Array Theory and the Design of Nial

MICHAEL JENKINS

Preface

This monograph reports on research that I conducted in 1999 while on a visit to the Technical University of Denmark visiting Peter Falster. Together we wrote a longer technical report, but it was never submitted for publication. I am providing this abbreviated version as a free electronic book to ensure that the ideas are available to the academic and commercial communities interested in array-base programming languages.

I spent a good portion of my working life on a project to create an effective programming methodology based on mathematical principles. The work was begun in 1979 in collaboration with Dr. Trenchard More of the IBM Scientific Center, Cambridge, but grew out of extensive experience with APL and other programming languages in the field of numerical analysis. The purpose of this monograph is not to describe the historical developments and the various players, but to document, for future efforts in programming language projects, the rationale behind the many design decisions that were made in my work on array theory and Nial.

The work grew out of an interest in APL that started when I worked in the IBM APL project at IBM Research, Yorktown, immediately after my graduate work at Stanford. On moving to Queen's University in 1969, I took part in many of the early APL conferences, reporting on work I did on a number of topics. Trenchard More attended many of these conferences and described his work on a mathematical treatment of array data structures that he called ***Array Theory***. In the early 1980s we began a collaboration to build a programming system that would utilize his array mathematics with my knowledge of programming language concepts and implementation methodology. The target language was

called the Nested Interactive Array Language, Nial. With support from IBM and Queen's University I built a team to implement the Q'Nial interpreter, originally designed under Unix, for both the IBM personal computer and the IBM mainframe. Using early versions of C compilers, a portable version of the interpreter was ready for release in late 1983.

Designing and implementing a programming notation is a nontrivial task. There are many tradeoffs to be made and many detailed decisions on the exact behavior expected by the specific functions chosen for the system. We used two main guiding principles. First, that the expression language would be based on array theory, and second, that the programming language constructs would reflect current practice at the time. The decision was also made to assume an ASCII character set for the language to avoid the complications that APL's special character set imposed on the technology available in the 1980s.

A first task was to design the notation for array theory expressions based on ASCII characters rather than the APL based notation that More had developed. We also had to decide on the choice of functionality that would be implemented directly in the interpreter, depending on using the definition mechanism to extend the functionality. The goal to have a portable interpreter that would run effectively on both personal computers and mainframes of the 1980s severely constrained the choices.

Some of the material presented below is extracted from a joint report prepared with Peter Falster at the Technical University of Denmark in the summer of 1999. I have extracted the material that concerns the relationship between the the versions of the language Nial as implemented in Q'Nial and the array theory that underlies the mathematics of the data structures of Nial.

The author is indebted to Trenchard More, Ole Franksen, Peter Falster, Janice Glasgow and Jean Michel for their many conversations over the years that have contributed to the ideas presented here, and to Carl McCrosky, Lynn Sutherland, who, with many others on the Queen's research team, worked with me on the development and refinement of Q'Nial.

Michael Jenkins
Kingston, Ontario, Canada
June 2013

CHAPTER 1

Array Theory Concepts

Trenchard More conceived of the idea of a one-sorted theory of nested rectangular arrays whose items are themselves arrays. He began the work in the early 1960s at MIT and Yale. At Yale he became interested in APL, and joined the IBM APL project in 1968 to be able to work full time on the theory. Jenkins became interested in nested array concepts through participation in the Minnowbrook APL Implementers workshops. This interest was shared with Walter Gull, resulting in a paper proposing the extension of APL to have recursive arrays [GuJe79]. In 1979, Jenkins worked with More at IBM and they began a collaboration that eventually led to the Nial Project. A more complete discussion of the evolution of array theory ideas can be found in More's papers [More73, More81, More06] and in [Jenk07].

Array theory concepts and notations have evolved throughout the course of More's work. In this chapter we present an overview of the concepts in the terminology and notation adopted for use as the mathematical basis for Nial. This choice of notation for the manuscript will assist the reader in using Q'Nial as a test bed to further explore the ideas. Trenchard More has worked on generalizations of some of the concepts presented here and has adopted different terminology in some cases [More93].

Data Objects

Array Theory is primarily a theory about the definition and manipulation of array data objects. Every data object in the theory is an array, even numbers and characters, which are given structure as array scalars. The data objects are viewed as collections of data objects arranged along axes with an addressing scheme to indicate where each item of the collection is located. The items of the collection are themselves arrays.

We illustrate arrays by using box diagrams where each box holds an item. For example, the diagram

```
+-+--+--+--+
|1| 2| 3| 4|
+-+--+--+--+
|5| 6| 7| 8|
+-+--+--+--+
|9|10|11|12|
+-+--+--+--+
```

depicts a 3 by 4 array of two dimensions. An array with one dimension is called a **list**, one with two dimensions is called a **table**. An array is rectangular in that there are the same number of items along any line in one direction. In the example, all the rows have 4 items and all the columns have 3 items. Because of this property, the outer structure of an array can be described by a list of extents in each direction.

The **shape** of an array is the description of its rectangular structure. For the above example it is the list of two items

```
+-+-+
|3|4|
+-+-+
```

The number of items in an array is called the **tally** and is given by the product of the extents of the axes.

On the page we depict higher dimensional arrays by laying out slices along the last two dimensions. For example, the three dimensional array of shape 2 by 3 by 4 of the first 24 integers is depicted as

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

where the left table is the first slice. The number of dimensions of an array is called its **valence**.

The items of an array can themselves be other arrays. For example,

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

is a 2 by 3 table where each item is a list of length 4. We say the lists are **nested** inside the table. In fact, this last array is a rearrangement of the data in the previous array. Much of array theory is concerned with functions that map between such arrays.

It is also possible to have an array in which the nested items are of different shapes. For example,

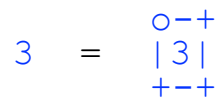
```
+---+-----+-------+---------+-----------+
|+-+|+-+-+|+-+-+-+|+-+-+-+-+|+-+-+-+-+-+|
||1|||1|2|||1|2|3|||1|2|3|4|||1|2|3|4|5||
|+-+|+-+-+|+-+-+-+|+-+-+-+-+|+-+-+-+-+-+|
+---+-----+-------+---------+-----------+
```

is a list of 5 lists with lengths increasing from one to five.

An array with no axes is also possible. Indeed, the numbers and characters are viewed as arrays with no axes. An array with no axes is called a **single** and has exactly one item. We diagram a single by drawing a box around the item and placing a small circle in the upper left corner. For example, the single holding the list of the first 3 integers is depicted as

```
o-------+
|+-+-+-+|
||1|2|3||
|+-+-+-+|
+-------+
```

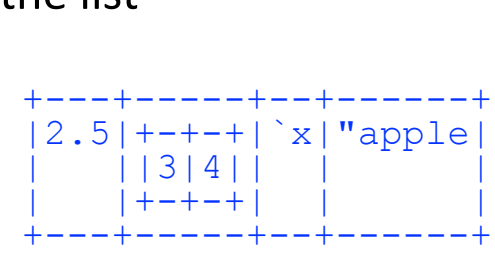
An important issue in the design of a theory of arrays is whether a scalar like 3 is the same as the single holding it or is different from it. Should



or should the two arrays be different? The APL community agonized over this issue for many years and eventually different APL systems decided on different answers to the issue. In array theory, the equations of the theory imply that this equality must hold as the means of terminating nesting.

As well as including integers as fundamental data, array theory considers other basic types as well. These include Boolean values, real numbers, characters, symbolic names (called **phrases**), and special values to mark errors and undefined values (called **faults**). All of these basic data objects are called atomic arrays, or **atoms**, and are self-containing singles. An array, all of whose items are atomic, is called a **simple** array.

There are no restrictions on what arrays can be collected together as items of another array. For example, the list



contains the real 2.5, the pair [3, 4] the character `x, and the phrase "apple.

The extent of any axis of an array can be any finite integer including zero. When one or more zeros appears in the shape of an array, the array is known to have no items and is said to be **empty**. The empty list, Null is the principal empty list since it is the shape of a single. In array theory, as in many programming languages, strings are treated as lists of characters, so the notation 'abc' denotes a list of three characters and '' is the empty string. In APL, the two empty lists that correspond to Null and '' are different arrays since they lead to different results when certain functions are applied to them.

