Array Theory and Nial

Mike Jenkins
Peter Falster

August 1999

The research reported here was supported by the National Bank of Denmark, the Technical University of Denmark, The Natural Sciences and Engineering Research Council of Canada, and by NIAL Systems Limited.

Preface

This report is the result of collaboration between the authors during the first 8 months of 1999 when Mike Jenkins was a Visiting Professor at DTU. The goal of Jenkins visit was first, to develop a tool for the investigation of array theory concepts, and second, to document the array theory used in the current version of Q'Nial for future reference.

The materials gathered here were originally developed as discussion papers and notes of experiments with the above-mentioned tool. They have been organized into a manuscript in order to present a state-of-the-art snapshot of our current research on array theory topics.

The authors are indebted to Trenchard More, Ole Franksen and Jean Michel for their conversations over many years that have contributed to the ideas presented here, and to the many people who have worked with Mike Jenkins on the development and refinement of Q'Nial.

Table of Contents

Chapter 1 Introduction

The programming language Nial was designed jointly by Mike Jenkins and Trenchard More in the early 1980's. It was based on concepts from other programming languages, mostly from APL, Lisp and ALGOL, and on the mathematics of Array Theory that More had developed at IBM [More73, More79, More81]. The goal of the design was to provide a tool for very high level problem solving in such diverse fields as business, science, engineering and artificial intelligence. The design assumed that the language would be used interactively to explore problem domains and that problems would be solved incrementally within a workspace containing both data and programs [JeGl86].

A portable interpreter, Q'Nial was built at Queen's University, Canada under Jenkins' leadership. The first version of the interpreter was written in C and ran only under UNIX. IBM supported the development of versions for the then fledgling IBM Personal Computer and the IBM mainframe computer. NIAL Systems Limited was formed to market Q'Nial on behalf of the University.

During the early period of the development of Nial, the language and its supporting array theory were refined to make it simpler and small enough to fit within the constraints of the IBMPC. By 1983, the first commercial version of Q'Nial was released. As experience with the use of the language was gained, the programming language and its implementation were modified to give better functionality and to improve efficiency. This was embodied in the version 4.1 system that was distributed quite widely.

The language remained largely unchanged until 1995, when Jenkins released version 6.1 of the Q'Nial for commercial use. In this version, Jenkins provided a simpler language and one with a variant of Array Theory as its underlying mathematics. In addition, the implementation techniques for many of the built-in functions had been improved by using state-of-the-art algorithms. The work on this version began during a sabbatical spent at the Technical University of Denmark (DTU) in 1991/92 and continued at Queen's University. In 1995, Jenkins took early retirement from the university to work exclusively with NIAL Systems Limited.

The purpose of this manuscript is to provide more details about the design of Nial and its relationship to Array Theory to the interested reader. It is based on research conducted while Jenkins was visiting DTU to work with Peter Falster and Ole Franksen in 1999. The main goal of the visit was to develop a tool that could be used by students and researchers in exploring Array Theory concepts. Jenkins has constructed the V4 Shell, described in Chapter 7, to meet the goal and several of the chapters describe work accomplished using the tool.

The Manual for Q'Nial and a student version are available at *http://www.nial.com*. The reader who wishes to learn more about the use of Nial and array theory is encouraged to read Parts 1 to 3 of Volume 1 of the Manual [JeJe98] and to experiment with the free version of Q'Nial. The V4 Shell Tool and the folders described in the manuscript are also available from the web site.

Chapter 2 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] and in his forthcoming book.

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 recently 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

```
ÚÄÂÄÄÂÄÄÂÄÄ¿
³1³ 2³ 3³ 4³
ÃÄÅÄÄÅÄÄÅÄÄ´
³5³ 6³ 7³ 8³
ÃÄÅÄÄÅÄÄÅÄÄ´
³9³10³11³12³
ÀÄÁÄÄÁÄÄÁÄÄÙ
```

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

```
ÚÄÂÄ¿
³3³4³
ÀÄÁÄÙ
```

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

```
ÚÄÂÄÄÂÄÄÂÄÄ¿ ÚÄÄÂÄÄÂÄÄÂÄÄ¿
³1³ 2³ 3³ 4³ ³13³14³15³16³
ÃÄÅÄÄÅÄÄÅÄÄ´ ÃÄÄÅÄÄÅÄÄÅÄÄ´
³5³ 6³ 7³ 8³ ³17³18³19³20³
ÃÄÅÄÄÅÄÄÅÄÄ´ ÃÄÄÅÄÄÅÄÄÅÄÄ´
³9³10³11³12³ ³21³22³23³24³
ÀÄÁÄÄÁÄÄÁÄÄÙ ÀÄÄÁÄÄÁÄÄÁÄÄÙ
```

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,

```
ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄ¿
³ÚÄÂÄÂÄÄÄ¿  ³ÚÄÄÄÂÄÂÄÄ¿  ³ÚÄÂÄÄÄÂÄÂÄÄÄ¿ ³
³³1³2³3³4³  ³³5³6³7³8³  ³³9³10³11³12³ ³
³ÀÄÁÄÄÁÄÄÙ  ³ÀÄÄÄÁÄÄÁÄÄÙ  ³ÀÄÁÄÄÄÁÄÄÁÄÄÙ ³
ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄ´
³ÚÄÄÂÄÄÂÄÄÂÄÄÂÄÄ¿³ÚÄÄÂÄÄÂÄÄÂÄÄÂÄÄ¿³ÚÄÄÂÄÄÂÄÄÂÄÄÂÄÄ¿³
³³13³14³15³16³³³17³18³19³20³³³21³22³23³24³³
³ÀÄÄÁÄÄÁÄÄÁÄÄÙ³ÀÄÄÁÄÄÁÄÄÁÄÄÙ³ÀÄÄÁÄÄÁÄÄÁÄÄÙ³
ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

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 and the 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,

```
ÚÄÄÄÂÄÄÄÄÂÄÄÄÄÄÂÄÄÄÄÄÄÂÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄ¿
³ÚÄ¿³ÚÄÂÄ¿³ÚÄÂÄÂÄ¿³ÚÄÂÄÂÄÂÄ¿³ÚÄÂÄÂÄÂÄÂÄ¿³
³³1³³³1³2³³³1³2³3³³³1³2³3³4³³³1³2³3³4³5³³
³ÀÄÙ³ÀÄÁÄÙ³ÀÄÁÄÁÄÙ³ÀÄÁÄÁÄÁÄÙ³ÀÄÁÄÁÄÁÄÁÄÙ³
ÀÄÄÄÁÄÄÄÄÁÄÄÄÄÄÁÄÄÄÄÄÄÁÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÙ
```

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

```
oÄÄÄÄÄÄÄ¿
³ÚÄÂÄÂÄ¿³
³³1³2³3³³
³ÀÄÁÄÁÄÙ³
ÀÄÄÄÄÄÄÄÙ
```

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

```
   oÄ¿
3 = ³3³
```

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. This is shown in the formal development of V6 Array Theory presented in Chapter 6.

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-contained singles.

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 Nial, designed jointly by More and Jenkins, there are multiple empty lists. In Version 6.21 of Nial, Jenkins has changed to an array theory that has only one empty list, resulting in the equation

$$Null = \text{''}.$$

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 4 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,** 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
```

```
ÚÄÄÂÄÄÂÄÄ¿
³ 1³ 2³ 3³
ÃÄÄÅÄÄÅÄÄ´
³ 4³ 5³ 6³
ÃÄÄÅÄÄÅÄÄ´
³ 7³ 8³ 9³
ÃÄÄÅÄÄÅÄÄ´
³10³11³12³
ÀÄÄÁÄÄÁÄÄÙ
```

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

```
     shape A
```

```
ÚÄÂÄ¿
³4³3³
ÀÄÁÄÙ
```

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. The formal development of some of the equations is given in Chapter 6.

