshape Y;
     (valence Y rotate tell (valence X + valence Y) fuse Newx) f Newy }
```

X MOA_INNER [f,g] Y computes the inner product using a construction involving reshapes and a generalized transpose. All the g combinations are computed in an array with one extra axis on the end, which is removed by an f reduction.

```
MOA_INNER IS TRANSFORMER f g
   OPERATION X Y {
    IF last shape X = first shape Y THEN
```



```
     Newx := rest shape Y link shape X reshape X;
     Newy := front shape X link shape Y reshape Y;
     Vals := (valence Y - 1 rotate tell (valence X + valence Y - 1)
            fuse Newx) g Newy;
     EACH REDUCE f cols Vals
    ELSE
     ??invalid_inner_product
    ENDIF }
```

a natural definition for the MOA equivalent of cart is formed by mixing the results of the AT cart.

```
moa_cart IS OPERATION A {
   mix cart A }
```

```
We denote this in this simulation by X index Y.

index IS OPERATION X Y {
 IF valence X = 1 THEN
   IF atomic Y THEN
    [Y] psi X
   ELSEIF valence Y = 1 and and (Y EACHLEFT in iota tally X) THEN
    Y EACHLEFT pick X
   ELSE
     ??invalid_index
   ENDIF
 ELSE
   ??invalid_index
 ENDIF }


# A compress B uses a list of 0 and 1's to select items from the vector B.
For higher dimensional arrays, the selection is done over the first axis.
It uses blend1 to fix a minor problem in Nial's blend operation in version 4.1.

compress IS OPERATION A B {
  IF atomic B THEN
    B := list B;
  ENDIF;
  IF A allin [0,1] and (tally A = first moa_shape B) THEN
    0 blend1 EACH ( A match 1 sublist ) ( 0 split1 B )
  ELSE
    ??invalid_compress
  ENDIF}

# A expand B uses a list of 0 and 1's to expand items from the vector B.
For higher dimensional arrays, the expansion is done over the first axis.
It uses blend1 to fix a minor problem in Nial's blend operation in version 4.1.

expand IS OPERATION A B {
  IF atomic B THEN
    B := list B;
  ENDIF;
  IF A allin [0,1] and (sum A = first moa_shape B) THEN
   IF valence B = 1 THEN
     Res := tally A reshape 0;
     J := 0;
     FOR I WITH grid Res DO
```



```
      IF A@I = 1 THEN
        Res@I := B@J;
        J := J + 1;
      ENDIF;
    ENDFOR;
    Res
  ELSE
    0 blend1 EACH ( A expand ) ( 0 split1 B )
  ENDIF
 ELSE
  ??invalid_expand
 ENDIF}

# moa_reverse A reverses the items of a list. If A is higher dimensional
the reversal is done along the first axis.

moa_reverse IS OPERATION A {
   0 blend1 EACH reverse ( 0 split1 A ) }

# N moa_rotate A rotates along the first axis of A if N is a scalar. If N
is an array of values of shape rest moa_shape A, then the individual
rotation is done for each corresponding vector in 0 split A.

moa_rotate IS OPERATION N A {
  IF isinteger N THEN
   0 blend1 EACH ( N rotate ) ( 0 split1 A )
  ELSEIF moa_shape N = rest moa_shape A THEN
   0 blend1 (N EACHBOTH moa_rotate (0 split1 A))
  ELSE
   ??invalid_rotate
  ENDIF }

# A moa_transpose B does a generalized transpose using the MOA encoding
of the left argument.

moa_transpose IS OPERATION A B {
  IF tally A = valence B and (gradeup A choose A = axes B) THEN
   Aa := sortup cull A EACHLEFT findall A ;
   Aa fuse B
  ELSE
   ??invalid_transpose
  ENDIF )

# V OMEGA f A implements the version of "omega" in MOA for
```

```
  (1 hitch rest moa_shape B reshape A) catenate B
 ELSEIF atomic B THEN
  A catenate (1 hitch rest moa_shape A reshape B)
 ELSE
  ??invalid_catenate
 ENDIF }
```

iota A generates a list of integers if A is an integer; otherwise if A is a shape, it generates an array of shape A catenate tally A with rows corresponding to the indices of A.

```
iota IS OPERATION A {
  IF isinteger A THEN
   tell A
  ELSEIF is_moa_shape A THEN
   mix tell A
  ELSE
   ??invalid_iota
  ENDIF }
```

A moa_take B selects a subarray from B with a shape that depends on the absolute values of integers in A. If an item of A is negative, the selection is taken from the end of the axis rather than the beginning. If A is shorter than the valence of B then the missing axes are taken in their entirety. If A is an integer it is treated as if it were the corresponding 1-vector.

```
moa_take IS OPERATION A B {
  IF and EACH isinteger A and and (abs A <= (tally A take moa_shape B)) THEN
   I gets tell abs list A EACHLEFT +
       (A < 0 * (tally A take moa_shape B + list A));
   I OUTER link tell (tally A drop moa_shape B) choose B
  ELSE
   ??invalid_take
  ENDIF }
```

A moa_drop B selects a subarray from B with a shape that depends on shortening the axes of B by the absolute values of the items of A. If an item of A is negative the dropping is done from the end of the axis. If A is shorter than the valence of B then the missing axes are not shortened. If A is an integer, it is treated as if it were the corresponding 1-vector.

```
moa_drop IS OPERATION A B {
  IF and EACH isinteger A and and (abs A <= (tally A take moa_shape B)) THEN
```



```
   I gets tell (tally A take moa_shape B - abs list A) EACHLEFT +
       (A > 0 * list A);
   I OUTER link tell (tally A drop moa_shape B) choose B
  ELSE
   ??invalid_take
  ENDIF }
```

MOA_REDUCE A does the right to left reduction of a vector using the binary operation f. It is assumed that f is "reductive" and has an identity element. The code below only works for the AT operations that have such an identity. For a higher dimensional array the reduction is done over the partitions along the first axis.