The decision as to whether array theory should have only one empty list or many is a central choice in the theory. In all of his work, Trenchard More has followed the path of having multiple empty lists. In Version 4.1 of Q'Nial, designed jointly by More and Jenkins, there are multiple empty lists. In Version

6.21 of Q'Nial, Jenkins changed to an array theory that has only one empty list, resulting in the equation

Null = ''.

In order to distinguish these two main versions of the theory, we refer to the one with many empty lists as V4 array theory and the one with a unique empty list as V6. Chapter 3 discusses the theoretical impact of the decision on empties.

An important feature of the data objects of array theory is that any collection of arrays can be used to form a new array. This is important because it allows a single array to represent all the arguments that are provided to a function, and hence are all functions on data objects can be viewed as having a single argument.

Functional Objects

First-order functional objects, called **operations** in array theory, are used to manipulate arrays. An operation is an abstract function that takes one array as its argument and returns an array as a result. For example, if *A* is formed in Nial by

```
A := 4 3 reshape count 12
+--+--+--+
| 1| 2| 3|
+--+--+--+
| 4| 5| 6|
+--+--+--+
| 7| 8| 9|
+--+--+--+
|10|11|12|
+--+--+--+
```

then the operation, shape, when applied to *A*

```
shape A
+-+-+
|4|3|
+-+-+
```

returns the pair (a list of length two) holding the extents of the axes of A.

Array theory has many predefined operations that are used for measuring or manipulating arrays. The operations have been chosen to satisfy a large number of equations universally.

Although, the operations are monadic, array theory syntax adopts a convention that permits infix usage of operations. In the above computation for A the pure monadic form would be

```
A:= reshape [[4, 3], count 12]
```

There are several ways of forming a new operation from existing ones: by composition, by transformation, by left currying, by a parameterized

expression (equivalent to a lambda notation) and by a list of operations called an **atlas**.

An operation is transformed to another by applying a second-order function called a **transformer** to the operation. For example, the transformer, EACH transforms an operation f to the operation EACH f, which is called the EACH transform of f. When EACH f is applied to an array A, the result is an array of the same shape as A with each item being the result of applying f to the corresponding item of A.

An example using the above array A and the operation (5+) is

```
   EACH (5+) A
+--+--+--+
| 6| 7| 8|
+--+--+--+
| 9|10|11|
+--+--+--+
|12|13|14|
+--+--+--+
|15|16|17|
+--+--+--+
```

Transformers are monadic functions that map an operation to an operation. In Chapter 2 we describe the transformer expressions that can be used to form new transformers from existing ones. These include transformer composition, a parameterized operation expression (equivalent to a second order lambda notation) and a list of transformers called a **galaxy**.

The operation and transformer expressions of Array Theory syntax provide Nial with a strong functional component. It is possible to write substantial programs in Nial using only the functional subset of the language.

Syntax for Array Theory and Nial

The following chapter gives a detailed description of the syntax of Nial and the rationale behind many of the decisions made. Here we give a brief description of how the syntax is used in some simple examples in order to give an overview of array theory notation.

Nial can be viewed as having two distinct components, the mathematical expression language that describes array-theoretic computations, and the linguistic mechanisms added to make it a programming language. In order to integrate the two parts of the language in a clean fashion, it was decided that the syntactic constructs should map on to the three kinds of semantic objects in array theory: arrays, operations and transformers. Thus, there are expressions of the three kinds, and one means of defining new names to associate with expressions of the three kinds.

A one-dimensional array or list can be formed in syntax by either of two mechanisms: **bracket-comma notation** [A, B, C] or, **strand notation** A B C where A, B and C are any array expressions. The bracket-comma notation can denote lists of any length, whereas a strand is always of length 2 or more. In a strand, the individual array expressions may need to be parenthesized in order to indicate the intended grouping.

To construct an array with no axes we use the operation single. For example,

```
        single 'hello world'
o-----------+
|hello world|
+-----------+
```

We apply an operation to an array by placing the operation immediately before the array. The general syntax form is f A, where f is any operation expression and A is any array expression. However A may need to be parenthesized to get the intended scope for the application.

To construct higher dimensional arrays, the operation reshape is used to convert a list of items to an array of the appropriate valence. For example,

```
      2 3 reshape 3 7 5 2 7 4
3  7  5
2  7  4
```

constructs a table as shown. Note that we have displayed the output without boxes around the items. This is called the **sketch** picture of an array and is used by default in Q'Nial.

In the above usage it appears that reshape is a binary operation with two arguments. However, the syntax rules allow us to write any application of an operation to a list of length two as an infix usage. In general,

A f B = (A f) B = f (A B)

If an infix usage involves array expressions that also have a prefix operation application then the prefix applications are done first. For example, in

```
      sum [2, 3] * count 4
5  10  15  20
```

sum is applied to the pair [2, 3] and count is applied to 4 before * ,which denotes product, is applied. The example could also be written as

```
      sum 2 3 * count 4
5  10  15  20
```

where the strand 2 3 would be formed prior to applying sum.

Another writing of the same example is

```
      2 + 3 * count 4
5  10  15  20
```

where + is an alternative notation for sum. This form of the example illustrates that multiple infix uses of operations are evaluated left to right.

The operation expressions that are used in an application are often transforms, that is, operations formed by applying a transformer to an operation. For example,

```
      (EACH first) [2 3 4, 'abc', 3.5 4.5 5.5]
2 a 3.5
```

The parentheses around the transform are not necessary, since the syntax rules imply that

T f A = (T f) A.

However, in the example

```
      EACH (first rest) [2 3 4, 'abc', 3.5 4.5]
3 b 4.5
```

they are needed so that the **composition** of the two operations first and rest is used on each item.

A transform can also have an infix usage. For example, the transformers EACHLEFT, EACHRIGHT and EACHBOTH are often applied to operations that expect a pair as an argument and are used infix as in

```
      3 4 5 EACHLEFT reshape 'abcde'
+---+----+-----+
|abc|abcd|abcde|
+---+----+-----+
```

The definition

```
frequency IS OPERATION Values A
{ EACH sum ( Values EACHLEFT EACHRIGHT = A ) }
```

shows a common use of the transformers where EACHLEFT is transforming the transform EACHRIGHT =. A good exercise is to experiment with this definition so that you understand how it computes the frequency that the items given in the first array argument Values occur in the second array argument A. Consider the example

```
      'abc' frequency 'The cat sat on the baseball bat'
5 3 1
```

The above examples illustrate some of the more commonly used syntax for writing array theory expressions. The general rules for forming them are given in Chapter 2.

CHAPTER 2

Syntax Decisions in Nial

We present here a summary of the syntax of Nial with some commentary on the decisions made in the design. The present syntax was designed jointly by Mike Jenkins and Trenchard More, with some input from Carl McCrosky and other people working on the Nial project at the time. The syntax exists in two versions: V4, used in V4.1 Q'Nial and V6 used inV6.21 and later versions of Q'Nial. To a large extent the V6 syntax is a subset of the V4 syntax. In this note we describe the full V4 syntax and then summarize at the end the restrictions that have been made in V6.

Syntax can be broken up into two aspects: the lexical rules for forming tokens, and the grammar rules for forming syntactic constructs from tokens. We begin with the Lexical rules in Section 1.

Lexical Rules

The input to a language processor such as Q'Nial is a string of characters to be analyzed. We use the word **glyph** to mean a single character symbol such as a letter, digit, delimiter, operation symbol or diacritical mark. The lexical rules of the language indicate which sequences of glyphs are to be taken as single units that are not examined internally by the grammar rules described in the next section.

We use the term **token** to mean a sequence of glyphs determined by the lexical rules. The tokens fall into various classes.

Identifiers: Identifiers consist of a letter followed by zero or more letters or digits. The rule is expressed by the following regular expression:

<identifier> ::= <letter> (<letter> | <digit>)*

A design choice here is the set of glyphs chosen to be letters. In Nial, it is the upper and lower case Roman letters plus underscore, _ and ampersand, &.

The choice of which glyphs to allow as letters is quite arbitrary. One suggestion that has been made is to have the dash, "-" be a letter. It could still be used to name the operation minus but spaces would be required between the arguments and the glyph.

Most identifiers are used to name objects in the language, but a few are **reserved words** that are used in forming syntactic constructs. In addition, the tokens consisting of only the letters l and o are not treated as identifiers, but are used in forming constant Boolean values and bit strings.