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 it 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
```

```
ÚÄÄÂÄÄÂÄÄ¿
³ 6³ 7³ 8³
ÃÄÄÅÄÄÅÄÄ´
```

³ 9³10³11³
ÃÄÄÅÄÄÅÄÄ´
³12³13³14³
ÃÄÄÅÄÄÅÄÄ´
³15³16³17³
ÀÄÄÁÄÄÁÄÄÙ

Transformers are monadic functions that map an operation to an operation. In Chapter 3 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 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 two mechanisms:

bracket-comma notation [A, B, C]

or, a **strand** 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'
oÄÄÄÄÄÄÄÄÄÄÄÄ¿
³hello world³
```

ÀÄÄÄÄÄÄÄÄÄÄÄÄÙ

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 5.5]
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'
ÚÄÄÄÂÄÄÄÂÄÄÄ¿
³abc³abcd³abcde³
ÀÄÄÄÁÄÄÄÁÄÄÄÙ
```

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 of the items given in *Values* in the array *A*. Consider the example

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

The above examples illustrate the most commonly used syntax for writing array theory expressions. The general rules for forming them are given in Chapter 3.

Chapter 3 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 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.

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

Symbols. These are single glyphs or pairs 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 sign	<	lt
greater than sign	>	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.

This decision was not made since it makes 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 (except *colon*, which, 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(s)**	**Usage**
at symbol	@	at indexing
	@@	path indexing
number sign	#	at-all indexing, remarks
vertical bar	|	slice indexing
exclamation mark	!	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 *dash* to denote negative numbers, the *period* 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 and the V4 Shell.

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 the notations for a phrase and a fault are omitted in the value. There are also operations that convert strings to phrases and faults. Thus, *“apple* is equivalent to *phrase ‘apple’*.

There are problems with all of these choices. 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.

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*.

Program text may be entered directly at the prompt or by using the *loaddefs* mechanism 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. It is useful if you want to load 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 are introduced as array expressions and are given interpretations for the value returned. In general it is the value of the last expression evaluated, but if no value is available a special fault value, *?noexpr* is the implicit value given. This 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 "{" and "}".

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 it is recursive then the replacement is repeated dynamically 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.

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, e.g. +/. We have 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

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

APL does not support compositions directly. In mathematics, the symbol ° is often used. Backus chose this notation for 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

*+ INNER **

similar to the APL notation , it is denoted by

*INNER [+, *]*

using the operation list notation discussed below.

Using the general rule and the juxtaposition rules, the expression *7 2 * opp 5* is interpreted as follows.

*7 2 * opp 5*
*= (7 2) * opp 5*
*= ((7 2) *) opp 5*
*= (((7 2) *) opp) 5*
*= ((7 2) *)(opp 5)*
*= ((7 2) *) -5*
*= * ((7 2) -5)*
= -35 -10

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

*3 + 4 * 5*
*= (3 +) 4 * 5*
*= + (3 4) * 5*
*= 7 * 5*
*= (7 *) 5*
*= * (7 5)*
= 35

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	*A f*	*A T*
strand	**currying**	
array	operation	transformer

f A	*f g*	*f T*
op application	**op composition**	
array	operation	transformer
T A	*T f*	*T U*
	tr application	**tr composition**
transformer	operation	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 functions. 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 *Null* and *[A]* denotes the 1-dimensional array of length one with its item the value of *A*.

Having two ways to denote lists 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 examples,

average IS OP A { sum A / tally A }

can be written as

average IS / [sum, tally]

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 an array expression and in order to get the expected meaning from *A|[,3]* an explicit interpretation for the array list notation with missing values is given. The fault *?noexpr* is the value given to the missing values, and in slice indexing it is interpreted 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 *?noexpr*, whereas the case with no comma: *[ ]* means the empty list *Null*.

Conditional notation. Nial has two forms of conditional constructs; the if-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* 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.

This 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 V6 subset of Nial syntax

The production version of Q'Nial, currently version 6.21, 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 4 Array Theory Choices

1. Introduction

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 authors, 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 has 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.

1.1 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. The operation that tests equality of arrays in the theory is *equal* and in Nial the symbol, = 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 atomic objects correspond to basic scalars in APL and to numbers and symbols in Lisp.

1.2 Nesting properties analogous to set theory

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

1. 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.

2. In 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.

3. If *A* is an array then there exists an array *B* such that *A* is an item of *B*.

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

4. 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.
5. There exists an operation, called *link*, that joins arrays together, analogous to *union* in set theory.

6. 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.

1.3 Properties from dimensionality

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

1. 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.

2. 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*.

3. The dimensionality of an array is described by an array having as its items the extents of the axes. There is an operation, called *shape*, which 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.

4. 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,

$$I \text{ pick } B = A.$$

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

5. Numbers behave like scalars in tensor theory and are considered 0-dimensional arrays.

6. 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.

1.4 Simple consequences from combining dimensionality with nesting

We now study some of the consequences of trying to develop a one-sorted theory that combines the concepts of the two previous sections.

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

2. 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 an array holding the integer *0* as its item.

3. 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*.

4. 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*.

5. As a consequence of the properties of *cart*, there exist multidimensional 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]*.

1.5 Summary

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.

2. Desirable properties for the theory

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

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.

2. In general the operations that do mappings between dimensionality and nesting of equishaped arrays 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.

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 difficult to design an *array theory* to have all three of these universal laws. The problems arise precisely due to the richness of the collection of empty arrays.

2.1 The distributive law for EACH

The first equation 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 of the Axiom for 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 *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 address *I* of *f A*,

$$I\ pick\ f\ A = I\ pick\ g\ A.$$

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

(I pick) EACH (f g) A = (I pick) (EACH f) (EACH g) A, for every address *I* of *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 the equation

$$mix\ rows\ A = A$$

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 there is only one empty list

$$rows\ B = Null.$$

Now consider

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

with

$$shape\ C = [0, 4].$$

By the same argument

$$rows\ C = Null.$$

If the desired equation,

$$mix\ rows\ A = A$$

holds for every array *A*, then

$$B = mix\ rows\ B = mix\ Null = mix\ rows\ C = C.$$

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

2.2 Multiple empty lists

If we want the second desired equation to hold we must allow for the possibility that there are multiple empty lists that hide information. 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 *B*,

$$A = B$$

if and only if

$$shape\ A = shape\ B$$

and for every array I,

$$I\ pick\ A = I\ pick\ B.$$

If we 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 and every array A,

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

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 an array that is the information hidden in the empty array.

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.

Since the operation *first* is defined by $(0\ pick)\ list$ and by Axiom 3 all uses of *pick* on an empty list return the virtual item, 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. Hence

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* (Defn 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* (Defn of *h*)
= *f h g first void A* (Theorem 4)
= *f h g h A* (Defn 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

$$A \text{ reach } B = B,$$

otherwise

$$A \text{ reach } B = \text{rest } A \text{ reach (first } A \text{ pick } B).$$

Axiom 6: There does not exist an array B such that

$$B \text{ reach void } A = \text{void } A.$$

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

$$\text{mix rows } A = A$$

for all empty arrays.

This section has 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.

2.3 Associativity of link

The property that an operation *f* is associative is expressed by the law:

$$\text{For every array } A, \text{ f } (EACH\ f)\ A = f \text{ link } A.$$

Let A be the array $[\ [\ 2\ 3\ 4\ ,\ 5\ 6\ 7]\ ,\ [\ 10\ 20\ 30\ ,\ 40\ 50\ 60\ ]\ ]$. The left-hand side is

$$\begin{aligned} & f\ [\ \ f\ [2\ 3\ 4\ ,\ 5\ 6\ 7\ ],\ \ \ f\ [\ 10\ 20\ 30\ ,\ 40\ 50\ 60\ ]\ \ ] \\ & = f\ [\ 2\ 3\ 4\ f\ 5\ 6\ 7,\ 10\ 20\ 30\ f\ 40\ 50\ 60\ ] \\ & = (2\ 3\ 4\ f\ 5\ 6\ 7)\ f\ \ (10\ 20\ 30\ f\ 40\ 50\ 60) \end{aligned}$$

and the right hand side is

$$\begin{aligned} & f \text{ link } [\ [2\ 3\ 4\ ,\ 5\ 6\ 7\ ],\ [10\ 20\ 30\ ,\ 40\ 50\ 60\ ]\ ] \\ & \qquad = f\ [2\ 3\ 4\ ,\ 5\ 6\ 7\ ,\ 10\ 20\ 30\ ,\ 40\ 50\ 60\ ]\ . \end{aligned}$$

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]
= link void [8]
= void first [8]
= void 8 .

The right hand side is

link link void [void 3, 8]
= link void first [void 3, 8]
= link void void 3
= void first void 3
= void 3 .

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.

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

2.4 First conclusions

The design of an 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 the associative law for *link* universally holds, but 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.

3. Universal theory and total operations

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 restate the second one as

not *empty A* implies *mix rows A = A* ,

or to restate the third one as

not *empty A* implies *link EACH link A = link link A* .

While such a qualification weakens the law, 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 will be many equations that are true over only a subset of arrays.

3.1 Extending operations to totality

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 operation *divide* on real numbers. For *divide* to be total we must define the result of *divide [3.0,0.0]*. Other mathematical functions are defined for numeric domains but are undefined for literal data. For *plus* to be total we need to decide the result of *plus[3,"apple]* , 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 *divide* and *plus*.

A basic philosophical choice in the design of a 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 could 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.

4. Other choices in defining an array theory

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

1. *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.

2. *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 are 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.

3. *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

$$\textit{atomic } A \text{ implies } \textit{first } A = A \ .$$

This interpretation appears to be a consequence forced by the combination of treating nesting as analogous to set theory and including atoms as scalars. If we accept that

$$\textit{atomic } A \text{ implies } \textit{list } A = \textit{solitary } A \ ,$$

then the result is immediate from the equations

$$\textit{first } A = \textit{first list } A$$

and

$$\textit{first solitary } A = A \ .$$

From the equations

$$\textit{shape } A \textit{ reshape (first } A \textit{ hitch rest } A) = A$$
$$\textit{first } A = \textit{first list } A$$
$$\textit{rest } A = \textit{rest list } A$$
$$A \textit{ hitch void } B = A \textit{ hitch void } C$$

and the definition

$$\textit{single } A = \textit{Null reshape (} A \textit{ hitch Null)}$$

we can also show that

$$\textit{atomic } A \text{ implies } \textit{single } A = A \ .$$

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

$$\textit{enclose } A \neq A$$

for any array. This is done 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.

4. *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 plus "apple* could return a fault, the number *3* or the phrase *"apple*. In spreadsheets it is common to ignore the literal data. What universal laws do we want to have hold for arithmetic operations?

5. *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 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?

6. *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?

7. *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 *pair* 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 for the Axiom of Extensionality permits 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 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.

5. Constructing a definition sequence

In describing a design for array theory, we select a small number of primitive operations and transformers and a set of mechanisms for making new definitions. We then define the remainder of the operations and transformers we wish to consider from this initial set. There are several possible goals for such a *definition sequence*. Three of these are:

1. To establish the soundness of the theory in a logic sense. This can be achieved by providing axioms that express the properties of the arrays, operations, and transformers and demonstrating that there exists a model for the theory in which the axioms hold. Once this is done, additional properties of the arrays, operations and transformers can be expressed as theorems that can be proven from the axioms. A subgoal might be to express as many of the axioms and theorems as equations that hold universally.

2. To develop the definitions in order to show the relationships that exist between the key arrays, operations, and transformers of the theory. The goal here could be pedagogical rather than a formal mathematical development. Provided the primitives chosen and the properties given to them can be defined by a development of the first kind, the resulting operations and deduced properties for them are soundly defined.

3. To demonstrate how part of the theory can be developed from a small set of primitives. For example, the definition of the operations that are related to the structural properties of the data objects of the theory could be developed, leaving the operations related to the substance of the atomic data defined only to the extent needed to define the structural operations.

If the goal is to establish soundness as in 1), we could seek to find the fewest number of primitives that are required, or a set of primitives that require the least number of axioms.

If the goal is the elucidation of the theory as in 2) then the primitives could be chosen to capture the essential assumptions on which the theory is founded.

A description with goal 3) could choose a set of primitives that permits the construction of some of the predefined operations using only simple definition mechanisms.

The construction of a definition sequence depends on the rules for forming new definitions from primitives and/or objects defined earlier in the sequence. We can allow definitions to be made by equations, by using lambda forms like the Nial operation form and transformer form constructs, or by using variable free definitions that use only the functional notation of Nial. Other possible rules are to allow the inclusion of recursive definitions and the use of conditional constructs.

Chapters 6, 8 and 10 concern the construction of definition sequences. These have all been developed with the help of the V4 Shell tool described in Chapter 7.

Chapter 5 Recursion in Array Theory and Nial

Array theory is a theory about recursive data structures that have both rectangularity and nesting properties. The theory is recursive in that items of arrays are themselves arrays. The recursion of the data structures in depth ends at the atomic arrays that hold themselves. The one-dimensional arrays of the theory, called *lists*, are viewed as vectors with indices, but are given many recursive properties analogous to head-tail lists in functional languages. As a result of the use of recursion in these two ways, many of the properties of arrays are described by recursive equations and many of the predefined functions are given recursive definitions.

This chapter is concerned with the identification of patterns in the use of recursion so that only a small number of recursive mechanisms are needed to express the definitions of the theory. The goal of finding a small number of useful recursive transformers serves two purposes. First, it shows the structure of the theory by having similar operations defined in the same manner. Second, it provides a framework for achieving effective implementation of many operations from only a small initial core.

Recursion and iteration are closely related concepts. Both allow a computation to be repeated a number of times. In recursion, the repetition is controlled by the function calling mechanism; in iteration, the repetition is controlled by an explicit loop construct. In recursion, the temporary values generated during the repetition are stored in the stack associated with the function calling mechanism; in iteration, they are stored in local variables, or in an explicitly managed stack. A recursive definition has the advantage of brevity, but its iterative equivalent may be more efficient since the overhead of function calls is removed.

The comparison between recursive and iterative solutions in Nial is complicated by the fact that Q'Nial is an interpreter, which means its execution of loops and function calls is done indirectly by the mutually recursive internal C routines *eval*, *apply* and *apply_transform*. During the execution of a recursive definition, multiple calls to the C routines occur each time that the definition recurs at the Nial level. As a result, C stack space is consumed rapidly when a recursive definition in Nial is executed using function applications.

On the other hand, if the equivalent solution is achieved as an iteration, then the execution can be carried out without excessive use of C stack space. (In both cases the same amount of Nial heap space is consumed to hold temporary values.) By avoiding the excessive use of the C stack, much larger problems can be solved recursively.

The task of recognizing arbitrary recursions in Nial definitions and converting them automatically to loop executions is a difficult one. Instead of attempting such conversions, we provide three built-in transformers that capture important classes of recursive computations. We define the transformers formally in Nial as recursions and then provided alternative definitions for them that achieve the same effect with loops. The internal implementations of the transformers are based on the loop versions. Thus, by

using the transformers, effective implementation of some definitions that are thought of as recursions is achieved by the use of the internal iterations.

General recursion

Recursive definitions can be written in many different styles. However, all of them depend on three capabilities: a test to terminate the recursion, a recursive call on a reduced problem, and a means to synthesize the result. The latter step requires a value to begin the synthesis when the end condition is met, a value associated with the argument, and rule for combining the values as the recursion unwinds. By having each of these capabilities achieved by an operation we can define a general transformer that captures many recursive definitions.

We define a general recursion transformer, *RECUR*, with five operation arguments:
test checks that the argument meets an end condition,
endf is applied to the end argument before starting to build the result,
parta left value computed from the argument and stacked,
joinf combines the left and right values as the recursion unwinds, and
partb gives the value to be recurred on to produce the right value.

The recursion terminates provided repeated application of the operation *partb* results in an array that satisfies *test*. The recursive definition is:

```
RECUR IS TRANSFORMER test endf parta joinf partb
( FORK [test, endf, joinf [parta, RECUR [test, endf, parta, joinf, partb] partb] ] )
```

The corresponding loop definition is:

```
RECUR IS TRANSFORMER test endf parta joinf partb OPERATION A {
    Elements := Null;
    WHILE not test A DO
        Elements := Elements append parta A;
        A := partb A;
    ENDWHILE;
    Res := endf A;
    FOR E WITH reverse Elements DO
        Res := E joinf Res
    ENDFOR }
```

The transformer *RECUR* can be used to achieve several examples of recursive definitions. A definition of the factorial function is given by:

```
factorial IS RECUR [ 0 =, 1 first, pass, product, -1 +] .
```

A second example is to define FOLD using RECUR:

```
FOLD IS TRANSFORMER f
```

```
(RECUR [0 = first, second, pass, f second, [-1 + first, second] ] pair ) .
```

A third example is a definition of a summation of the items of an array:

```
dosum IS RECUR [ empty, 0 first, first, plus, rest ] .
```

Recursions over the length of an array

The definition of *dosum* is an example of a recursion that is using the head-tail (here *first-rest*) form of recursion to walk over the top level of an array. This is a recursion over the length of the list of items of an array. Another example of a recursion over length is using *RECUR* to achieve a definition of the transformer *EACH* given by:

```
EACH IS TRANSFORMER f
    ( reshape [shape, RECUR [ empty, void f first, f first, hitch, rest] list ] ) .
```

Recursive definitions like these are useful for describing meaning, but are computationally inefficient in Q'Nial. The inefficiency arises because arrays are stored as large containers with contiguous elements and the implementations of *rest* and *hitch* use copying of partial arrays. The definitions of *dosum* and *EACH*, which are recursions across the length of the array, can be made special cases of a transformer *ACROSS* that knows that the test is for an empty array and that the recursion is done on the rest of the list. The definition is:

```
ACROSS IS TRANSFORMER endf parta joinf
     ( RECUR [empty, endf, parta first, joinf, rest ] list )
```

Then the definitions become:

```
dosum IS ACROSS [ 0 first, pass, plus ]
```

and

```
EACH IS TRANSFORMER f ( reshape [ shape, ACROSS [ void f first, f, hitch ] ] )
```

Since *ACROSS* is defined in terms of *RECUR* its execution is achieved with loops. However, a much simpler loop execution is possible since all the items are available. It is defined by:

```
ACROSS IS TRANSFORMER endf parta joinf OPERATION A {
    A := list A;
    Res := endf void last A;
    FOR I WITH reverse tell tally A DO
        Res := parta A@I joinf Res
    ENDFOR }
```

With this definition of *ACROSS* as the basis for its implementation in C, the execution of *dosum* is very efficient. Thus, the recursive definition of reductions such *sum*, *product*, *max*, *min*, *and* and *or* for simple arrays can be achieved effectively by using *ACROSS*. It should be noted that using *ACROSS* to achieve *EACH* still involves the use of *hitch* and hence a direct implementation of *EACH* is still desirable to avoid unnecessary internal copying of partial arrays.

A definition of *reverse* can also be attained.

reverse IS reshape [shape, ACROSS [pass, pass, CONVERSE append]]

ACROSS can also be used directly to achieve a definition of *FOLD*:

```
FOLD IS TRANSFORMER f OPERATION N A
    { ACROSS [A first, pass, f second] tell N }
```

Recursions over depth

The fundamental recursive property of nested arrays is that the items of arrays are themselves arrays. In order to define operations that use the arrays that are nested at some arbitrary level below the top level of the array, a recursion in depth is required. For example, the operation *opposite* is defined by:

opposite IS FORK [atomic, 0 minus, EACH opposite]

The use of the *EACH transform* of the operation being defined achieves the recursion in depth. We can define all the unary pervasive operations in a similar fashion. Another example of a depth recursion is in defining the pervasive operation *sum* by

sum IS FORK [simple, dosum, EACH sum pack] ,

where we have used the operation *dosum* defined above. Notice that in this example the *EACH sum* transform is applied after processing the argument with *pack*.

Another example is a definition of *depth* by:

depth IS FORK [atomic, 0 first, 1 plus max EACH depth]

In this last example, the result of the *EACH depth* transform is processed as the recursion unwinds.

The following general transformer DOWN that recurs over the depth of an array to some arbitrary level can achieve all of these examples. *DOWN* has four operation arguments:

test	tests when the recursion has gone as deep as necessary,
endf	is applied to the argument that satisfies *test*,
structf	rearranges the argument before recurring on each item, and

joinf combines the results of the recursion on the items.

DOWN terminates as long as each path into the array eventually satisfies *test*.