```
MOA_REDUCE IS TRANSFORMER f
  OPERATION A {
   IF valence A = 1 THEN
    IF empty A THEN
     f A
    ELSE
     first A f MOA_REDUCE f rest A
    ENDIF
   ELSE
    mix EACH MOA_REDUCE f (0 split1 A)
   ENDIF }
```

MOA_SCAN A does the right to left scan of a vector using the binary operation f. It is assumed that f is "reductive" and has an identity element. The code below only works for the AT operations that have such an identity. For a higher dimensional array the scan is done over the partitions along the first axis.

```
MOA_SCAN IS TRANSFORMER f
  OPERATION A {
   IF valence A = 1 THEN
    IF empty A THEN
     Null
    ELSE
     MOA_SCAN f front A append MOA_REDUCE f A
    ENDIF
   ELSE
    0 blend EACH MOA_SCAN f (0 split1 A)
   ENDIF }
```

x[y] is defined in MOA for a vector x and y an index or a vector of indices.

```
# some simple MOA definitions

moa_shape IS list shape

dimensionality IS valence

rav IS list

moa_pi IS prod

total IS tally

iota IS EXTERNAL OPERATION

moa_grid IS iota moa_shape

is_moa_shape IS OPERATION A {
  valence A = 1 and (and EACH isinteger A) and (and (A >= 0)) }

# I psi A implements the indexing operation of MOA denoted by psi.
  There are 3 cases:
    if I is a vector of length equal to the valence of A :
        return the element at address I if I is a valid index.
    if I is a higher dimensional array whose rows are indices :
        return the array of elements at the indices.
    if I is a vector of length less than the valence of A :
        return the partition of A selected by I

psi IS OPERATION I A {
    IF valence I = 1 and (tally I = valence A) and
        (I in rows moa_grid A) THEN
      I pick A
    ELSEIF valence I > 1  and and (rows I EACHLEFT in rows moa_grid A) THEN
      rows I choose A
    ELSEIF valence I = 1 and (tally I < valence A) and
         (I in tell (tally I take moa_shape A)) THEN
      I pick (tally I raise A)
    ELSE
      ??invalid_psi
    ENDIF }

# I gamma S converts an index I for shape S into the corresponding
position in the list of items of an array of shape S.
```



```
gamma IS OPERATION I S {
    IF shape I = shape S and (valence I = 1) THEN
      IF empty I THEN
        0
      ELSEIF tally I = 1 THEN
        first I
      ELSE
        last I + (last S * (front I gamma front S))
      ENDIF
    ELSE
      ??invalid_gamma
    ENDIF }

# N inv_gamma S converts a position N in the list of items of
an array of shape S to the corresponding index.

inv_gamma IS OPERATION N S {
    IF isinteger N and (N >=0) and (N < prod S) THEN
      IF tally S = 1 THEN
        [N]
      ELSE
        (N div last S inv_gamma front S) append (N mod last S)
      ENDIF
    ELSE
      ??invalid_inv_gamma
    ENDIF }

# A catenate B implements the calentation operation in MOA. For scalars
or vectors, it behaves the same as "link" in AT. For arrays of the
same valence it permits the calenation if their shapes differ only
in the first axis extent, in which case the arrays are joined along the first axis.
Catenate is also extended to allow catenation of scalars to higher
dimensional arrays by replication.


catenate IS OPERATION A B {
    IF valence A <= 1 and ( valence B <= 1 ) THEN
      A link B
    ELSEIF valence A = valence B and
        ( rest moa_shape A = rest moa_shape B ) THEN
      sum EACH ( first moa_shape ) A B hitch rest moa_shape A
        reshape link A B
    ELSEIF atomic A THEN
```

enkins 80]
M.A. Jenkins, *On Combining the Data Structure Concepts of Lisp and AP*
Tech. Rep. 80-109, Dept. of Computing and Information Science, Queen'
University, Kingston (1980).

enkins 84]
M.A. Jenkins, *The Role of Equations in Nial*, Tech. Rept. 84-161, Queen'
University, July 1984.

enkins 85]
M.A. Jenkins and W.H. Jenkins, *The Q'Nial Reference Manual*, Nial System
Ltd., Kingston, Canada, 461 pages (1985).

enkins 88]
M.A. Jenkins, J.I. Glasgow, E. Blevis, E. Hache, D. Lawson, *The Nial A*
Toolkit, Proceedings of the Avignon 8th International Workshop on Expert Sy
tems and their Applications, June 1988.

ore 73]
T. More, *Axioms and Theorems for a Theory of Arrays*, IBM J. Res. Develop
ment 17, number 2, pp. 135-175 (1973).

ore 90]
T. More, *private communication*,(1990).

ullin 88]
L. M. Mullin, *A Mathematics of Arrays*, PhD Thesis, Computer and Information
Science, Syracuse University, CASE Centre T.R. 8814, Syracuse, 1988.

ullin 89]
L. Mullin, D. Iyengar, and A. Krishnamurthi, *The Dot Product: A New Definition*, CASE Centre T.R. 8901, Syracuse University, Syracuse, 1988.

ullin 90]
L. Mullin, *The Phi Function: a basis for FFD with arrays*, First International
Workshop on Arrays, Functional Languages and Parallelism, Montreal, 1990.

Appendix

```
# MOA definitions in Nial

# support operations for the definitions below

axes IS tell valence

sortup IS SORT <=

gradeup IS GRADE <=

# the definitions of split1 and blend1 below reflect the correct
versions of split and blend that have been implemented in version 5.