There is no distinction in the case of letters forming an identifier. Thus, EACH, each and eAcH all denote the same object. This decision was made to allow users to type in a convenient fashion but still allow a convention for the canonical spelling of identifiers. The canonical spellings are: data variables and named array expressions begin with a capital letter followed

by lower case letters, named operations are all lower case letters, and named transformers and reserved words are in all upper case.

Symbols: These tokens are either a single glyph or a pair of glyphs recognized as a single token. A symbol is used like an identifier, but the glyphs are not treated as letters. A symbol can appear next to an identifier or another glyph without requiring a space.

Glyph name	Glyph(s)	Usage
dollar	$	not used
caret	^	not used
asterisk	*	product
dash	-	minus
plus sign	+	sum
equals sign	=	equal
backslash	\	not used
slash	/	divide
less than	<	lt
greater than	>	gt
tilde	~	not used
	<=	lte
	>=	gte
	~=	unequal
	:=	GETS

The glyphs marked as *not used* can be used to name any object in the language. For example, ~ could be used to name the operation not.

The set of glyphs chosen to be symbols is rather arbitrary. The above ones are chosen to follow common practice. For example, in most programming languages + denotes the binary addition operation and does not require spaces between it and its operands. If we are willing to use spaces around symbols then all the single glyphs are candidates to become letters and only the := double glyph would need to be special.

We decided not to require spaces around glyphs since it would make writing of expressions sensitive to spacing. When reading program text displayed in non-proportional fonts or on the blackboard it is difficult to

discern the spaces. Entering text that is space sensitive at the keyboard is also more error prone.

Delimiters: The glyphs commonly used for punctuation and grouping are called delimiters. These are

Glyph name	Glyph(s)	Usage
parentheses	( )	grouping
brackets	[ ]	list formation
braces	{ }	block formation
comma	,	list formation
period	.	in numbers, dot notation
colon	:	in CASE expression
semicolon	;	expression separator

The delimiters do not need a space to separate them from adjacent tokens with one exception - a colon in CASE labels must be separated from a phrase constant on the left and the equal glyph on the right.

Other Glyphs: Some other glyphs are special notations in the syntax.

Glyph name	Glyph	Usage
at symbol	@	at indexing
	@@	path indexing
number sign	#	at-all indexing, remarks
vertical bar	|	slice indexing
exclamation	!	casts, host commands
percent symbol	%	comments in definitions

Constants. These are tokens that denote specific values in Nial. There are notations for the six atomic types and for character and bit strings.

Real numbers are denoted in a standard way, using – to denote negative numbers, the . as the decimal point, and the letter *e* to begin an exponent part. There is no requirement that a digit both precede and follow the decimal point. A sequence of digits without a decimal point or an exponent denotes an integer; all other number constants denote a real number. In V4.1 of Q'Nial, complex numbers were included and were denoted by a pair

of real constants joined by the letter j. This notation has been omitted in V6.21 of Q'Nial.

The letters l and o denote Boolean values. The upper case letters are considered equivalent to the lower case ones. Bit strings are sequences of two or more Boolean values juxtaposed without spaces. Bit string constants could be eliminated without much loss in usability since the same effect can be achieved by writing a strand of Boolean numbers with spaces between each. An advantage of choosing this alternative would be that fewer identifiers are lost.

There are four forms of literal constants in Nial: atomic characters, character strings, phrases and faults. These all need to be decorated to distinguish them from identifiers and numbers. The design choices are:

Form	Decoration	Example
character	accent symbol to the left	`x
string	surrounded by single quotes	'hello world'
phrase	preceded by a double quote	"apple
fault	preceded by a question mark	??error

A strand of characters is equivalent to a string. The decorations in all these notations are omitted in the stored value and do not display in sketch mode. There are also operations that convert strings to phrases and faults to allow for spaces in the constructed object. Thus, "apple is equivalent to phrase 'apple'.

There are issues with many of the choices of glyphs. The accent symbol is very difficult to spot. Also on European keyboards it behaves as a diacritical mark and is difficult to type. However, there does not appear to be another single glyph that works as well.

In many other programming languages double quotes are used to surround strings. Also in English text a quotation is usually placed in double quotes. The use of single quotes around strings in Nial was inherited from APL. We also inherited the rule for doubling a single quote to indicate an internal single quote.

The use of a one-sided decoration for phrases is convenient for single word phrases, but is inconvenient for ones that include spaces or other symbols. Originally a phrase ended with a blank, but experience showed that it was better to have the delimiters other than ":" also terminate phrases.

It has been suggested that requiring a closing double quote mark would be a better design choice for phrases. Similarly, faults might be better with a closing decoration. The design decision to leave phrases with one-sided decorations came from their heavy use in artificial intelligence (AI) and database applications where single word symbols are frequently used.

One alternative design choice would be to merge characters with phrases and have only one kind of atomic literal data other than faults. That would eliminate the use of the accent glyph. The problem with the alternative choice is that it makes output more awkward. When displaying a list of such atoms in sketch mode, would spaces be put between the atoms or not? Also, the choice eliminates the treatment of a string as a list of characters.

Another design choice could be to eliminate phrases and use just strings in the language. However, phrases have proven very effective in many database and AI applications and eliminating them would make programming such applications less effective.

Other issues. The part of Nial that tokenizes input is very forgiving if you omit spaces between glyphs that it can determine do not form a single token. For example, if A is a variable, then you can type 2A and it is interpreted as the strand 2 A. Similarly, tally"foo is treated as the application of tally to the phrase "foo. This feature was removed in one experimental version of Q'Nial and users complained that it slowed down their use because of the additional faults generated. One other glitch occurs because of the use of – for both negative numbers and for the operation minus. The expression A–B is parsed correctly, but A–1 is parsed as the strand A –1.

Grammar Rules

The syntax of Nial consists of array theory expression syntax embedded as the expression language within a fairly conventional programming language syntax for control constructs and definitions. The design assumes that the language is used interactively. The unit of executable code is an **action**, which can be either a **definition**, an **array expression**, or a ***remark***. A definition is translated to internal form and stored in the current **workspace**. An array expression is executed and any changes to global variables caused by the execution are recorded in the workspace. A remark is simply passed over since it is intended for the reader of the code it is embedded in.

Program text may be entered directly at the prompt or by using the loaddefs operation to process a file consisting of a script of actions separated by blank lines. The script can contain additional uses of loaddefs. By using loaddefs recursively in this manner a problem solution can be broken into a hierarchy of script files.

Linguistic mechanisms. The grammatical constructs of Nial are either **definitions** or **expressions**, with the latter divided into three classes according to the three kinds of objects in Nial: array expressions, operation expressions and transformer expressions.

Definitions are of the form

<name> **IS** <expression>

where a <name> can be either an identifier or symbol. The definition mechanism permits the recursive use of the name being defined in the right side of the definition. There is also a means to declare the role of a name using a syntax for declaring external objects. This feature is useful for loading interdependent scripts, without worrying about the order in which they are loaded. It is also used to achieve mutual recursion.

The **role** of a name is either the kind of expression (array, operation, or transformer) it denotes as a predefined or user-defined association, a variable, a reserved word, or an unused identifier. The role of all the names used in the right hand expression of a definition must be known when the definition is processed. If the name being defined also appears on the right then it is assumed that it refers to the same object recursively.

The control constructs of Nial are introduced as array expressions and are given interpretations for the value returned. In general it is the value of the last expression evaluated. However, if no value is defined then a special fault value, ?noexpr is given implicitly. This special fault value is not displayed if it is the value returned in the top-level loop.

The control constructs are **if-then-else** and **case** constructs for selection, **while**, **repeat-until** and **for** loop constructs for repetition, and **assign**, using := or gets, for assignment to variables. The value fields of the constructs are array expressions whereas the body fields are sequences of array expressions separated by semicolons.

The value of an array expression sequence is the value of the last expression. If the sequence is followed by a semicolon then the default value, ?noexpr is the value. All the control constructs have matching leading and trailing keywords to make it easier to visually delimit their scope.

The Nial syntax for control constructs can be used in both a statement and an expression style. For example,

```
if A < 0 then X else Y endif
```

selects the value of either X or Y as the value to be returned, whereas

```
if A > 0 then X:=tell 20; else write 'A<=0'; endif
```

selects one of two actions to be performed. In the first case the semicolons are omitted so the selected value is the result of the *if-then-else* expression. In the second case a value is not expected and semicolons are used to terminate each expression in the selection and the result will be the ?noexpr fault.