```
DOWN IS TRANSFORMER test endf structf joinf
    ( FORK [test, endf, joinf EACH DOWN [test, endf, structf, joinf] structf ] )
```

The corresponding loop definition is:

```
DOWN IS TRANSFORMER test endf structf joinf OPERATION A {
  Candidates := [A];
  Results := Null;
  WHILE not empty Candidates DO
    Candidates B := [front, last] Candidates;
    IF B = "Start THEN
      Candidates Shp := [front, last] Candidates;
      N := prod Shp;
      IF N = 0 THEN N := N + 1; ENDIF;
      Results Items := opposite N [drop, take] Results;
      Results := Results append joinf (Shp reshape Items);
    ELSEIF test B THEN
      Results := results append endf B;
    ELSE
      B := structf B;
      Candidates := Candidates link [shape B, "Start] link reverse list B;
     IF empty B THEN Candidates := Candidates append first B; ENDIF;
ENDIF;
  ENDWHILE;
  first Results }
```

The C level implementation of *DOWN* avoids much of the work implied by the above description by using two arrays that hold the successive values given to *Candidates* and *Results* by taking advantage of the stack like access implied by the algorithm. As a result, recursions in depth can be expressed with *DOWN* and achieve effective execution without worry of C stack overflow.

The above definitions now become

```
opposite IS DOWN [ atomic, 0 minus, pass, pass]

sum IS DOWN [ simple, dosum, pack, pass ]