split1 IS OPERATION A B {
  IF valence B = 0 THEN
    single B
  ELSEIF diverse A and . A allin tell valence B THEN
    Axesup := tell valence B except A ;
    tally Axesup raise ( gage ( Axesup link A ) fuse B )
  ELSE
    ??invalid_split
  ENDIF }

# mix1 restricts mix to only work on arrays whose items are of the same shape.

mix1 IS OPERATION A {
  IF equal EACH shape A THEN
    mix A
  ELSE
    ??unequal_shapes_in_items
  ENDIF }

blend1 IS OPERATION A B {
  Bb := mix B ;
  IF A allin axes Bb THEN
    Axesup := tell valence Bb except list A ;
    C := gage GRADE <= link Axesup A ;
    C fuse Bb
  ELSE
    ??invalid_blend
  ENDIF }
```

stems have turned out to be completely consistent with each other.

OA and AT have taken different approaches in generalizing operations on lists to
ose on higher dimensional arrays as observed in section 4.6. In AT, the tendency is
define the operation on the list of items of the higher dimensional array. For exam-
e, if A is [3,4] *reshape count* 12

1	2	3	4
5	6	7	8
9	10	11	12

en *sum A* is 78, whereas $_+$ *red A* is

10	26	42

essence, AT decided not to bias any of the operations to prefer a particular axis,
hereas MOA extends most operations by assuming partitioning along one axis.
nce such partitions are easily expressed in AT, the extended meanings of MOA are
sily achieved in AT as we have seen above. As far as expressibility is concerned,
en, it appears to be a matter of taste as to which system is preferred.

is clear that AT is superior to MOA as a notation for expressing algorithms for
neral purpose problem solving. AT expressions can be directly executed in a Nial
ocessor, whereas MOA must be translated to some implemented array language.
oreover, AT with its complete treatment of symbolic scalar data and its ability to
ndle heterogeneous and nested arrays, is much better suited for many problems of
ta representation than MOA. In particular, AT has proven to be a rich environment
r doing applications in knowledge based systems [Jenkins 88].

owever, MOA is well suited to problems within its target domain. The constraint to
at homogeneous arrays guarantees efficient space storage and efficient access to the
ems of arrays. Thus, for those problems in which MOA provides a succinct algo-
thm, the latter can be translated into effective programs on both conventional and
rallel computers [Mullin 89]. The same effect could be obtained in AT by provid-
g mechanisms for constraining the types of arrays, by providing more APL-like
erations that utilize flat arrays [Jenkins 78], and by implementing AT expressions
Nial so that intermediate nested arrays are not explicitly constructed.

. Conclusion

he advent of readily available parallel computers has focussed attention on the need
r programming systems in which parallel algorithms can be easily expressed. Many
searchers are looking to n-dimensional arrays as an underlying data structure that
herently supports parallel decomposition of some classes of engineering and

scientific problems. The desire is to find an array system that fits well with a pure functional programming language and encourages parallel descriptions of algorithms for such problems. It is hoped that both AT and MOA will prove to be useful steps along the way to achieving this goal.

8. References

[Abrams 70]
P.S. Abrams, *An APL Machine*, PhD Thesis, Computer Science, Stanford University, STAN-CS-70-158, Palo Alto, California (1970).

[Bird 87]
R. S. Bird, *An Introduction to the Theory of Lists*, in Logic of Programming and Calculi of Discrete Design, ed. M. Broy, Springer-Verlag, Berlin (1987).

[Brown 81]
J.A. Brown, M.A. Jenkins, *The APL Identity Crisis*, APL81 Conference Proceedings, San Francisco, October 1981.

[Brown 84]
J.A. Brown, *The Principles of APL2*, Technical Report TR 03.247, IBM Santa Theresa Lab, San Jose, (1984).

[Franksen 88]
O.I. Franksen, *private communication*,(1989).

[Franksen 90]
O.I. Franksen, *DTH Notes 1- 3 on More's Proposal for Version VIII: revised operations "tell" and"size", and associated operations*, Electric Power Engineering, Technical University of Denmark, Lyngby, (1990).

[Gull 79]
W.E. Gull, M.A. Jenkins, *Recursive Data Structures in APL*, Comm. ACM. 22 Number 1, pp 79-96 (1979).

[ISO 82]
International Standard for the Programming Language APL, ISO TC97/SC5/WG6-N28, 1982.

[Jenkins 78]
M.A. Jenkins, J. Michel, *Operators in an APL Containing Nested Arrays*, APL Quote/Quad, Vol 10, No. 2, 8-20 (1978).

plied to a list with fewer than two items? What is its meaning on binary opera- that are not scalar ones? What should be assumed about the order in which the tions are applied? How should the operation be generalized to higher dimen- l arrays? Some of these issues are examined in depth in a theory of lists oped by Bird [Bird 87] and in a paper proposing an approach for APL2 [Brown

OA, as in APL, the solution to the first problem is to introduce identity scalars ch of the operations to which *red* can be applied. MOA defines reduction for ithmetic operations and for *max*, *min* and *mod*. For example, the identity ele- of + is 0, while that of × is 1. Thus, $_{+}red\Theta$ is 0. Initially, MOA addressed associative binary operations and later left associative ones [Mullin 90]. For r dimensional arrays the reduction is done along the first axis.

as taken a different approach. It has defined the reductive process directly into rimitive operations where it appropriate, and has provided transformers *UCE* and *LEFTREDUCE* that do not assume the existence of an identity for an tion. The approach in AT is to define the reductive process directly in the operations *sum*, *prod*, *max*, *min*, *and* and *or*. These operations can also be in infix notation and for convenience the binary operations *plus*, ×, etc., and the ols + and * are provided as renamings of the unary operation.

so builds the reduction process into the operation *link* which joins lists into one list, and into *cart*, which forms a generalized cartesian product of its items.

also defines the second order operation *scan* which produces the list of partial s that are formed in doing a reduction. AT does not have an equivalent primi- out it can easily be defined in Nial.

Inner and Outer Products

oncepts of inner product and outer product in array languages are generaliza- of similar operations in linear algebra. Given two column vectors $\bar{x}$ and $\bar{y}$ of n, then

$$\bar{x}^T \cdot \bar{y} = \sum_{i=0}^{n-1} x_i y_i$$