The use of expression syntax to achieve both statement and expression effects means that the programmer must take care in the placement of semicolons to ensure that the value for an expression is passed out to the intended scope.

The remaining linguistic constructs are blocks, operation forms and transformer forms. A **block** is a construct to localize definitions and variables to a limited scope. It has three sections: local and non-local declarations, a definition sequence, and an expression sequence. The sections are separated by semicolons and the block is delimited by braces.

Declarations are used to specify whether variable names assigned in the block are local or non-local (they are local by default). Definitions given in the block are not visible outside it. The value of the block is the value of the last expression in the sequence.

An **operation form** is used to express an operation with parameters. It has two forms

OPERATION <parameters> <block>

or

OPERATION <parameters> (<expression sequence>)

There may be one or more parameter names. If there is only one, the supplied argument is assigned to it. If there is more than one then the argument must have the same number of items and its items are assigned to the parameters. In the first case, the parameters become initialized local variables in the scope of the block; in the second, the definition forms a local scope unit with the parameters as local names, but variables assigned in the expression sequence are non-local.

The second form allows the assignments made in the expression sequence to modify the surrounding scope. This is useful during debugging to make sure intermediate computations are correct. In both cases, when the operation is applied, the actual arguments are evaluated and assigned to the parameter names. Then the body is evaluated and the value returned is the result of the block or expression sequence.

An operation form can appear on the right hand side of a definition, as the argument of a transformer or can be applied directly to an array expression.

In the latter two cases it must be parenthesized. The design of operation forms is similar to lambda forms in functional languages such as Lisp except for the handling of multiple parameter names.

A **transformer form** is used to express a transformer with operation parameters. It has the form

TRANSFORMER <operation parameters> <operation expression>

where the operation expression must be parenthesized if it is not an operation form. It provides a template for an operation that is formed by replacing the operation parameters by the given operations. If the transformer definition is recursive then the replacement is repeated dynamically during execution of an application of the transformed operation until the end condition is reached.

A transformer form is a second order lambda expression. The parameter is known to be an operation and hence when the transformer is applied the interpreter uses the closure of the actual argument operation (the operation with its current environment) as the operation used in the template. In languages, such as Lisp, with functions of arbitrary order, the rules for the use of closures are much more complex.

The reserved words OP and TR can be used as abbreviations in operation and transformer definitions respectively.

SECTION 3

Array Theory Expressions

The expression language of Nial is based on a juxtaposition syntax supplemented by bracket-comma forms for lists of arrays, operations and transformers. The main feature is that an expression is formed by the juxtaposition of a sequence of expressions each of which can denote an array, an operation, or a transformer. The meaning is determined by juxtaposition rules and a general rule for left to right interpretation.

The syntax gives an interpretation to the nine juxtapositions of the three kinds of objects: arrays, operations and transformers. In the remainder of this section we explain these interpretations using the variables A and B to denote array expressions, f, g and h to denote operation expressions, and T, U and V to denote transformer expressions.

The juxtaposition A B is an array expression interpreted as a pair. The rule is extended to an arbitrary sequence of two or more array expressions, called a strand, which is interpreted a list of the length of the sequence with items that are the values of the corresponding array expressions. This extends the APL notation of having constant number lists denoted by a sequence of numbers. The more general notation was adopted by APL2.

The juxtaposition f A has a natural interpretation as the array expression that denotes operation application to an array. Similarly, T f is the operation expression that denotes transformer application to an operation. The notation T f A is interpreted as (T f) A.

In APL the second order functions use the opposite juxtaposition for operator application to a function. For example, the notation for reduction by the binary function + in APL is +/. This was a natural choice in APL given that it evaluated in a right-to-left manner. In Nial, we made the above choice to support a general left-to-right interpretation rule.

The juxtaposition f g is interpreted as the operation expression denoting operation composition and hence

(f g) A = f (g A) .

Similarly, T U is interpreted as the transformer expression denoting transformer composition and hence

$$(T\ U)\ f\ A = T\ (U\ f)\ A\ .$$

APL does not support compositions directly. In mathematics, the symbol ∘ is often used. John Backus chose this notation for composition in his functional language, FP [Back78]. We have found that the juxtaposition notation for composition works very well.

The juxtaposition A f is interpreted as an operation expression denoting by currying A on the left with f so that

$$(A\ f)\ B = f\ (A\ B)\ .$$

If the requirement to supply parentheses around A f is removed then infix notation for operations is achieved. Thus, A f B is interpreted as (A f) B.

The remaining three juxtapositions are interpreted as transformer expressions by the rules

$$(A\ T)\ f = A\ (T\ f)$$
$$(f\ \ T)\ g = f\ (T\ g)$$
$$(T\ A)\ f = T\ (A\ f)$$

With these interpretations, the general rule for interpreting a sequence of expressions is that strands are gathered first and then the sequence is interpreted in pairs from the left.

An alternative meaning for the juxtaposition f T would have been to have it mean

$$(f\ T)\ g = T\ [f,\ g]$$

in symmetry to the currying rule for A f. However, this choice would make the general rule harder to state and the visual parsing of a general expression much more difficult. Thus, rather than denoting the inner product of linear algebra by

$$+\ \mathrm{INNER}\ *$$

similar to the APL notation , it is denoted by

INNER [+, *]

using the operation list notation described below.

Using the general rule and the juxtaposition rules, the expression

7 2 4 * count 3

is interpreted as follows.

```
7 2 4 * count 3
   =  (7 2 4) * count 3        gathering strand
   = ((7 2 4) *) count 3       left currying
   = (((7 2 4) *) count) 3     forming composition
   = ((7 2 4) *) (count 3)     composition application
   = ((7 2 4) *) (1 2 3)       application of count
   = * (7 2 4)  (1 2 3)        left-curried application rule
   = 14 4 12                   application of *
```

The interpretation rules imply that infix uses of operations are evaluated left to right. For example,

```
3 + 4 * 5
   = (3 +) 4 * 5       left currying
   = + (3 4) * 5       left-curried application rule
   = 7 * 5             application of +
   = (7 *) 5           left currying
   = * (7 5)           left-curried application rule
   = 35                application of *
```

The rule for interpreting array theory expressions does not distinguish between predefined and user defined symbols and identifiers. This feature

of the Nial syntax prevents the establishment of precedence of certain symbols, such as * over +, as is commonly done in procedural programming languages.

The table below summarizes the nine juxtaposition rules for array theory expressions.

A B ... strand *array*	A f left currying *operation*	A T *transformer*
f A op application *array*	f g ... operation composition operation	f T *transformer*
T A *transformer*	T f transformer application *operation*	T U ... transformer composition *transformer*

Bracket-comma lists. The rule for strands provides a means to denote a list of array items of length two or more. However, using strands to denote long lists in which many of the components are themselves complex are difficult to read and understand. The bracket-comma notation was introduced to provide a way of denoting lists for all three orders of semantic objects. The rule is that all the expressions in such a list must be of the same order.

If all the items are array expressions then the construct is an array expression. For example,

```
[1+3, 25, -2]
```

denotes a list of 3 integers (a 1-dimensional array). The notation is extended so that [ ] denotes the empty list, Null and [A] denotes the 1-dimensional array of length one with its item the value of A.

Having two ways to denote lists of arrays is redundant, but it proves useful when building nested lists, since it is visually simpler to use

```
[2 3 4, 5 6]
```

than

[[2, 3, 4], [5, 6]] .

A list of operation expressions, called an **atlas**, is an operation expression, which is interpreted by the rule:

[f, g, …, h] A = [f A, g A, …, h A] .

A list of transformer expressions, called a **galaxy**, is a transformer expression, which is interpreted by the rule:

[T, U, …, V] f = [T f, U f, …, V f] .

The atlas notation is very useful for abbreviating some Nial expressions. It permits many definitions to be written in a variable-free way. For example,

```
average IS OP A { sum A / tally A }
```

can be written as

```
average IS / [sum, tally] .
```

Another common use of an atlas is as the operation argument to a transformer that requires two or more operations. For example,

INNER[+,*] .

Constants in array expressions can be treated as an operation in the equivalent atlas expression by currying it with first since, for any arrays A and B

(A first) B = A .

The inclusion of galaxies in the syntax provides completeness, but is not needed in practical programming.

Other Syntax Topics

Indexing notations. The notations:

A@Address
A#Addresses
A@@Path
A | Slices

allow both selection and insertion of components within an array variable. The first three notations have a counterpart as operations in the theory for both selection and insertion.