depth IS DOWN [ atomic, 0 first, pass, 1 plus max ]
```

Conclusions

The above discussion suggests that the new transformers that achieve the equivalent of recursive definitions using loops are a worthwhile extension to Nial. They provide most of the useful recursive definitions associated with array theory. They also make explicit the structure of all the pervasive operations in the theory.

For a unary pervasive operation such as *not* or *abs*, both the distribution of the operation for a simple array and the descent to the leaves is achieved with *DOWN* using *pass* as the restructuring operation.

abs IS DOWN [atomic, &abs, pass, pass]

For a binary pervasive operation such as *minus* or *divide*, the distribution of the operation across a pair of simple arrays and the descent to the corresponding leaves is achieved with *DOWN* using *pack* as the restructuring operation.

minus IS DOWN [simple, &minus, pack, pass]

For a multi-pervasive operation such as *sum* or *and*, the reduction on a simple array is achieved using *ACROSS* with *0 first* as the starting value for the reduction. The descent to simple arrays is achieved with *DOWN* using *pack* as the restructuring operation.

sum IS DOWN [simple, ACROSS [0 first, pass, &plus], pack, pass]

In these three examples the operations beginning with ”&” determine all of the behavior having to do with the types of the atoms. By supplying different versions of these routines, this behavior can be modified to suit a particular application of array theory.

The three recursion transformers have been implemented in C using the iterative algorithms and added to the current version of Q’Nial and the V4 Shell. We discuss their use in connection with the V4 shell in Chapter 7.

Chapter 6 A formal development of V6 Array Theory

This paper provides a formal presentation of the array theory that forms the mathematical basis for the expression language of the Nial programming language as implemented in version 6.21 of Q'Nial. In array theory all data objects are viewed as nested rectangular arrays with items that are themselves arrays. Thus, it is a one-sorted theory of a recursive data type that captures the dimensionality concepts of tensor theory and the aggregation concepts of set theory. Combining these two ways of organizing data provides a new way of viewing data that is mathematically rich.

As we have shown in Chapter 4, there are many choices to be made in developing array theory from the basic assumptions of combining dimensionality with nesting. In choosing the theory for the current version of Nial, a deliberate decision was made to reduce the complexity of the theory. It was simplified by limiting the set of empty arrays so that there is a unique empty list, and by producing *fault* values when an operation is only partially defined by the geometry of the arrays.

We make the following assumptions in the development:

- there are five structural primitive operations: *first*, *rest*, and *hitch* corresponding to *car*, *cdr*, and *cons* from Lisp, and *shape* and *reshape* from APL,
- an operation *equal* for array equality,
- nonnegative integers, Boolean values and fault values are atomic arrays
- the arithmetic (+, -, *) and comparison (≤, <, ≥, >) operations for the *nonnegative* integers and the successor and predecessor functions (*&succ, &pred*) are primitive,
- there is only a single empty list, *Null*, and
- shapes are lists.

We use a meta-syntax to express definitions, axioms and theorems, with
- = to denote equality of array expressions,
- **for** to indicate a definition, which may be given by cases and may be recursive,
- **if-then** to indicate implications,
- **iff** to indicate bi-implications,
- **and**, **or** and **not** as logical connectives, and
- **if-then-else** for conditional array expressions.

The theory is one that extends the theory of nonnegative integers with equality and induction [MaWa85, GlJe89]. We assume variables are universally quantified over the universe of arrays in definitions, axioms and theorems unless otherwise indicated.

The definitions, axioms and theorems use the Nial expression syntax. A presentation of the syntax is found in Chapter 3. The basic properties of the array theory operations can be found in the Nial Tools Manual [JeJe98].

1. Fundamental Axiom and properties from Lisp and APL

We state a fundamental axiom that combines the array properties of APL with the list properties of Lisp.

A0 *shape A reshape (first A hitch rest A) = A*

Axiom A0 immediately implies a theorem that states when two arrays are identical.

T0 *A = B* **iff**
shape A = shape B **and** *first A = first B* **and** *rest A = rest B*

The operation *list* gives the list of items of the array. It is used implicitly in many of the operations as described below. We can define it from the implicit use as follows:

D0 *list A* **for** *rest (first A hitch A)*

1.1 Axioms corresponding to Lisp properties

The one-dimensional arrays of the theory are called lists. The operation *hitch* is a list constructor and *first* and *rest* are selectors.

A1 *first (A hitch list B) = A*

A2 *rest (A hitch list B) = list B*

The results of *hitch*, *rest* and *shape* are lists.

A3 *list (A hitch B) = A hitch B*

A4 *list rest A = rest A*

A5 *list shape A = shape A*

A6 to A10 are axioms for coercion to lists of the argument of *first* and *rest*, the second argument of *hitch* and both arguments of *reshape*.

A6 *first list A = first A*

A7 *rest list A = rest A*

A8 *A hitch list B = A hitch B*

A9 *A reshape list B = A reshape B*

A10 *list A reshape B = A reshape B*

We prove that *list* is idempotent.

T1 *list list A = list A*

Proof: *list list A*

= rest (first list A hitch list A)	D0
= rest (first A hitch list A)	A6
= rest (first A hitch A)	A8
= list A	D0

1.2 Axioms corresponding to APL properties of *shape* and *reshape*

The operation *shape* gives the lengths of the axes for an array as a list of integers. The operation *reshape* uses its left argument, expected to be an integer or a list of integers to build an array using the items of the right argument cyclically. In the axioms and theorems for *reshape* we indicate that the left argument is appropriate by stating it as the result of *shape*.

We can view *reshape* as a constructor and *shape* and *list* as selectors for building the dimensionality structure of an array.

A11 *shape A reshape list A = A*

A12 *shape (shape A reshape B) = shape A*

T2 *shape A reshape A = A*

Proof: Immediate from A9 and A11.

2. Geometrical properties

The arrays of the theory have arrays as items at the top level. The number of items of the array is called the **tally** of the array.

2.1 Properties of *tally* and *Null*

We define an operation *tally* by:

D1 *tally A* **for** *first shape list A*

T3 *tally list A = tally A*

Proof: Immediate from D1 and T1.

The array *Null* is the empty list that gives the shape of atoms in the theory.

D2 *Null* **for** *shape tally A*

T4 *list Null = Null*

Proof: Immediate from D2 and A5.

A13 Either *list A = Null* or ∃ arrays *B, C* such that *list A = B hitch list C*

A14 *tally Null = 0*

The operation *hitch* extends the length of a list.

A15 *tally (A hitch B) = &succ tally B*

The operation *rest* shortens a list unless it is already empty.

A16 *rest Null = Null*

T5 *tally rest A = &pred tally A*

Proof: Consider the two cases for *list A* given by A13. Let *list A = Null*. Then

tally rest A

= tally rest list A	A7
= tally rest Null	assumption
= tally Null	A16
= 0	A14

and

&pred tally A

= &pred tally list A	T3
= &pred tally Null	assumption
= &pred 0	A14
= 0	property of *&pred*

Let *list A = B hitch list C*. Then

tally rest A

= tally rest list A	A7
= tally rest (B hitch list C)	assumption
= tally list C	A2

and

&pred tally A

= &pred tally list A	A7
= &pred tally (B hitch list C)	assumption
= &pred &succ tally list C	A15
= tally list C	property of *&pred*

Thus both cases satisfy the theorem.

We prove that the tally of an array is a nonnegative integer.

T6 *tally A* is a nonnegative integer.
Proof: By T3 and A13 either *tally A = 0* or *tally A = tally (B hitch list C) = &succ tally list C*. Thus, by the properties of *&succ* in both cases *tally A* is a nonnegative integer.

We now define the operation *solitary* that makes a list with one item and prove some of its properties.

D3 *solitary A* **for** *A hitch Null*

T7 *tally solitary A = 1*

Proof: Immediate from D3, A14 and A15.

T8 *first solitary A = A*

Proof: Immediate from D3 and A1.

T9 *list solitary A = solitary A*

Proof: Immediate from D3 and A3.

We postulate that the operation *shape,* when applied to a list, returns a solitary integer holding the tally of the list and that the same array is given by applying *list* to the tally.

A17 *shape list A = solitary tally A*

A18 *list tally A = solitary tally A*

2.2 The relationship between *shape* and *tally*

We can show that reshaping an array with its tally gives the list of items of the array.

T10 *tally A reshape A = list A*

Proof:

tally A reshape A	
= tally A reshape list A	A9
= list tally A reshape list A	A10
= solitary tally A reshape list A	A18
= shape list A reshape list A	A17
= list A	T2

T11 **if** *tally A = 0* **then** *list A = Null* **else** *list A = first A hitch rest A*

Proof: Assume *tally A = 0*. By A13, if *list A* ≠ *Null* then *list A = B hitch list C* for some arrays *B* and *C*. But in that case, by A15, *tally A > 0* holds, contradicting the condition. Hence, by A13 the theorem must hold.

Assume *tally A* ≠ *0*. We have

tally (first A hitch rest A)	
= *&succ tally rest A*	A15
= *&succ &pred tally A*	T5
= *tally A*	since *tally A* ≠ *0*

Then *list A*

= *shape list A reshape (first list A hitch rest list A)*	A0
= *tally A reshape (first A hitch rest A)*	D2, A6, A7
= *tally(first A hitch rest A) reshape (first A hitch rest A)*	above fact
= *list (first A hitch rest A)*	T10
= *first A hitch rest A*	A3

We define *&prod* for a list of nonnegative integers.

D4 Let *A* be an array of nonnegative integers. We define

&prod A **for** *1* **if** *list A = Null*

*first A * &prod rest A* **otherwise**

We postulate that arrays are rectangular, that is, the number of items of the array is given by the product of the lengths of each axis of the array.

A19 *tally A = &prod shape A*

2.3 Axioms for other properties of reshape