OA, this is denoted $\bar{x}_{+}\cdot_{\times}\bar{y}$. The inner product operation in MOA can be applied pair of arrays *A* and *B* such that the last dimension of *A* has the same length as st dimension of *B* and results in the array formed from all the inner products of from *A* and columns from *B*.

, the similar transformer is denoted by *A INNER* [+,*] *B*. For both systems the tions are parameterized because there are other combinations of operations for products that provide interesting results. For example, *A INNER* [*and*,*or*] *B* a boolean matrix product and *A INNER* [*and*,*match*] *B* gives a boolean pattern e a vector matches a row of a table.

The outer product corresponds to the linear algebra expression $\bar{x}\cdot\bar{y}^T$ which produces the matrix of all products of items of the column vectors $\bar{x}$ and $\bar{y}$. This involves only one operation. In MOA, the *outer product* operation can be applied to any binary scalar operation to form a new operation. For example, $\bullet_{\times}$ denotes the conventional linear algebra outer product. The shape of the result is the catenation of the shapes of the arguments.

AT has a more general form of outer product in that it can apply to any operation and can combine any number of arrays. It is defined by the equation

$$OUTER\ f\ A = EACH\ f\ cart\ A$$

where *cart* is the generalized cartesian product operation. For a scalar binary operation *f*, *A OUTER f B* has the same meaning as the corresponding notation in MOA.

There are direct relationships between *inner product* and *outer product*. In a universe of flat arrays such as MOA, the inner product can be expressed in terms of an outer product, a diagonalizing generalized transpose, and a reduction. This result has been independently discovered by Abrams [Abrams 70] and in a much earlier treatment of array-like concepts by the American mathematician Peirce [Franksen 88]. In AT notation the relationship is

$$A\ INNER\ [f,g]\ B = EACH\ f\ rows\ (C\ fuse\ (A\ OUTER\ g\ B))$$

where $C = (front\ axes\ A)\ link\ (valence\ A + rest\ axes\ B)\ append\ \ [last\ axes\ A,\ valence\ A + first\ axes\ B]$.

In a universe of nested arrays, the relationship can also be expressed by partitioning the arrays into lists, applying the outer product of the partitions and then doing the reduction on each item. In AT, assuming that *f* is one of the reductive scalar operations, and *g* is a binary scalar operation, then

$$A\ INNER\ [f,g]\ B = EACH\ f\ (rows\ A\ OUTER\ g\ (0\ split\ B))$$

6. Strengths and Weaknesses of AT and MOA

The preceding sections have given an overview of two systems that provide a mathematical treatment for n-dimensional array data structures. Both systems have their roots in APL, but both have made significant contributions beyond APL.

MOA is essentially a mathematical notation intended for use at the blackboard or in doing proofs of theorems. The target has been mathematical areas where flat arrays are conventionally used, such as descriptions of registers and memories in computers, and in vectors and matrices used in scientific computation.

AT has been developed as an axiomatic theory intended to suit descriptions of broad areas of finite mathematics and also to serve as a computational notation for constructive solutions to problems. The concept of nesting is central to AT; flat arrays are simply those arrays all of whose items are scalars. It is pleasing that these two en *EACH tell A* is

[0 1 2]	[0 1]
[0]	[0 1 2 3]

he *unary pervasive* operations of AT, such as *abs*, sin, etc., satisfy the equation

$$f\,A = EACH\ f\ A$$

hich implies that the result has the same nesting structure as A with the atoms at the aves of the array replaced by the result of applying f to the corresponding leaves of .

he *binary pervasive* operations of AT, such as *plus*, *minus*, etc., satisfy the equation

$$A\ f\ B = EACH\ f\ (A\ pack\ B)$$

here *pack* is an operation that produces an array of pairs of items from A and B in orresponding positions. If one of A or B is an atom then it is replicated to the shape f the other. If they are not of the same shape then they are *trimmed* to have the same hape before doing the pairing. There is a related operation *flip* that assumes that A nd B are of the same shape and has the effect of interchanging the top two levels.

he effect of this identity is that the result of such an operation between two nested rrays of the same structure is an array of the same structure with leaves formed by pplying the operation on the corresponding leaves of the arguments.

ince MOA has only flat arrays, it has a more elaborate second order operation, enoted by Ω, which applies an operation across partitions of an array, provided the esults can be glued together into a flat array again. The second order operation has wo arguments, a left argument that is an array operation and a right argument that is vector of one or two integers. The result of Ω differs depending on whether the left rgument is a unary or binary operation.

n the case that the left argument f is unary, the definition of $f\Omega_{<C>}A$ can be xpressed in AT as *mix EACH f (C takeright axes A split A)*, where *axes* is ell valence. In words, the result is obtained by partitioning A so that the last C axes re placed in the partitions, apply f to each partition, and mix the resulting arrays to a lat one. This is well defined only if all the results of applying f to the partitions of A re of the same shape. For example using *reverse*, if A is

1	2	3	4	13	14	15	16
5	6	7	8	17	18	19	20
9	10	11	12	21	22	23	24

then $_{\phi}\Omega_{<2>}\ A$ is

9	10	11	12	21	22	23	24
5	6	7	8	17	18	19	20
1	2	3	4	13	14	15	16

In the case of the left argument g being binary, then the definition of $A\ _g\Omega_{<C,D>}B$ can be expressed in AT as
mix EACH g ((C takeright axes A split A) pack (D takeright axes B split B)).
In words, the result is obtained by partitioning both A and B as above, applying g to the pairs formed from the partitions, and mixing the result into a flat array. An example using *rotate* with the same A as above is $<1\ 2>_{\theta}\Omega_{<1\ 2>}\ A$

5	6	7	8	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	17	18	19	20

which rotates the first plane of A by one and the second plane by two.