The notation for **slice indexing** was added to the language because the corresponding feature of APL notation is heavily used. The Slices component of the notation is a bracket-comma list of length two or more in which one or more of the items is omitted. For example,

A | [,3]

is used to select column with index 3 from a 2-dimensional array A. The missing values are given the fault ?noexpr as the value in the list, and the slice indexing semantics interprets it to mean all the indices in the corresponding position. This choice does lead to some confusion since the case with one comma, [,] means the pair with both items being the fault ?noexpr, whereas the case with no comma, [ ] means the empty list Null.

Conditional notation. Nial has two forms of conditional constructs; the **if-then-else array** expression control construct which selects between array expressions, and the predefined FORK transformer that conditionally applies one of its argument operations depending on the result of applying the first argument. FORK, in the case where it has three parameters, is defined by

```
FORK IS TRANSFORMER f g h OPERATION A
{ IF f A THEN g A ELSE h A ENDIF }
```

It has been suggested that it would be better to have a consistent notation to express conditional constructs for all three orders of functions. However, no agreed upon notation has emerged.

Dot notation. The array theory syntax includes a **dot** notation that changes the order of evaluation so that the operation to the left of the dot takes the entire expression to its right as its argument (up to the next delimiter excluding another dot). The dot notation avoids the use of parentheses to delimit the right argument.

Remarks and comments. Nial uses **remarks** started with the number sign glyph, # in definition files and **comments** started with the percent glyph, % inside definitions. Q'Nial distinguishes the two in that remarks, which can span several lines, are not processed, but comments become part of the internal representation of the definition and show up in a reconstruction of the definition.

This approach uses two glyphs. An alternative would be to treat comments as purely textual information to be discarded and have them terminate at the end of line. For example, Nial could use % or // as a comment symbol. Remarks would be achieved by placing the symbol at the beginning of each line of the remark. Comments could be attached to the end of line in a definition and could include a semicolon. They would be thrown away as remarks are currently.

The change would make the comment notation simpler and closer to that used in other languages. The loss of reconstruction of comments is the only disadvantage. Since most definitions are stored in text files anyway the concept of reconstruction of a definition is primarily an aid in getting definitions into canonical form. An alternative would be to store the text as written as part of the internal representation.

Casts. The cast notation allows a shorthand form to get the parse tree associated with a construct. It can also be obtained by using the composition

parse scan on the corresponding string. If the cast notation were removed it would free up the exclamation mark glyph, ! for another purpose.

The production version of Q'Nial, currently version 6.3, supports a subset of the full syntax described above. The changes were made to make the notation simpler to explain and use.

The main feature removed was the full treatment of transformer expressions. The four transformer expressions formed by the juxtaposition rules: *A T*, *f T*, *T A* and *T U* were removed, as was the galaxy notation. As a result the only form of transformer expressions allowed in a definition are an operation name and a transformer form.

In order not to lose the generality of the expressions that could be used in Nial, the general rule for interpretation was modified. After strands are gathered, the sequence of expressions is considered left to right in pairs. If a pair of juxtaposed expressions does not have a meaning, then the focus is shifted over one position to the right and the rule applied again. When a reduction is made, the result is combined with the expression to the left if possible.

The effect of these combined changes is that, except for galaxies, the same expressions are allowed, but the four juxtaposition transformer expressions cannot be directly named in a definition and cannot be isolated by parentheses. Since there is almost no need to use them in these ways, the change has had little impact on the expressiveness of Nial.

The other major syntax change was the removal of the dot notation. Experience had shown that this construct was confusing to users and often led to programming errors.

CHAPTER 3

Array Theory Choices

This chapter discusses the choices made in the development of **array theory** from basic principles. Trenchard More initiated the study of array theory [More73, More81]. The author and Peter Falster at DTH, influenced by his work, have been interested in understanding the structure of the theory. Jenkins has taught courses on the theory at Queen's and developed several versions of notes concerning the theory. He has also used the equations of the theory in validating the implementation of Q'Nial. Falster has studied the choice of primitives and the sequence of definitions needed to establish all the key array theory concepts.

What is unique about array theory is that it attempts to combine two different organizing principles for data: rectangular arrangement and nested collections. The former is observed in the vectors and matrices used by linear algebra and in tensors used in physics. The latter corresponds to finite sets, nested lists and various forms of hierarchy. They correspond to the two classical ways of looking at ordering. From the point of view of programming languages, the data structures of APL correspond to the first organization and those of Lisp correspond to the second.

Trenchard More has developed a sequence of versions of array theory, each with the goal of combining these two forms of organizing data into a single universal theory [More79]. The difficulty he has faced is that the constraints imposed by the desire to retain properties from both forms of organization have forced choices to be made. The interaction between the possible choices and the resulting impact on equations in the theory has not been well understood. In this chapter, we examine the theory from its basic assumptions in order to understand the constraints and to assess

their impact. We show that the basic assumptions lead to a theory with empty arrays of different dimensionality and that the existence of such empties forces a choice between fundamental properties we would like to have hold.

Basic Assumptions

We begin by stating basic assumptions on what the theory concerns. These are assumptions that Trenchard More either adopted initially, or came to realize were central to the topic. A brief discussion follows some of these assumptions in order to justify their inclusion.

1. The theory is about a single universe of finite data objects. These are what we call **arrays**.

The decision to view the theory as a one-sorted one is driven by the desire to have implicit quantification over the universe in the statement and proofs in the theory.

2. The theory contains functional objects of first and second order called **operations** and **transformers** respectively. Operations map arrays to arrays and transformers map operations to operations.

The decision to restrict to precisely two orders of functions follows the choice made in APL. The major advantage is that the functionality of each object can be determined statically.

3. The theory permits nesting of arrays in a manner analogous to set theory.

This choice makes the theory a theory of collections. This form of nesting corresponds to way lists nest in Lisp.

4. All the data objects are viewed as having dimensionality properties analogous to the treatment of dimensionality in tensor theory.

The dimensionality properties are exactly as they are in APL.

5. The theory is one with **equality**, which means there is a binary relation between pairs of arrays that is transitive, symmetric and reflexive, and which allows for substitution of one array expression for another equal to it in argument positions of operations and predicates of the theory.

In the theoretical work in this manuscript, we use the symbol, = between two array expressions to denote the equality relationship between them. For convenience, we use the same symbol to state equivalence between two operations or between two transformers when discussing them as mathematical objects. The operation that tests equality of arrays in the theory is equal and in Nial the glyph, = also denotes this operation.

6. The theory has primitive data objects called **atoms** or **atomic arrays**. The theory includes integers, real numbers, Boolean values and characters as atoms.

The numeric atomic objects correspond to basic scalars in APL and to numbers in Lisp. Nial has two additional atomic types: **phrases**, used for symbolic names equivalent to symbols in Lisp, and **faults**, which are like phrases but are used to represent error conditions or unusual values.

These six basic assumptions set the stage for a discussion of their implications in the choices available for array theory.

Properties from Set Theory and Dimensionality

We now discuss properties that array theory has based on principles from set theory.

SP1: The concept of **item of an array** corresponds to the concept of *member of a set*.

The term **item** is introduced to be a neutral one so that discussions of sets can be made without confusion. It refers to a relationship between two arrays.

SP2: In an analogy with the *empty set*, there exists an *empty array*, Null that has no items.

The structure of the *Null* is left undetermined at this point.

SP3: If A is an array then there exists an array B such that A is an item of B.

This property corresponds to the concept of a singleton set. In array theory, there can be many arrays B that hold A.

SP4: There exists an operation, called link, that joins arrays together, analogous to *union* in set theory.

SP5: There exists a replacement transformer EACH such that for any operation f, the transformed operation (EACH f) when applied to an array A applies f to each of the items of A, placing the results of the applications in the corresponding locations.

SP6: There exists an operation, called cart, which forms all combinations of items of items of an array, analogous to the *Cartesian product* of set theory.

A major difference between array theory and set theory is that arrays are finite objects, whereas sets allow the description of infinite collections.

We now discuss properties that array theory has based on the geometry of multi-dimensional data objects.

DP1: Every array has **axes**, each of which is of finite extent. The operation valence returns the number of axes of its argument as an integer.

DP2: An array is **rectangular**, i.e. the total number of items is the product of the extents of the axes. The operation, tally, returns the number of items of its argument as an integer.