We postulate that the operation *first* on the result of *shape A reshape B* gets the first item of *B* provided the argument and result are not empty.

A20 **if** *&prod shape A > 0* **and** *tally B > 0* **then**
first (shape A reshape B) = first B

We postulate that, if *B* has as many items as needed for an array of shape *shape A*, then a similar property for *rest* holds.

A21 **if** *&prod shape A = tally B* **then** *rest (shape A reshape B) = rest B*

We postulate that an intermediate reshaping has no effect if the number of items it uses is greater than or equal to the number needed in the final reshape.

A22 **if** *&prod shape A* ≤ *&prod shape B* **then**
shape A reshape (shape B reshape C) = shape A reshape C

We defer specifying the effect of *shape A reshape B* in the case where *B* is empty or has fewer items than *&prod shape A*.

The operation *reshape* can create arrays with an arbitrary number of dimensions. The operation *valence* is defined to measure the dimensionality of an array.

D5 *valence A* **for** *tally shape A*

We show that lists are one-dimensional arrays.

T12 *valence list A = 1*

Proof:

valence list A	
= tally shape list A	D5
= tally solitary tally A	A17
= 1	T7

2.4 Properties of atoms and singles

We show that the nonnegative integers are self-containing.

T13 *first tally A = tally A*

Proof: *first tally A*

= first list tally A	A6
= first solitary tally A	A18
= tally A	T8

We prove that the operation *rest* applied to a nonnegative integer gives the *Null*.

T14 *rest tally A = Null*

Proof: *rest tally A*

= rest list tally A	A7
= rest solitary tally A	A18
= rest (tally A hitch Null)	D3
= Null	A2

An array with no axes is called a **single** in array theory. We introduce the operation *single* that constructs such an array holding an arbitrary array as its sole item.

D6 *single A* **for** *Null reshape solitary A*

T15 *first single A = A*

Proof: *first single A*

= *first (Null reshape solitary A)*	D6
= *first solitary A*	A20, *&prod Null = 1*
= *A*	T8

The operation *single* applied to a nonnegative integer has no effect.

T16 *single tally A = tally A*

Proof: *single tally A*

= *Null reshape solitary tally A*	D6
= *Null reshape (tally A hitch Null)*	D3
= *shape tally A reshape (tally A hitch Null)*	D2
= *shape tally A reshape (first tally A hitch Null)*	T13
= *shape tally A reshape (first tally A hitch rest tally A)*	T14
= *tally A*	A0

We use the above property of nonnegative integers to introduce the concept of an atom.

D7 *atomic A* **for** *single A = A*

Theorem T16 holds for any atom since the only properties of *tally A* used in its proof are T13 and T14 that hold due to A18, which we assume from this point applies to any atom.

We prove some simple facts about *single*.

T17 *shape single A = Null*

Proof: *shape single A*

= *shape (Null reshape solitary A)*	D6
= *shape (shape tally A reshape solitary A)*	D2
= *shape tally A*	A12
= *Null*	D2

T18 *tally single A = 1*

Proof: *tally single A*

= *&prod shape single A*	A19
= *&prod Null*	T17
= *1*	D4

Cor. to T18 *tally tally A = 1*

We show that the list of items of a single is the solitary of the only item.

T19 *solitary A = list single A*

Proof: *list single A*

	= *tally single A reshape single A*	T10
	= *tally single A reshape (Null reshape solitary A)*	D6
	= *1 reshape (Null reshape solitary A)*	T18
	= *1 reshape solitary A*	A22
	= *tally solitary A reshape solitary A*	T7
	= *list solitary A*	T10
	= *solitary A*	T9

T20 *rest single A = Null*

Proof: *rest single A*

	= *rest list single A*	A7
	= *rest solitary A*	T19
	= *Null*	D3, A2

T21 *1 reshape single A = solitary A*

Proof: *1 reshape single A*

	= *tally single A reshape single A*	T18
	= *list single A*	T10
	= *solitary A*	T19

2.5 Some non-equality theorems about lists

The following theorems and axioms indicate some of the distinctions between data objects in the theory.

T22 *tally A ≠ Null*

Proof: Suppose *tally A = Null*. Then

0	= *tally Null*	A14
	= *tally tally A*	assumption
	= *1*	Cor. to T18

which contradicts the Peano axioms.

Cor. to T22 *0 ≠ Null*

We show that using *hitch* to extend an array yields a different array.

T23 *A hitch B ≠ B*

Proof: Assume *A hitch B = B*. Then

tally B		
	= *tally (A hitch B)*	by assumption

= *&succ tally B* A15

contradicting a Peano axiom.

We postulate that a list built with *hitch* is not self-containing.

A23 *A hitch B* ≠ *A*

2.6 Empty arrays

By axiom A13 a list is either the *Null* or a list constructed by *hitch*. By axiom A15 the tally of the latter must be greater than 0. Hence *Null* is the only empty list.

The *Null* has no items. Hence the operation *first* cannot return an item of the array. We introduce special atomic values called **faults** that can be used as the result of an operation for arrays that do not have a "natural" result for the operation being defined. We extend all operations that select items to return the fault, *?address* when the selection is out of bounds for the array.

A24 *first Null = ?address*

We show that when *reshape* is used with a shape containing a zero it produces an array with *Null* as its list of items.

T24 **if** *&prod shape A = 0* **then** *shape A reshape B = shape A reshape Null*

Proof: We have

tally (shape A reshape B)
= *&prod shape (shape A reshape B)* A19
= *&prod shape A* A12
= *0* condition

Then

shape A reshape B
= *shape (shape A reshape B) reshape list (shape A reshape B)* A11
= *shape A reshape list (shape A reshape B)* A12
= *shape A reshape Null* T11, above fact

We prove a theorem about equivalence of arrays that simplifies A0.

T25 *A = B* **iff**
shape A = shape B **and** *list A = list B*

Proof: Assume *A = B*. Then *shape A = shape B*. If *tally A = 0* then *tally B = 0* and by T11 we have *list A = Null = list B*.

If *tally A > 0*, by T0 we have *first A = first B* and *rest A = rest B*, and hence by T11 we have *list A = list B*. This completes the proof of the first implication.

Assume *shape A = shape B* and *list A = list B*. Then by A11 *A = B*.

3. The replacement transformer EACH

An important property of array theory is that the replacement transformer, *EACH* can be used to map one array to another of the same shape by applying an operation to each of the items of the array. In the following let *f* denote a **total** operation on arrays, that is, an operation defined for all arrays.

D8 *EACH f A* **for** *shape A reshape (f first A hitch EACH f rest A)* **if** *tally A ≠ 0*
shape A reshape Null **otherwise**

Using D8 and T24 we can show

T26 *EACH f A = shape A reshape (f first A hitch EACH f rest A)*

With T26 and A12 we can immediately prove the following three theorems:

T27 *shape (EACH f A) = shape A*

T28 *tally EACH f A = tally A*

T29 *EACH f Null = Null*

The effect of *EACH* on array data structures is described by the following theorems.

T30 *EACH f list A = list EACH f A*

Proof:

EACH f list A	
= shape list A reshape (f first list A hitch EACH f rest list A)	D8
= shape list A reshape (f first A hitch EACH f rest A)	A6, A7
= list tally A reshape (f first A hitch EACH f rest A)	A18, A19
= list tally A reshape (shape A reshape (f first A hitch EACH f rest A))	A22
= list tally A reshape EACH f A	D8
= tally EACH f A reshape EACH f A	A10, T28
= list EACH f A	T10

T31 *EACH f (A hitch B) = f A hitch EACH f B*

Proof:

EACH f (A hitch B)	
= shape (A hitch B) reshape	
(f first (A hitch B) hitch EACH f rest (A hitch B))	D8

= shape list (A hitch B) reshape (f A hitch EACH f list B)	A3, A1, A2
= solitary tally (A hitch B) reshape (f A hitch list EACH f B)	A17, T30
= solitary tally (f A hitch EACH f B) reshape (f A hitch EACH f B)	A15, T28
= shape (f A hitch EACH f B) reshape (f A hitch EACH f B)	A19, A3
= f A hitch EACH f B	T2

T32 *EACH f rest A = rest EACH f A*

Proof: By A13 either *list A = Null* or *list A = B hitch list C*.

Consider *list A = Null*. Then

EACH f rest A

= EACH f rest list A	A7
= EACH f rest Null	assumption
= Null	A16, T29

rest EACH f A

= rest list EACH f A	A7
= rest EACH f list A	T30
= rest EACH f Null	assumption
= Null	T29, A16

Consider *list A = B hitch list C*.

EACH f rest A

= EACH f rest list A	A7
= EACH f rest (B hitch list C)	assumption
= EACH f list C	A2

rest EACH f A

= rest list EACH f A	A7
= rest EACH f list A	T30
= rest EACH f (B hitch list C)	assumption
= rest (f B hitch EACH f list C)	T31
= list EACH f list C	A2
= EACH f list C	T30, T1

Both cases have been proved.

T33 *EACH f single A = single f A*

Proof:

EACH f single A

= shape single A reshape (f first single A hitch EACH f rest single A)	D8
= Null reshape (f A hitch EACH f Null A)	T17, T15, T20
= Null reshape (f A hitch Null)	T29

= *single f A* D3, D6

T34 *EACH f A = shape A reshape EACH f list A*

Proof: Immediate from T27, T30 and A11.

T35 **if** *tally A > 0* **then** *first EACH f A = f first A*

Proof: *first EACH f A*

= *first list EACH f A*	A6
= *first EACH f list A*	T30
= *first EACH f (first A hitch rest A)*	T11
= *first (f first A hitch EACH f rest A)*	T31
= *f first A*	A1

T36 **if** *tally A > 0* **then** *EACH f list A = f first A hitch EACH f rest A*

Proof: Immediate from T11 and T31.

T37 **if** *&prod shape A = tally B* **then**
EACH f (shape A reshape B) = shape A reshape EACH f B

Proof: Assume *&prod shape A* = 0. Then both sides reduce to *shape A reshape Null* by T34, T24. Assume *&prod shape A > 0*.

EACH f (shape A reshape B)

= *shape (shape A reshape B) reshape (f first (shape A reshape B) hitch EACH f rest (shape A reshape B)*	D8
= *shape A reshape (f first B hitch EACH f rest B)*	A11, A20, A21
= *shape A reshape (first EACH f B hitch rest EACH f B)*	T35, T32
= *shape A reshape list EACH f B*	T36
= *shape A reshape EACH f B*	A9

We can prove the following **structural induction** theorem for lists that simplifies many proofs below.

T38 Let F be a formula where *A* is not free. **If** for every array A,
if *list A = Null* **then** F(*list A*), **and**
if *tally A > 0* **and** F(*rest A*) **then** F(*list A*),
then for every array *A*, F(*list A*).

Proof: The proof is on induction on the length of *list A*.

F(*list A*) holds for *tally A = 0* by the first condition.

Assume *tally A > 0* and that F(*list B*) holds for all arrays *B* with *tally B < tally A*.
Since *tally rest A = &pred tally A < tally A,* by the assumption F(*rest list A*) = F(*rest A*) holds and hence by the second condition F(*list A*) holds.