The *rank* operation in MOA can only partition along the last axes of an argument. To partition along others, a *generalized transpose* must be applied first. In AT, the generality of split allows direct partitioning along a chosen set of axes. The restriction in MOA is imposed by the constraints on the number of arguments and on the fact that all arrays are flat.

5.2. Reduction and Scan

There is a natural extension of any binary operation to a list by applying the operation between each pair of items in the array. This process is called *reduction*. The terminology comes from APL, where the corresponding second order function can only be applied to a binary scalar operation and reduces a list of scalars to a single one. MOA follows APL's lead and introduces a second order function *red*. It is restricted to be used on the binary scalar operations. Thus, if A is a list of numbers then $_+red\ A$ gives the sum of the numbers. The introduction of a reduction operation raises a number of difficult questions. What should its effect be when the resulting operation

1	5	9
13	17	21

2	6	10
14	18	22

3	7	11
15	19	23

4	8	12
16	20	24

and <0 1 0> ⓘ A is

1	5	9
14	18	22

The use of the direct mapping of the axes in the AT version rather than the inver
mapping used in MOA is permitted because the left argument can be a nested array,

A closely related idea to axis transposition is that of partitioning an array along on
or more axes. There are four primitive operations in AT that do this: *rows*, *cols*, *rai*
and *split*. The first three are special cases of the last one which does a general parti
tioning. The expression A *split* B partitions B such that the axes mentioned in
becomes axes of the items of the result and the remaining axes are the axes of th
result. For example, if B is

1	2	3	4
5	6	7	8
9	10	11	12

13	14	15	16
17	18	19	20
21	22	23	24

then [0,2] *split* B is

1	2	3	4
13	14	15	16

5	6	7	8
17	18	19	20

9	10	11	12
21	22	23	24

By reordering the axis numbers in the left argument an axis transposition in the items
can also be achieved. For example, [2,0] *split* B is

1	13
2	14
3	15
4	16

5	17
6	18
7	19
8	20

9	21
10	22
11	23
12	24

The operation *raise* is defined in terms of split by

$$A \text{ raise } B = (A \text{ drop tell valence } B) \text{ split } B$$

The **AT** operations *mix* and *blend* undo a partitioning. With *mix*, the axes of the items are added to the right end of the axes of the argument; with *blend*, its left argument specifics where the axes of the items are to be put. These operations satisfy the identities

$$\text{mix } (A \text{ raise } B) = B$$

and

$$A \text{ blend } (A \text{ split } B) = B$$

There are no corresponding operations in MOA that build and undo partitions since the AT operations assume the existence of nested arrays. However, in section 5.1 we discuss a second order function in MOA that operates on partitions.

5. Second Order Functions

Both AT and MOA have inherited the concept of second order functions from APL, where they are called *operators*. In this section the principal second order functions of AT (called *transformers*) and MOA (called *second order operations*) are described. In AT, a transformer T is applied to a single operation f by the expression $T\ f$. If T_1 takes two operations then the form is $T_1\ [f, g]$. In both cases the result is an operation which can be applied to an array by juxtaposition. A given transformer has a fixed number of operations to which it applies.

In MOA, the application notation is similar in spirit to APL, but uses subscripting to make the resulting operation easier to distinguish visually. A second order operation can take one or two arguments. If it takes one it is a subscript on either side. If it takes two arguments they are placed as subscripts on either side, but only one of the arguments needs to be a function.

5.1. Positional Transforms and Axis

The most fundamental transformer in AT is *EACH*, which behaves as a mapping function over arrays. Given an arbitrary operation f and an array A, then *EACH* $f\ A$ results in an array of the same shape as A with each of the items being the result of applying f to the corresponding item of A. For example, given the table A,

3	2
1	4

1	2	3	4
5	6	7	8
9	10	11	12

 is

9	10	11	12
5	6	7	8
1	2	3	4

A is

9	10	11	12
1	2	3	4
5	6	7	8

d 1 0 1 / A is

1	2	3	4
9	10	11	12

7. Axis Transposition and Partitions

e concept of axis transposition is familiar from the matrix transpose operation
ear algebra. If A is the matrix

45	39	21
14	256	-5

en its transpose is

45	14
39	256
21	-5

In AT, the operation *transpose* has this functionality, while in MOA the corresponding operation is denoted by unary ⍉. For a general array of $n \geq 2$ dimensions, both AT and MOA provide a generalized version of this operation that can be used to do an arbitrary remapping of the axes, including the combining of two axes resulting in a diagonalization.

In AT, the operation is called *fuse* and its left argument indicates the mapping of the axes of the right argument. The left argument is encoded so that the value of the i^{th} item of the left argument indicates which axis (or axes) of the right argument are mapped to the i^{th} axis of the result. Thus, the tally of the left argument indicates the valence of the result. Let A be the 3 dimensional array

1	2	3	4
5	6	7	8
9	10	11	12

13	14	15	16
17	18	19	20
21	22	23	24

Then [2,0,1] *fuse* A is

1	5	9
13	17	21

2	6	10
14	18	22

3	7	11
15	19	23

4	8	12
16	20	24

and [[0,2],1] *fuse* A is

1	5	9
14	18	22

In MOA, following APL, the operation is called *generalized transpose* and is denoted by binary ⍉. In this version the left argument is encoded by having the value of the i^{th} item in the left argument indicate where th i^{th} axis of the argument is to be mapped. Thus, <1 2 0> ⍉ A is

9	11	13

abs [3,−5,−9]

3	5	9

[3,4,5] + 5

8	9	10

AT extends this idea by applying it recursively at each level of a nested arr
described in section 5.1 below.

4.5. Operations on Lists

There are many operations that are naturally defined for lists. For example, giv two lists A and B, the list made up of the items of A followed by the items of B i convenient list to be able to build. In AT this is done by the operation *link*, whe in MOA it is done by the operation *catenate*, which is denoted by the symbol ,.

The operations *take* and *drop* are used to select a contiguous part of a list. In AT left argument must be positive and indicates how many items to take or drop fr the front of the list. There are operations *takeright* and *dropright* that operate fr the end of the list. MOA follows APL's lead by permitting the left argument to b negative integer to indicate that the operation should be done at the end of the li MOA uses ↑ to denote *take* and ↓ for *drop*.