We call a 1-dimensional array a **list**, and a 2-dimensional array a **table**.

DP3: The dimensionality of an array is described by an array having as its items the extents of the axes. There is an operation, shape that maps a data object to its dimensionality description.

The result returned by shape is left unspecified for a list, but is a list for arrays of zero or two or more dimensions.

DP4: There is an addressing scheme for arrays corresponding to the way subscripts are used for vectors and matrices in linear algebra.

If A is an item of B then there exists a selection operation, pick, and an address I such that selecting in B at I yields A, that is, I pick B = *A* .

The address I specifies a **location** in B that holds A. There may be more than one location in B holding A.

DP5: Numbers in array theory behave like scalars in tensor theory and are considered 0-dimensional arrays.

DP6: There exists a transformer, OUTER that applies an operation between all combinations of items from two arrays forming an array of valence that is

the sum of the valences of the two arrays. The items of the shape of the result are the link of the extents of the two arrays.

The effect of OUTER times is the same as the outer product of tensor theory.

We now study some of the consequences of trying to develop a one-sorted theory that combines the properties stated by the SPn and DPn statements above.

C1: If numbers are treated as scalars, then all atoms in the theory should have dimensionality 0, i.e. they have no axes.

C2: The result of shape applied to an atom is an array that has no items since there are no axes in the atom.

This suggests that the shape of an atom is the array Null. By the assumption on the result of shape, Null is a list. This implies that the operation shape applied to Null results in a list holding the integer 0 as its item.

C3: There are arbitrary arrays with no axes. By using the replacement transformer EACH it is possible to construct an array with no axes that holds a non-atomic item. Arrays with no axes are called **singles**.

C4: The operation cart, drawn from set theory, can be used to define the transformer OUTER, drawn from tensor theory by the equation

OUTER f A = EACH f cart A

provided the dimensionality and extents of cart A are chosen appropriately. In particular, the shape of cart [B, C] has as its items, the items of the shape of B followed by the items of the shape of C.

As a consequence of the properties of cart, there exist multi-dimensional arrays with one or more zeros in the shape. For example, the array formed by

OUTER * [Null, [1, 2, 3, 4, 5]]

is a table with shape

[0, 5] .

We have seen that in order to establish a one-sorted theory of arrays that encompasses the ideas of set theory and tensor theory it is natural to include empty arrays with arbitrary dimensionality and with shapes that include nonzero items. This raises the issue of how empty data structures in the theory should be treated.

We want array theory to be suitable for the processing of textual and symbolic information. We have already postulated the existence of characters as atoms. This suggests that character strings be treated as 1-dimensional arrays as is done in APL, Pascal and C. The existence of the empty string of characters is important for string manipulation. It arises, for example, when the operation sublist applied to a character string selects no characters. Should the empty character string, '' and the *Null* be the same data object? In APL they are different objects, but in array theory the choice has far-reaching consequences.

Desirable Properties for Array Theory

In designing array theory, we are trying to have a theory that has many universal laws. Three desirable laws are:

Law 1: The transformer EACH distributes over operation composition, that is

EACH (f g) A = (EACH f) (EACH g) A

holds for all operations f and g and all arrays A.

Law 2. In general, the operations that do mappings between dimensionality and nesting of arrays all of the same shape should be invertible. If rows is the operation that nests the last axis of an array and *mix* is the operation that pulls up all the axes at the second level appending them to the end, then the equation

mix rows A = A

should hold for all arrays A.

Law 3. The operation link should be an associative operation in array theory, just as *union* is in set theory. The corresponding equation is

link EACH link A = link link A

which should hold for all arrays A.

We show that it is **not** possible to design array theory to have all three of these universal laws. The problems arise precisely due to the richness of the

collection of empty arrays implied by the above assumptions and their consequences.

The equation for Law 1 above states what we call the **distributive law for EACH**. It is a fundamental property we would expect to hold in the theory since EACH is the fundamental transformer that corresponds to the Axiom of Replacement in set theory. We show that it can hold with two very different choices for the analogy to the Axiom of Extensionality in set theory, which states that two sets are equal if an only if they have the same members.

The most direct analogy for the Axiom in array theory is that two arrays are equal if and only if they have the same shape and hold the same items at the same addresses. Using the operation pick, the analogous array theory axiom is stated as:

Axiom 1: For every array A and every array B,

A = B

if and only if

shape A = shape B

and for every address I of A,

I pick A = I pick B.

From Axiom 1 we can state the conditions under which two operations f and g are equivalent.

Theorem 1: Let operations f and g be defined for every array A. Then

f = g

if and only if for every array A,

shape f A = shape g A

and for every for every address I of f A,

I pick f A = I pick f B.

An operation defined for every array A is said to be a ***total*** operation. Using Theorem 1 we can **define** any operation by giving

- the shape of the result of applying the operation to an arbitrary array, and
- the value of I pick on the result for each address I defined for the given shape.

Thus, we can define EACH for a total operation f as follows.

Definition 1: The operation EACH f, for any total operation f, is defined by:
For every array A,

shape EACH f A = shape A,

and for every address I of A,

(I pick) EACH f A = f (I pick) A.

By Axiom 1, we can establish the distributive law for EACH by showing for every array A,

shape EACH (f g) A = shape (EACH f) (EACH g) A

and for every address I of A,

(I pick) EACH (f g) A = (I pick) (EACH f) (EACH g) A.

Both of these results are easily shown from the definition of EACH.

For an empty array, there are no addresses and hence by Axiom 1 all empty arrays of the same shape are equal.

Theorem 2: If arrays A and B are both empty, then

A = B

if and only if

shape A = shape B.

We now show that Theorem 2 implies that there exists an array for which Law 2 does not hold. Consider

B = OUTER * [Null, [1, 2, 3, 4, 5]].

Then

shape B = [0, 5]

and rows B is an empty list. Since by Theorem 2 there is only one empty list

rows B = Null.

Now consider

C = OUTER * [Null, [1, 2, 3, 4]]

which has shape C = [0, 4]. By the same argument

rows C = Null.

If Law 2 holds for every array, then

B = mix rows B
= mix Null
= mix rows C
= C

But the equality of B and C contradicts Theorem 2 since they have different shapes and hence cannot be equal by Axiom 1. Thus, it is not possible to have Law 2 hold for all arrays under the assumption of Axiom 1.

If we want the second desired equation to hold we must allow for the possibility that there are multiple empty lists that hide information. This is not an unreasonable assumption in that we know that APL has made this choice by having the empty string and the empty list of numbers differ.

Let us call the information hidden in an empty array the **virtual item** of the empty array. If we extend the operation pick to select the virtual item when it is applied to an empty array, then the following choice for the analogy for the Axiom of Extensionality allows for multiple empty arrays of the same shape.

Axiom 2: For every array A and every array B,

A = B

if and only if

shape A = shape B

and for every array I,

I pick A = I pick B.

The difference between Axiom1 and Axiom 2 is subtle. Axiom 2 defines the effect of I pick for all arrays instead of just ones that are addresses for shape A.

With Axiom 2 we can now define EACH by

Definition 2: The operation EACH f, for any total operation f, is defined by:
For every array A,

shape EACH f A = shape A,

and for every array I,

(I pick) EACH f A = f (I pick) A.

The argument used with the Axiom 1 to show that the distributive law for *EACH* is obeyed can be carried through in the same manner.

We make the assumption that for any address, the operation pick, applied to an empty array, yields the virtual item.

Axiom 3: For every empty array E, for all arrays I and J,

I pick E = J pick E.

Let us postulate the existence of an operation, void that given an array A returns an empty list that hides the required information about A. Then (I pick) void A returns the virtual item of void A for every value of I.

The operation first is defined by

first A = (0 pick) list A

where list A is the operation that returns the items of an array as a list. By Axiom 3, all uses of pick on an empty list return the virtual item, which implies that the result of first void A is the hidden information. The use of void again should not change the data, so we have

Axiom 4: For every array A,

void first void A = void A.

The composition of operations first void is the mapping done to A when it is hidden as the virtual item of an empty list. Let us call this composition h. Then we have

void h A = void A .

If we apply first to both sides of this equation we get

h h A = h A ,

showing that h is idempotent.

We assume that every empty list is produced by void as stated by:

Axiom 5: If A is an empty list, then there exists an array B such that

A = void B.