This completes the proof by induction over *tally A*.

We now prove that *EACH* distributes over the composition of two total operations.

T39 Let *f* and *g* be total operations. Then
EACH (f g) A = (EACH f) (EACH g) A

Proof: By D8 we see that the shape of both sides is the same and by T34 it is sufficient to prove the theorem holds for *list A*. We prove the result by structural induction using T38. Assume *list A = Null*. Then both sides reduce to *Null* by T29.

Assume *tally A > 0* and that the theorem holds for *rest A*. Then

EACH (f g) list A	
= f g first A hitch EACH (f g) rest A	T36
= (f g first A hitch (EACH f) (EACH g) rest A)	assumption
= EACH f (g first A hitch EACH g rest A)	T31
= (EACH f) (EACH g) list A	T36

The proof by induction is complete.

3.2 The replacement transforms *EACHLEFT* and *EACHRIGHT*

We define two transformers analogous to *EACH* that apply to a pair of arrays, but do the replacement based on only one of the arguments.

D9 *A EACHLEFT f B* **for**

shape A reshape (first A f B hitch (rest A EACHLEFT f B))	**if** *tally A ≠ 0*
shape A reshape Null	**otherwise**

D10 *A EACHRIGHT f B* **for**

shape B reshape (A f first B hitch (A EACHRIGHT f rest B))	**if** *tally B ≠ 0*
shape A reshape Null	**otherwise**

Theorems analogous to T26 to T39 hold for these transformers and are assumed to hold in the rest of the paper.

T40 *A EACHRIGHT f B = EACH (A f) B*

Proof: Using D10 and a simple structural induction proof with T32 yields the result.

We introduce the transformer *CONVERSE* that applies an operation with two arguments in the reversed order.

D11 *A CONVERSE f B* **for** *B f A*

T41 *A EACHLEFT f B = EACH (B CONVERSE f) A*

Proof: Using D9, D11 and a simple structural induction proof with T38 yields the result.

Theorems T40 and T41 allow us to convert expressions that use *EACHRIGHT* and *EACHLEFT* into ones using *EACH* and this capability is useful in several of the proofs that follow.

4. Construction of lists

The operation *hitch* provides a way to construct a list by adding items one at a time to the front. We proceed to define an operation *link* that joins the items of a list together. We begin by defining an intermediate operation *&link* that joins two array together.

D12 *A &link B* **for** *list B* **if** *tally A = 0*
first A hitch (rest A &link B) **otherwise**

We show that the result of *&link* is a list and *&link* treats its arguments as lists.

T42 *list (A &link B) = (A &link B)*

Proof: Each case in the definition D12 produces a list. Thus, the result is a list.

T43 *A &link B = list A &link B = A &link list B*

Proof: Immediate from D12, A6 and A7.

We show that the result of *&link* has the length of the sum of the lengths of the arguments.

T44 *tally (A &link B) = tally A + tally B*

Proof: We prove the theorem by induction on *tally A*.
Let *tally A = 0*. Then *list A = Null* by T11 and
tally (A &link B)

= tally list B	T43, D12
= 0 + tally B	T3, property of *0*
= tally A + tally B	assumption

Let *tally A > 0* and assume the theorem holds for all arrays *C* with *tally C < tally A*.
tally (A &link B)

= tally(first A hitch (rest A &link B))	D12
= &succ tally (rest A &link B)	A15

= *&succ (tally rest A + tally B)*	assumption
= *&succ (&pred tally A + tally B)*	T5
= *tally A + tally B*	since *tally A > 0*

T45 *A &link Null = list A*

Proof: A proof can be done by induction on *tally A*. The details are omitted.

We now introduce the general operation that joins an arbitrary number of arrays together end-to-end.

D13 *link A* **for** *Null* **if** *tally A = 0*
first A &link link rest A **otherwise**

We prove the following simple properties of *link*.

T46 *list link A = link A*

Proof: Immediate from D13, T4 and T42.
T47 *link list A = link A*

Proof: Immediate from D13, A6 and A7.

T48 *link EACH list A = link A*

Proof: By structural induction using T38. Let *list A = Null*. Then
link EACH list A

= *link list EACH list A*	T47
= *link EACH list list A*	T30
= *link EACH list Null*	assumption
= *link Null*	T29
= *link A*	T4, assumption

Assume *tally A > 0* and that the theorem holds for *rest A*. Then
link EACH list A

= *first EACH list A &link link rest EACH list A*	D13
= *list first A &link link EACH list rest A*	T35, T32
= *first A &link link rest A*	T43, assumption
= *link A*	D13

We introduce an intermediate operation *&sum* for arrays of nonnegative integers.

D14 Let *A* be an array of nonnegative integers. Then
&sum A **for** *0* **if** *list A = Null*
first A + &sum rest A **otherwise**

The length of *link A* is the sum of the lengths of the items of *A*.

T49 *tally link A = &sum EACH tally A*

Proof: The theorem can be proved using the structural induction of T38 together with D13, D14 and T44. We omit the details.

T50 *EACH f (A &link B) = EACH f A &link EACH f B*

Proof: Proof by structural induction. If *tally A = 0* then

EACH f (A &link B)	
= EACH f (Null &link B)	assumption, T11
= EACH f list B	D12
= list EACH f B	T30
= EACH f A &link EACH f B	D12, T27, assumption

Assume *tally A > 0* and that the theorem holds for *rest A*. Since *A &link B* is a list

EACH f (A &link B)	
= f first (A &link B) hitch EACH f rest (A &link B)	T36
= f first (first A hitch (rest A &link B)) hitch EACH f rest (A &link B)	D12
= f first A hitch EACH f rest (A &link B)	A1
= f first A hitch EACH f rest (first A hitch (rest A &link B))	D12
= f first A hitch EACH f list (rest A &link B)	A2
= f first A hitch (EACH f rest A &link EACH f B)	T32, inductive assumption
= first EACH f A hitch (rest EACH f A &link EACH f B)	T35, T32
= (first EACH f A hitch rest EACH f A) &link EACH f B	D12
= list (EACH f A) &link EACH f B	T36
= EACH f A &link EACH f B	T43

The proof by induction is complete.

T51 *EACH f link A = link EACH EACH f A*

Proof: By induction using T38. If *tally A = 0* then both sides reduce to *Null*.
Assume *tally A > 0* and the theorem holds for *rest A*. Since *link A* is a list

EACH f link A	
= EACH f (first A &link link rest A)	D13
= EACH f first A &link EACH f link rest A)	T50
= first EACH EACH f A &link EACH f link rest A	T35
= first EACH EACH f A &link link EACH EACH f rest A	inductive assum.
= first EACH EACH f A &link link rest EACH EACH f A	T32
= link EACH EACH f A	D13

We know from the theory of nonnegative integers that addition is associative. That is that (A + B) + C = A + (B + C). The concept of associativity can be expressed in array theory by the statement:

An operation *f* is said to be **associative** over a set of arrays U if for every *A* in U

f link A = f EACH f A.

We extend the usual concept of the associativity of addition by assuming that *&sum* is associative for arrays that are arrays of arrays of nonnegative integers.

A25 *&sum link EACH EACH tally A = &sum EACH &sum EACH EACH tally A*

We can now prove the following theorem about *link*.

T52 *tally link EACH link A = tally link link A*

Proof:

tally link EACH link A	
= &sum EACH tally EACH link A	T49
= &sum EACH (tally link) A	T39
= &sum EACH (&sum EACH tally) A	T49
= &sum EACH &sum EACH EACH tally A	T39
tally link link A	
= &sum EACH tally link A	T49
= &sum link EACH EACH tally A	T51

and both sides are equal by A25.

The above theorem suggests that link is associative:

T53 *link EACH link A = link link A*

A proof of this theorem is deferred.

4.1 Properties of pairing

We define the operation *pair* that gives the meaning of the strand notation *A B*.

D15 *A pair B* **for** *A hitch solitary B*

By syntax convention the infix use of an operation in array theory means the application formed by the operation applied to the pair of arguments. Thus, for any operation *f* we have

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

We introduce the operation *second* to select the second item of an array.

D16 *second A* **for** *first rest A*

We can prove the following theorems about *pair* and *second*. The details are omitted.

T54 *tally (A pair B) = 2*

T55 *first (A pair B) = A*

T56 *rest (A pair B) = solitary B*

T57 *second (A pair B) = B*

T58 *A pair B = A hitch single B*

We can now show that the operation *link* applied to a pair is *&link*.

T59 *A link B = A &link B*

Proof: *A link B*

= link (A B)	syntax convention
= first (A pair B) &link rest (A pair B)	D13
= A &link link solitary B	T55, T56
= A &link (first solitary B &link link rest solitary B)	D13
= A &link (B &link link Null)	T20
= A &link (B &link Null)	D13
= A &link B	T45, T43

T60 *A hitch B = single A link B*

Proof: We can easily prove that the *tally* of both sides is *1 + tally B* and since both are lists they have the same shape. We now show that the *first* and *rest* of both sides are the same.

Consider *first (single A link B)*

= first (single A &link B)	T59
= first (first single A hitch (rest single A &link B))	D12
= first single A	A1
= A	T15
= first (A hitch B)	A1

Consider *rest (single A link B)*

= rest (single A &link B)	T59
= rest (first single A hitch (rest single A &link B))	D12
= list (rest single A &link B)	A2
= rest list single A &link B	A7, T42
= Null &link B	T20
= list B	D12
= rest (A hitch B)	A2

Thus, by T0 the theorem is proved.

T61 *A pair B = single A link single B*

Proof: *A pair B*

= A hitch solitary B	D15
= single A link list single B	T60, T19
= single A link single B	T59, T43

5. Construction of multidimensional arrays by Cartesian products

We define an intermediate operation, *&cart* that forms all combinations of pairs of items from its two arguments and stores them in an array with shape equal to the *link* of the shape of the two arguments.

D17 *A &cart B* **for**
shape A link shape B reshape link (A EACHLEFT EACHRIGHT pair B)

T62 *shape (A &cart B) = shape A link shape B*

Proof: Since both *shape A* and *shape B* are lists of nonnegative integers their *link* is also. Thus *shape A link shape B* is a valid shape and hence by A11 and D17 the theorem holds.

D18 *cart A* **for**
single A **if** *tally A = 0*
shape A EACHRIGHT reshape EACH hitch (first A &cart cart rest A) **otherwise**

T63 *shape cart A = link EACH shape A*

Proof: By structural induction on *list A*. Assume *tally A = 0*. Then

shape cart A

= shape single A	D18
= Null	T17

link EACH shape A

= link list EACH shape A	T47
= link EACH shape list A	T30
= link EACH shape Null	T11
= link Null	T29
= Null	D13

and the theorem holds if *list A* is *Null*.