The operations *reverse* and *rotate* in both MOA and AT behave like their APL co terparts on lists: *reverse* returns the list that has the items of the argument in reve order, and *rotate* uses the left argument to determine how much to shift the items the right argument to the left in a cyclical fashion. MOA uses the symbol ϕ *reverse* and θ for *rotate*.

The operation *sublist* in AT uses a left argument consisting of a boolean list to sel the items of the right argument in the corresponding positions where a *true* value found. MOA has a similar operation *compress*, denoted by /, which has the sa effect. Following APL, MOA also has an operation *expand*, denoted by \ , whi uses a boolean list to expand a list of numbers placing items of the given list or zer depending on whether the boolean entry is true or not.

4.6. Extension of List Operations to Arrays

The operations described in the previous section are extended to higher dimensional arrays and to scalars in different ways in AT and MOA. In AT, the operations are carried out on the list of items of the array. Depending on the operation, the result may be a list or a higher dimensional array. For example, if *A* is

1	2	3	4
5	6	7	8
9	10	11	12

then *reverse A* is

12	11	10	9
8	7	6	5
4	3	2	1

2 *rotate A* is

3	4	5	6
7	8	9	10
11	12	1	2

A > 5 *sublist A* is

6	7	8	9	10	11	12

and *A mod* 2 *match* 0 *sublist A* is

2	4	6	8	10	12

In MOA, the operations are carried out on the lists selected along the first dimension of an array. For example, with the same *A*,

0	1	2	3	4	5	6	7	8	9

n MOA, the operation ψ is extended to do multiple selections, whereas in AT the s a separate operation *choose* to do this. These are designed around the identities

$$tell\ shape\ A\ choose\ A = A$$

n AT, and

$$(\iota\rho A)\psi A = A$$

in MOA. This multiple selection is associative in both systems, that is,

$$(A\ choose\ B)\ choose\ C = A\ choose\ (B\ choose\ C)$$

and

$$(A\psi B)\psi C = A\psi(B\psi C)$$

provided A is an array of valid full indices for B and B is likewise an array of vali full indices for C.

MOA also extends ψ to do a subarray selection with a partial index. For example, A is

6	7	8	9
10	11	12	13
14	15	16	17

18	19	20	21
22	23	24	25
26	27	28	29

the $<1>\psi A$ is

18	19	20	21
22	23	24	25
26	27	28	29

and $<0\ 2>\psi A$ is

14	15	16	17

AT does not have an operation to do this directly, but it can easily be accomplishe using the length of the partial index vector to partition the array using *raise* and the selecting the partition with *pick*. For example, [0,2] *pick* (2 *raise* A) is

14	15	16	17

4.3. Array Construction

Both MOA and AT have adopted APL's approach to constructing arrays of higher dimension. The operation *reshape* in AT has identical semantics to the binary operation ρ in MOA for the arrays for which the MOA version is defined. If A is a shape vector and B is a nonempty list, then A *reshape* B is the array of shape given by A with items chosen from B in a row major order. If B does not have enough items, its items are used cyclically until the result is filled. For example, [2,3,4] *reshape tell* 5 is

0	1	2	3
4	0	1	2
3	4	0	1

2	3	4	0
1	2	3	4
0	1	2	3

In MOA, the left argument of reshape must be a vector, whereas in AT it can be an integer to generate a list.

There is also an operation in both systems that transforms an array of n dimensions into an array with 1 dimension. In AT this is called *list* while in MOA it is *rav*. Both systems satisfy a structuring identity. In AT it is

$$shape\ A\ reshape\ list\ A = A$$

whereas in MOA it is

$$(\rho A)\rho\ rav\ A = A$$

The extent of the list of items of an array is the number of items in the array. This is expressed in AT by

$$tally\ A = shape\ list\ A$$

and in MOA by

$$rav\ \tau A = \rho\ rav\ A$$

4.4. Scalar Operations

Both AT and MOA follow APL and use pointwise extension of scalar operations to entire arrays. Examples are

[2,3,4] + [7,8,9]

n MOA, the corresponding concept is a small number of second order functions, alled *second order operations*, which evolved from similar ones in APL. They are estricted by the requirement that the operation resulting from an application of a econd order operation to an operation must be guaranteed to always produce a flat rray.

. The Operations of MOA

he operations are discussed in roughly the same order in which they are presented n [Mullin 88], although in some cases the description is deferred until a complete escription can be made. The Appendix to this paper is a Nial model for the MOA perations.

.1. Array Measurement Operations

ne feature that distinguished APL from earlier programming languages was its use f functions that provided information on the size and dimensionality of arrays. This s a necessity in APL since arrays can be formed dynamically and passed to functions hat do not know how they were formed. Both MOA and AT provide such functions. here are three such operations in both MOA and AT:

measurement	AT	MOA
number of dimensions	valence	δ
vector of extents	shape	unary ρ
number of items	tally	τ

he operations *valence* and *tally* behave the same as their MOA counterparts, but *hape* in AT is a little unusual. Because it is convenient to deal with the shape of lists s a single integer, the *shape* operation in AT returns either an integer or a list of ntegers depending on whether the argument has one dimension or not.

.2. Indexing

n MOA, limited use is made of a notation based on subscript selection as done in ortran, Algol 60 and APL. For the array A of shape [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

he notation $A[1;0;2]$ denotes the element in the second plane in the first row and

third column, namely, 19. MOA, like APL, extends this notation to select slices along each axis.

This bracket notation for indexing is not functional in that no single symbol is used to denote the function, and the address must be given component by component. Hence, it cannot be used in conjunction with second order operations or to express indexing of an arbitrary n-dimensional array. For these reasons, both MOA and AT (and extended APLs) introduce an operation that does the equivalent of subscript selection.

In AT, the operation *pick* is used to select one item from an array. For example, for the above array A, the corresponding selection is done with the expression $[1,0,2]$ *pick* A.