Since EACH f void A is an empty list, its hidden data has been mapped by h and hence we know that applying void to its virtual item has no effect. We state this fact as

Theorem 3: For every array A,

void first EACH f void A = EACH f void A.

Using first in place of I pick and void A in place of A in the second part of Definition 2, we have

Theorem 4: For every array A,

first EACH f void A = f first void A.

Now, let us examine the effect of the distributive law for EACH on the information hidden in an empty list. The left side is

first EACH (f g) void A

= (f g) first void A (Theorem 4)

= f g h A (Definition of h)

The right side is

first (EACH f) (EACH g) void A

= f first ((EACH g) void A) (Theorem 4)

= f first (void first (EACH g) void A) (Theorem 3)

= f h first (EACH g) void A (Definition of h)

= f h g first void A (Theorem 4)

= f h g h A (Definition of h)

The equality of the two side says that h must satisfy the equation

f g h A = f h g h A

for every pair of total operations f and g and every array A.

The equality is possible only if h is the identity operation, pass and for every array A,

h A = A .

We have shown that under the assumptions given in Axioms 2 to 5 and Definition 2:

Theorem 5: For every array A,

first void A = A.

Theorem 5 implies that any array can be the hidden data in an empty array. To avoid circularity, we also need to state that there is no path into void A that yields void A. To do this we need to define the operation reach, which uses pick to descend into an array.

Definition 3. For every array A and every array B, if A is empty then

A reach B = B,

otherwise

A reach B = rest A reach (first A pick B).

Axiom 6: For every array A, there does not exist an array B such that

B reach void A = void A.

The introduction of multiple empty lists now allows us to define rows so that if the result is an empty list then the virtual item has the axis being pushed down. We can also define mix so that if it is applied to an empty list of this form, it brings the hidden axis back. In this way, we can achieve Law 2:

mix rows A = A

for all empty arrays A.

We have now shown that if we have multiple empty lists that can have an arbitrary array as the virtual item, then we can have both the first two desired equations. With any other choice for multiple empty arrays the distributive law of EACH is lost.

We now discuss the associativity of link. The property that an operation f is associative is expressed by the law:

For every array A,

f (EACH f) A = f link A.

We demonstrate that this law captures associativity. Let A be the array
[[2 3 4 , 5 6 7] , [10 20 30 , 40 50 60]] .

The left-hand side of the associativity law in this case is

f [f [2 3 4 , 5 6 7], f [10 20 30 , 40 50 60]]
= f [2 3 4 f 5 6 7, 10 20 30 f 40 50 60]
= (2 3 4 f 5 6 7) f (10 20 30 f 40 50 60)

and the right hand side is

f link [[2 3 4 , 5 6 7], [10 20 30 , 40 50 60]]
= f [2 3 4 , 5 6 7 , 10 20 30 , 40 50 60] .

If f is sum then we see that sum is associative on arrays of this form since both sides reduce to 57 79 101.

Since forming unions in set theory is associative it is desirable that the operation link satisfies

link (EACH link) A = link link A .

We can see that for the above non-empty array, the law is satisfied. A geometrical argument can be given to show that it is true for all nonempty arrays.

If there are multiple empty lists due to Axiom 2 then link has to be defined for an arbitrary empty list. Trenchard More has defined it by the equation

link void A = void first A ,

which states that the virtual item of link void A is the first of the virtual item of void A.

Consider the array formed by void [void 3, 8] in the associativity equation for link. The left-hand side is

```
link (EACH link) void [void 3, 8]
      = link void link [void 3, 8]      (Theorem 4)
      = link void [8]                   (property of link)
      = void first [8]                  (definition of link for void)
      = void 8 .                        (property of first)
```

The right hand side is

```
link link void [void 3, 8]
      = link void first [void 3, 8]     (definition of link for void)
      = link void void 3                (property of first)
      = void first void 3               (definition of link for void)
      = void 3 .                        (Theorem 5)
```

Thus, the introduction of multiple empty lists by using Axiom 2 causes the associative law for link to fail for some empty lists.

On the other hand, if Axiom 1 is used and there is only one empty list, Null, the definition of link on an empty list is

```
link Null = Null.
```

The associative law for link holds since both sides reduce to Null.

This section has shown that there is no version of array theory that supports all the desirable properties described at the beginning of this section as three universal laws. A programming language based on array theory is faced with deciding which two of the three laws should hold.

Choices in the Design of Array Theory

The design of Array Theory based on the principles discussed in the section 1 involves choices about which universal laws will hold. The discussion shows that there are at least two theories of arrays that meet the basic assumptions, but each of them fails on one of the three universal laws that are desired.

A fundamental design choice is the selection of the axiom that is the analogy to the Axiom of Extensionality. We have seen that the universal equation stating the distributive law for EACH restricts the choices of the axiom severely. It forces a decision on the number of empty arrays in the theory. With Axiom 1 and Definition 1, there is only one empty array for each shape containing a zero. In this case Law 3, the associative law for link universally holds, but Law 2, in which axis manipulation operations that map axes to nesting, are not universally invertible. With Axioms 2 to 5 and Definition 2 any array can be the hidden data in an empty array and the universality of the two laws is reversed.

The above problems about universality are due to the richness of empty objects in array theory. It is natural to accept that the empty array denoting the shape of an atom, Null is a list. Once Null exists in the theory, the transformer OUTER can be used to construct empty 2-dimensional arrays with shape [0, n], for any value of n. If there is only one empty list in the theory then rows maps all such arrays to Null and hence is not invertible. Thus, there appears to be a need for multiple empty lists.

The introduction of multiple empty lists requires that the effect of all operations has to be defined for empty arrays with arbitrary virtual items. This is a nontrivial task and makes the design of array theory quite complex.

Trenchard More's choice for the definition of link on void A causes the associative law for link to fail. Is there another definition of link void A that behaves as expected in other equations and would allow the associative law to hold? One candidate is

```
link void A = void first main A,
```

where main is the list of nonempty items of A. This definition for link void works for the example discussed above. However, the definition causes another universal equation for link,

```
EACH f link A = link EACH EACH f A
```

to fail. Thus, there does not seem to be any way of avoiding the dilemma.

The desirable properties discussed above are expressed in terms of equations that hold for all arrays. Such equations are called ***universal laws***, in that we want them to hold for all data objects in the universe of the theory. We make statements such as

```
link EACH link A = link link A
```

and assume that it is implicitly quantified over all arrays A. Universal validity is a desirable property for equations to have in that it allows direct symbolic substitution of one side of the equation for the other in an expression. Such symbolic substitutions can be done in forming a proof or in a transformation for improving computational efficiency.

The above discussion about the three desirable laws indicates that some useful laws cannot be universally quantified. In a theory of arrays of the kind we are discussing, we have to either qualify the second Law

```
not empty A =>
        mix rows A = A ,
```

or to qualify the third Law as

```
not empty A =>
        link EACH link A = link link A,
```

where => denotes implication. The requirement to use the equational part of such statements for substitution is that you have to "know" that the condition holds. Thus, such a qualification weakens the law, but it does not prevent the use of the equation in substitutions within contexts where it is known that the array being dealt with is not empty. For most practical work the additional qualification is easily determined.

Because of the constraints imposed by our fundamental assumptions, not all of the desirable laws can be universal in a theory of arrays. The best we can do is to develop a theory in which all data objects are arrays and where there are many equations that are universally true. In such a theory, there are many equations that are true over only a subset of arrays. As a result the development of the theory becomes a design project as well as an exercise in mathematical description.

The development of a theory that has many universal laws is made simpler if we assume that all of the operations predefined in the theory are total operations. The totality of all operations is a very strong requirement for the theory.

Many of the operations we want to have in the theory correspond to mathematical functions that are only partially defined in their natural domain. Consider the division operation, div on real numbers. For div to be total we must define the result of 3.5 div 0.0 . Other mathematical functions are defined for numeric domains but are undefined for literal data. For + to be total we need to decide what the result for 3 + "apple is, where "apple is a phrase.

One way to achieve totality of the predefined operations is to introduce special atomic array values that can be used to define the result in these special cases. In Nial, we call these special values **faults** and use them for extending the domain of operations such as div and +.

A basic philosophical choice in the design of an executable version of array theory is the approach taken to achieve the extension of operations to achieve totality. Is the decision made on an ad-hoc basis for each group of operations, or is there a systematic way that the extension is achieved? Is the emphasis on achieving universality of equations or on providing information about possible errors in the use of operations?