In general, I *pick* A denotes the item of array A at the address given by array I. In MOA, the corresponding operation is *index*, denoted by ψ.

The notion of a valid address in AT and of a valid index in MOA are identical. For an array of n dimensions with shape $[s_0, \cdots, s_{n-1}]$, an index $[i_0, \cdots, i_{n-1}]$ is valid if $0 \le i_k < s_k$, $k=0,1,\ldots,n-1$. For arrays with precisely one dimension, it is convenient to use an integer to denote an index rather than insisting that it must be a list of length one (a solitary). AT treats an integer as a valid index of a list. Both AT and MOA treat the empty numeric vector (*Null* in AT and Θ in MOA) to be the valid index for a scalar.

Both systems have an operation to generate the valid indices for a shape. In AT, the operation is called *tell*; in MOA, it is denoted by unary ι. In AT, *tell* generates an array of the given shape with items the corresponding addresses. In MOA, ι generates an array of one dimension higher with the items corresponding to the rows of the array. For example, *tell* [2,3] is

[0 0]	[0 1]	[0 2]
[1 0]	[1 1]	[1 2]

while ι<2 3> is

0	0
0	1
0	2

1	0
1	1
1	2

Both of these operations are defined for an integer argument and in this case generate the list of integers of the length of the argument starting at zero. For example, *tell* 10

concept	array theory	mathematics of arrays
a scalar object	atom	scalar
component of an array	item	element
subscript	address	index
vector of extents	shape	shape
number of dimensions	valence	dimensionality, rank
1-dimensional array	list	vector
2-dimensional array	table	matrix
vector of length 0	void	the empty vector, Θ
vector of length 1	singleton	-
vector of length 2	pair	-

Table 1. Comparison of Terminology

he two systems also use different conventions for writing and displaying arrays. I
T, 1-dimensional arrays (lists) may be denoted by either strand notatio
4 56 35 27, or by bracket-comma notation: [34,56,35,27]. In MOA the notation fo
vector is <34 56 35 27>. For both systems, the denotation of an array of dimen
onality ≥2 is achieved using an operation that reshapes a list to have a given shape.

n this paper we display pictures of arrays using the output diagrams of Nial. A
calar array is displayed directly. A 1 or 2-dimensional array is displayed in a grid o
nes with each cell holding the diagram of the corresponding item. The diagram of
igher dimensional array is displayed in 2 dimensions by displaying two dimension
ubarrays in alternating directions, with increasing spacing between higher dimen
ons. For example, the array 2 3 3 2 *reshape count* 48 is displayed as

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

. Operations on Arrays

he choice of operations on arrays in the two systems is affected by the limitations o
e respective universe of arrays. In MOA, the result of every operation must be a fla
ray; whereas in AT, if it is more convenient to store the result of an operation as a

array of arrays, this possibility exists. Similarly, the encoding of data as an argument to an operation must be as either one or two flat arrays in MOA, whereas in AT, the argument is a single array, but which may have any number of arrays as items.

In MOA, all operations are either unary or binary. Following APL, the same symbol can be used for both a unary or a binary operation. For example, the symbol ρ is used both for *shape* and *reshape*. Expressions consisting of a number of binary operations are associated right to left with no precedence.

In AT, each operation is unary, although it may be used in infix notation with the interpretation that its argument is formed by the list of length two of the values on each side. In an expression with a sequence of infix uses of operations, the association is left to right with no precedence.

In AT, one of the design goals was to develop a set of operations that are defined for all arrays and return an array value. Such operations are called *total* operations. In addition, the operations are constrained to obey certain *universal* laws or equations.

For example, the operation *rows* converts a matrix to a list of the list of items in each row, and the operation *mix* converts an array of equishaped arrays to an array with one less level of depth and with shape the concatenation of the shape of the original array and the shape of the items. On a table T, *mix* is the left inverse of *rows* and hence the equation

mix rows T = T

holds for all nonempty tables. In AT, the operations *mix* and *rows* have been extended to all arrays so that the above equation holds universally [Jenkins 84].

The search for total operations that satisfy universal equations has driven much of the development of array theory. This has led to an elegant mathematical system, but one which has placed constraints on some operations that may not be practical for computational use. Nial implements AT in its full generality and provides an experimental test bed for AT concepts.

Like APL, MOA has not attempted to achieve total operations. Instead the operations are defined over their natural domains and then extended to the extent that seems practical and/or elegant. There are some universal laws, but to a large extent they are those inherented from APL.

The set of operations in AT is very large and cannot be described in full detail here. A complete description of the operations, with many of the universal equations stated, can be found in [Jenkins 85]. Here we focus on a comparison of similar operations in AT and MOA, presenting all the central operations of MOA in terms of their counterparts in AT.

Both AT and MOA also have second order functions, that is functions whose arguments are operations. In AT, second order functions are called *transformers* and may take one or more operations as their argument. The transformers are general and may take any operation for their argument. Transformers can be constructed by a number of mechanisms in AT and give Nial much of its functional language flavour.

n Algol 60 and Fortran, arrays are treated as data containers for scalar values. Th rray is viewed as a collection of variables each capable of holding a scalar valu Arrays can be passed as parameters to procedures, but cannot be returned as resul rom value returning functions. Similar restrictions have been passed on in most su equent procedural languages such as Pascal and Turing.