It should be noted that once arbitrary recursive definitions are allowed, one can no longer guarantee totality of all operations since unbounded computations can be introduced. One approach to avoid the need for recursion is to provide transformers that hide recursions internally, yet always produce operations that are known to terminate. It is not clear that such a choice can achieve all the desirable recursive computations.

We assume that the executable array theory we are defining has the property that all the data objects are **arrays**. We call the 1-dimensional arrays ***lists***, and the 2-dimensional arrays **tables**.

Definition of shape of a list. An early choice in the theory is to decide on the semantics of the operation that describes the axis structure of the array. This is usually called shape and has two possible choices for the shape of a list. It can either return the extent as an integer or as a list holding the integer. The choice has little consequence on the overall theory, but does affect how some operations are defined from others.

Addresses. Another early choice is the decision about the addresses to be used to describe locations in the array. In vector and matrix notation subscripts counting from one are conventionally used for the same purpose. In APL, the user is given the choice of using either 1-origin or 0-origin addresses for selection. In array theory 0-origin addresses is preferred because the mapping between the address of a location in a multidimensional array and its index position in the list of items of the array is simpler.

The representation of addresses for a list has a similar choice as the decision about shape above. An address can be an integer or a list holding the integer. The former choice has the advantage that an array of addresses from a list is a simple array. Does the choice of representation for addresses and shapes for lists have to be the same? The answer is no, but it does affect the equations that describe the relationship between shapes and addresses.

Termination of nesting. The way nesting of objects in the theory terminates has to be decided. Since every data object is an array, it is a collection of items that are arrays. An elegant way to achieve this

termination is to state that every atom holds itself as its only item. This is stated by

```
atomic A =>
    first A = A .
```

This interpretation of nesting in array theory is a consequence forced by the combination of treating nesting as analogous to set theory and including atoms as scalars. The operation solitary is the list of length 1 holding its argument, i.e.

```
solitary A = [ A ] ,
```

which implies

```
first solitary A = A
```

for every array A .

We assume that forming the list of all items of a scalar is the list solitary holding the scalar:

```
atomic A =>
    list A = solitary A
```

and that first applied to any array returns the first item in the list of all items of the array:

```
first A = first list A .
```

With these assumptions we see that

```
atomic A =>
    first A
        = first list A
        = first solitary A
        = A .
```

In some APL systems and in J the treatment of integers as scalars does not include self-nesting. In such system the operation corresponding to *single*, usually called *enclose*, is chosen so that

enclose A ≠ A

for every array A. This choice is made to prevent heterogeneous simple arrays from existing. In such a system *enclose* is primitive and there is no operation equivalent to hitch, which constructs heterogeneous simple arrays in array theory.

Arithmetic on atomic arrays. The extension of the arithmetic operations to all atomic domains involves a choice of what to do with arithmetic involving literal data. For example, 3 + "apple could return a fault, the number 3 or the phrase "apple. In spreadsheets it is common to ignore the literal data in arithmetic operations. What universal laws do we want to have hold for arithmetic operations?

Arithmetic on lists. In linear algebra, the addition of two vectors of the same length produces a vector of the same length. In order to have the analogy hold in array theory, addition is extended to two arrays of numbers of the same shape. If we attempt to add together two lists of different lengths, what should the result be? In APL 1-element lists are extended to the length of the other list, but adding lists of length 2 and 3 respectively produces a LENGTH ERROR and the computation is interrupted. In Q'Nial, as implemented in Version 4.1, an elaborate algorithm is applied to produce arrays of the same shape (dimensionality and extents match) that are then added item by item. Version 6.21 gives the fault ?conform instead. Should arithmetic operations be extended to reshape their arguments to the same shape, and if so, by what algorithm?

Pervasive arithmetic. The distribution of arithmetic operations across arrays of the same shape described above can be extended to arrays of the same structure with the operation being applied at the leaves. This

extension leads to pervasive operations. Should the extension to pervasive operations be made? To which operations should it apply?

Extension of pick. The operation pick takes a pair, I A as its argument. The operation is defined if I is an address of A. How should pick be extended to a total operation? If the argument is not a pair, then a fault can be given or the operation 2 reshape can be applied to the argument. If the argument is a pair but I is not an address of A, then a choice has to be made. However, the choice is restricted if the choice for the analogy to the Axiom of Extensionality is Axiom 2. In this case the equation

```
I pick EACH f A = f (I pick A)
```

forces the choice to be the virtual item if A is empty, or to be some item of *A* if *A* is non-empty. In the latter case, the selected item could always be the first item, or an item chosen by coercing I to be in range by modular arithmetic on the shape of A.

If Axiom 1 is chosen, then the fault, ?address can be given for all argument pairs to pick in which I is not an address. Using a fault helps catch careless programming mistakes. For pragmatic reasons, I should be coerced to the structure of an address before testing whether it is in range so that either an integer or the list of an integer can serve as an address to a list.

We have now completed our description of the choices needed to define an array-based programming language using an array theoretic mode of data. Ultimately, the choices are made on a mixture of practical grounds based on their impact on ease of programming, and on considerations of the elegance of the mathematical system. Trenchard More and I struggled with these choices over many years and eventually agreed to disagree.

References

[Backus78]
Backus J., “Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Program”, *Comm. ACM* **21**(8), 613-641 1978.

[Fran84]
Franksen, O. I., “Are Data-Structures Geometrical Objects?”, *Sys. Anal. Model. Simul*. **1** 1984.

I: Invoking the Erlanger Program, 113-130
II: Invariant Forms in APL and Beyond, 131-150
III: Appendix A. Linear Differential Operators, 251-260
IV: Appendix B: Logic Invariants by Finite Truth-Tables, 339-350

[Fran85]
Franksen, O. I., “The Nature of Data - From Measurement to Systems”, *BIT* **25** 24-50 1985.

[Fran96]
Franksen, O. I., “Invariance under Nesting - An Aspect of Array-based Logic with relation to Grassman and Pierce”, *Hermann Gunther Grassman (1809-1877): Visionary Mathematician, Scientist and Neohumanist Scholar*, G. Schurbring (ed.), 303-335, Kluwer, Amsterdam 1996.

[FrJe92]
Franksen, O. I., Jenkins M. A. “On Axis Restructuring Operations for Nested Arrays”, Second International Workshop on Array Structures ATABLE-92, Montreal, 1992.

[GlJe89]
Glasgow J. I., Jenkins M. A., “A Logical Basis for Nested Array Data Structures”, *Computer Languages*, **14**(1), 35-51 1989.

[GuJe79]
Gull W. E., Jenkins, M. A. “Recursive data structures in APL”, *Comm. ACM* **22**(1), 79-96 (1979)

[JeJe85]
Jenkins, M.A., Jenkins, W.H., *Q’Nial Reference Manual*, NIAL Systems Limited, Kingston, Canada. 1985.

[JeGl86]
Jenkins, M. A., Glasgow, J. I., McCrosky, C. D., “Programming Styles in Nial”, *IEEE Software*, January 1986.

[JeMu91]
Jenkins, M. A., Mullin, L. M. R., “A Comparison of Array Theory and a Mathemtics of Arrays”, *Arrays, Functional Languages, and Parallel Systems*, 237-267 Kluwer, Boston 1991.

[Jenk85]
Jenkins, M.A., *The Nial Dictionary*, NIAL Systems Limited, Kingston, Canada. 1985.

[Jenk07]
Jenkins, M.A., "Programming with Arrays - Lessons from Nial", Proceedings of APL 2007, Montreal, October 2007.

[More73]
More T., “Axioms and theorems for a theory of arrays”, *IBM J. Res. Dev.* **17**(2), 135-175 1973.

[More79]
More T., “The nested rectangular array as a model of data”, *Proceedings of APL 79, APL Quote Quad* **9**(4) 55-73 1979.

[More81]
More T. “Notes on the diagrams, logic and operations of array theory”, *Structures and Operations in Engineering and Management Systems* (edited by Bjorke O. and Franksen O.), 497-666 Tapir Publishers, Trondheim, Norway 1981.

[More93]
More T., “Transfinite Nesting in Array-Theoretic Figures, Changes, Rigs and Arms. Part 1”, *APL Quote Quad* **24**(1) 170-184 1993.

[More06]
More T., "The Birth of Array Theory and Nial", Proceedings of "Dartmouth Artificial Intelligence Conference: The Next Fifty Years", July 2006.