APL extended the array concept by treating it as a value at the same level as a sing number or character, thus permitting expressions with array values to be used. One he original contributions of APL was to give a consistent treatment of scalar valu s arrays, thus allowing the universe of data objects in the language to be entir arrays. This brought to the notation a mathematical uniformity that has much appe n APL as originally developed, an array was either an array of characters or an arr of numbers. A formal description of APL arrays and the operations of APL w developed as part of the APL standard[ISO82].

This paper examines two mathematical treatments of arrays that have followed fro APL concepts. The first was motivated by a desire to extend APL to permit arrays arrays in a way consistent with the nesting concepts of set theory. The treatme called *array theory* (AT) [More 73], has gone through several versions and has be he basis for the data structures in the programming language Nial [Jenkins 85]. A earlier version motivated the extension of APL arrays to nested arrays APL2[Brown 84]. Based on experience gained using Nial, efforts are underway refine array theory and to publish a definitive account [More 90, Franksen 90].

The second mathematical treatment was developed to provide a firm mathemati reasoning system for algorithms involving flat arrays of numbers. This treatme called *a Mathematics of Arrays* (MOA) [Mullin 88], has been used to prove theore about register transfer operations in hardware design, and to describe the partition of linear algebra operations for parallel architectures [Mullin 89]. MOA follo APL more closely than AT and is largely concerned with having a succinct notati in which definitions can be stated and theorems on array transformations can proved. This work has attracted the attention of researchers attempting to exte functional languages to include array objects.

Both AT and MOA have their roots in APL and it is not surprising that they are co sistent with each other as theories of arrays. The purpose of this paper is twofo first, to explain the correspondence between concepts in the two notations in orde assist users of the two notations to communicate, and second, to compare effectiveness of the two approaches for their purposes.

2. The Universe of Arrays

The fundamental concepts of arrays in the two systems are identical. In both, array is a multidimensional rectangular object with items placed at locatio described by a 0-origin addressing scheme. An array can have an arbitrary num of dimensions, including zero. The object is viewed as being laid out alo

orthogonal axes, one for each dimension. The length of the object along each *axis* is called the *extent* of the corresponding dimension. The vector formed from the extents is called the *shape* of the array.

In MOA, the items of arrays are numbers. The formal development involves arrays of integers, but it is clear that the theory is applicable to arrays of numbers in general, and can easily be extended to arrays of any homogeneous scalar type. A scalar in this treatment is an array with no dimensions and with shape the empty vector. There is no concept of an item of a scalar.

In AT, the items of arrays are themselves arrays; thus, AT arrays are inherently nested. They can be vectors of matrices of integers, or matrices of 3-dimensional arrays of real numbers. The nesting recursion is terminated by postulating that the scalar arrays, of which there are seven types, are self-nesting. AT arrays are inherently heterogeneous since there are no constraints on the types of items of an individual array. Thus, the universe of arrays described by AT is much richer than that described by MOA.

The two approaches have a consistent interpretation of flat arrays. In [Gull 79], it is shown that the flat arrays of APL can be viewed in two ways. One is to view a flat array as hiding *atomic* data that is never directly accessible. The standard indexing function that corresponds to subscript notation, A[I], selects a hidden item and then containerizes it as a 0-dimensional array. The other view is that flat arrays simply contain scalar arrays as their items and the standard indexing function selects the item.

The two views are termed the *grounded* and *floating* views respectively. The difference between them is only clear when one wants to extend the universe to include nested arrays using an operation that *scalarizes* an arbitrary array. The two different views of arrays has led to two different extensions of APL systems. In [Jenkins 80], it is shown that the floating system of arrays is implied if the nesting concept in APL is the same as that used in Lisp.

In this terminology, MOA can be interpreted as either floating or grounded, whereas AT is a floating system of arrays. For the purposes of the comparison we will interpret the universe of arrays in MOA as a depth-limited floating system.

Different terminology is used in AT and MOA for the same concepts. Table 1 gives the correspondences.

14

A Comparison of Array Theory and a Mathematics of Arrays

Michael A. Jenkins
Queen's University
Kingston, Ontario, Canada

Lenore R. Mullin
Centre de recherche informatique de Montreal
Montreal, Quebec, Canada

Abstract

Array-based programming began with APL. Two mathematical treatments of array computations have evolved from the data concepts of APL. The first, More's array theory, extends APL concepts to include nested arrays and systematic treatment of second order functions. More recently, Mullin has developed a mathematical treatment of flat arrays that is much closer to the original APL concepts. The two approaches are compared and evaluated.

1. Introduction

The modern concept of an array as a multidimensional data structure has evolved from its use in early programming languages. The original motivation of arrays was to find a counterpart to subscript notation used for sequences, vectors, matrices and higher dimensional objects. The array concept was introduced in Fortran with static arrays of a fixed type and with a limited number of dimensions. It was extended in Algol 60 by allowing an arbitrary (≥ 1) number of dimensions and by having the size of the array determined dynamically at block entry.

This research was supported in part by the Ontario Information Technology Research Centre, the Centre de Reserche informatique de Montreal, and the Natural Sciences and Engineering Research Council of Canada.