Assume *tally A > 0* and that theorem holds for *rest A*. Then since *EACHRIGHT* preserves the shape of the right argument and *EACH* preserves shape, we have by D18

shape cart A

= shape (first A &cart cart rest A)	D18
= shape first A link shape cart rest A	D17, T62
= shape first A &link link EACH shape rest A	T59, assumption
= shape first A &link link rest EACH shape A	T32
= link (shape first A hitch EACH shape rest A)	D13, T59
= link list EACH shape A	T36, T30
= link EACH shape A	T47

The proof by induction is complete.

We also know from the theory of nonnegative integers that multiplication is associative and hence we postulate that *&prod* satisfies:

A26 *&prod link EACH EACH tally A = &prod EACH &prod EACH EACH tally A*

We can show that number of items of *cart A* is the product of the numbers of items in the items of *A*.

T64 *tally cart A = &prod EACH tally A*

Proof: *tally cart A*

= &prod shape cart A	A19
= &prod link EACH shape A	T63
= &prod EACH &prod EACH shape A	A26
= &prod EACH (&prod shape) A	T39
= &prod EACH tally A	A19

We prove some properties of *cart*.

T65 *cart Null = single Null*

Proof: Immediate from D18.

T66 *cart single A = EACH single A*

Proof: *cart single A*

= shape single A EACHRIGHT reshape EACH hitch (first single A &cart cart rest single A)	D18
= Null EACHRIGHT reshape EACH hitch(A &cart cart Null)	T17, T15, T20
= Null EACHRIGHT reshape EACH hitch (A &cart single Null)	T65
= Null EACHRIGHT reshape EACH hitch (shape A link Null reshape link (A EACHLEFT EACHRIGHT pair single Null))	D17, T17
= Null EACHRIGHT reshape EACH hitch (shape A reshape link (A EACHLEFT EACHRIGHT pair single Null))	T45, A5
= EACH (Null reshape) EACH hitch (shape A reshape link	

(A EACHLEFT EACHRIGHT pair single Null)) T40
= *shape A reshape EACH (Null reshape hitch) link*
(A EACHLEFT EACHRIGHT pair single Null) T37, T39
= *shape A reshape link EACH EACH (Null reshape hitch)*
(A EACHLEFT EACHRIGHT pair single Null) T51
= *shape A reshape link EACH (EACH (Null reshape hitch))*
EACH (single Null CONVERSE EACHRIGHT pair) A T41
= *shape A reshape link EACH (EACH (Null reshape hitch)*
(single Null CONVERSE EACHRIGHT pair)) A T39

Let *g = EACH (Null reshape hitch)(single Null CONVERSE EACHRIGHT pair)*.
We prove the following
Lemma 1: *g B = single single B*
g B
= *EACH (Null reshape hitch)*
(single Null CONVERSE EACHRIGHT pair) B definition of g
= *EACH (Null reshape hitch) (B EACHRIGHT pair single Null)* D11
= *EACH (Null reshape hitch) single (B pair Null)* T33 for *EACHRIGHT*
= *single (Null reshape hitch) (B pair Null)* T33
= *single (Null reshape (B hitch Null))* infix notation
= *single single B* D3, D6

Returning to the proof of the theorem we have
cart single A
= *shape A reshape link EACH g A*
and by T34 it is sufficient to prove
Lemma 2: *link EACH (single single) A = EACH single list A.*

Assume *tally A = 0*. Then both sides reduce to *Null*.
Assume *tally A > 0* and that the lemma holds for *rest A*. Then
link EACH (single single) A
= *first EACH (single single) A &link*
link rest EACH (single single) A) D3
= *single single first A &link link EACH (single single) rest A* T35, T32
= *first single single first A hitch (rest single single first A*
&link link EACH (single single) rest A D12
= *single first A hitch (Null &link link EACH (single single) rest A)* T15, T20
= *single first A hitch list link EACH (single single) rest A* D12
= *single first A hitch rest EACH single list A* assumption
= *EACH single list A* T32, T36

Cor. to T66 *cart solitary A = EACH solitary A*

Proof: A similar proof can be constructed.

We have seen that proof of theorems about *cart* from definitions D17 and D18 are quite lengthy. However, all the properties of *cart* stated in the following four theorems can be shown to hold. We defer the proofs.

T67 *cart list A = EACH list cart A*

T68 **if** *tally cart A > 0* **then** *first cart A = EACH first A*

T69 *A cart B = A &cart B*

T70 *EACH EACH f cart A = cart EACH EACH f A*

We prove the following relationship between *cart* and *link*.

T71 *tally cart link A = tally EACH link cart EACH cart A*

Proof:

tally cart link A
= &prod EACH tally link A T64
= &prod link EACH EACH tally A T51

and

tally EACH link cart EACH cart A
= tally cart EACH cart A T28
= &prod EACH tally EACH cart A T64
= &prod EACH (tally cart) A T39
= &prod EACH (&prod EACH tally) A T64
= &prod EACH &prod EACH EACH tally A T39

Thus the proof is complete by A26.

Theorem T71 suggests that the following theorem holds.

T72 *cart link A = EACH link cart EACH cart A*

A proof of the theorem is deferred.

6. Boolean operations and membership

The operation *equal* computes array equality. The result of *equal* is an atomic Boolean number, either *l* denoting *Truth* or *o* denoting *Falsehood*.

A27 *A equal B = l* **if** *A = B*
o **if** *A ≠ B*

From this point on in the development we will use Boolean valued expressions in conditions or statements of axioms or theorems without equating them to *Truth*. For example, we state that the Boolean values are atomic by:

A28 *atomic equal A*

We define preliminary definitions for operations corresponding to the logical connectives. The definitions for *&not*, *&and* and *&or* are first given for pairs of Boolean values.

D19 Let *A* be a Boolean value. Then
&not A **for** *l* **if** *A = o*
o **if** *A = l*

D20 Let *A* and *B* be Boolean values. Then
A &and B **for** *l* **if** *A = l* **and** *B = l*
o **otherwise**

D21 Let *A* and *B* be Boolean values. Then
A &or B **for** *o* **if** *A = o* **and** *B = o*
l **otherwise**

We extend the definitions for *&and* and *&or* to arrays of Boolean values.

D22 Let *A* be an array with Boolean values as items.
&and A **for** *l* **if** *tally A = 0*
first A &and &and rest A **otherwise**

D23 Let *A* be an array with Boolean values as items.
&or A **for** *o* **if** *tally A = 0*
first A &or &or rest A **otherwise**

We now give an axiom that extends *equal* to arrays that are not pairs.

A29 *equal A = l* **if** *tally A ≤ 1*
first A equal second A &and equal rest rest A **otherwise**

T73 *equal A = equal list A*

Proof: Immediate from A29, A6, A7 and D16.

We prove that if the first item of the argument to *&or* is true then the result is true.

T74 **if** *A* is an array with Boolean items **then** *&or (l hitch A) = l*

Proof: *&or (l hitch A)*

= *first (l hitch A) &or &or rest (l hitch A)*	D23
= *l &or &or list A*	A1
= *l*	D21

We introduce an operation that selects items from an array *B* based on the Boolean pattern of an array *A* of the same length.

D24 Let *A* be an array with Boolean values as items and let *B* an array with *tally A = tally B*. We define

A &sublist B **for** *Null* **if** *tally B = 0*
first B hitch (rest A &sublist rest B) **if** *first A = l*
rest A &sublist rest B **if** *first A = o*

We show the following theorem that assists in proofs below.

T75 **if** *tally A = tally B* **and** *first A = l* **then** *first (A &sublist B) = first B*

Proof: Since *first A = l*

first (A &sublist B)	
= *first (first B hitch (rest A &sublist rest B))*	D24
= *first B*	A1

We define the operation *in* to test whether an array *A* is an item of another array *B*.

D25 *A in B* **for** *&or (A EACHRIGHT equal B)*

T76 *A in B* = *A in list B*

Proof:

A in B	
= *&or (A EACHRIGHT equal B)*	D25
= *&or list (A EACHRIGHT equal B)*	D23, A6, A7
= *&or (A EACHRIGHT equal list B)*	T30 for *EACHRIGHT*
= *A in list B*	D25

The first item of a nonempty array is in the array.

T77 **if** *tally A > 0* **then** *first A in A*

Proof:

first A in A	
= *first A in list A*	T76
= *&or (first A EACHRIGHT equal list A)*	D25
= *&or EACH (first A equal) list A*	T40
= *&or (EACH (first A equal) (first A hitch rest A)*	condition, T11
= *&or (l hitch EACH (first A equal) rest A)*	T31, A27

$= l$ T74

We prove the following theorem that states that if *&sublist* selects items then the first item of the result is in the second argument.

T78 **if** *tally A = tally B* **and** *&or A* **then** *first (A &sublist B) in list B*

Proof: Since *&or A* holds we can apply *rest* to *A* until we have *C = l hitch D*. A proof of this fact can be given by induction. Apply *rest* to *B* the same number of times to obtain *E*. Since *tally A = tally B* we also have *tally C = tally E > 0*. Then

first (A &sublist B)
= *first (C &sublist E)* D24
= *first E* T75

Then

first (A &sublist B) in list B
= *&or (first (A &sublist B) EACHRIGHT equal list B)* D25
= *&or (first E EACHRIGHT equal list B)* above
= *&or (EACH (first E equal) list B)* T40

Applying T32 for as many times as *rest* was applied above,

first (A &sublist B) in list B
= *&or (EACH (first E equal) E)* T32
= *&or (l hitch EACH (first E equal) rest E)* T11, T31, A27
= *l* T74

7. Addressing and selection operations

We define the addressing scheme for arrays in this section. We begin by defining *&tell*, which generates the list of the first *N* integers beginning with zero.

D26 Let *N* be a nonnegative integer. We define

&tell N **for** *Null* **if** *N = 0*
0 hitch EACH &succ &tell &pred N **otherwise**

T79 **if** *N* is a nonnegative integer **then** *tally &tell N = N*

Proof: A simple induction proof on the integers can be constructed.

T80 **if** *N* is a nonnegative integer **then** *list &tell N = &tell N*
Proof: Immediate from D26, T4 and A3.

T81 **if** *N* is a nonnegative integer **then** *shape &tell N = list N*

Proof: *shape &tell N*
= *solitary tally &tell N* T80, A17
= *list tally &tell N* A18

$= list\ N$ T79

We define the operation *tell* to generalize *&tell* by generating an array of lists of nonnegative integers if its argument is a list of nonnegative integers.

D27 Let A be a nonnegative integer or a list of nonnegative integer. Then
tell A **for** *&tell A* **if** *A* is a nonnegative integer
cart EACH &tell A **if** *A* is a list of nonnegative integers

T82 **if** $tally\ A > 0$ **then** $first\ tell\ tally\ A = 0$

Proof: *first tell tally A*

$= first\ \&tell\ tally\ A$	T6, D27
$= first\ (0\ hitch\ \&EACH\ \&succ\ \&tell\ \&pred\ tally\ A)$	D26
$= 0$	A1

We want addresses for a list to be integers. We define the operation *suit* to produce the list of an array that has zero or two or more items and the single if it has one item.

D28 *suit A* **for** *single first A* **if** $tally\ A = 1$
list A **otherwise**

We prove some simple properties of *suit*.

T83 $link\ suit\ A = link\ A$

Proof: If $tally\ A = 1$ then
link suit A

$= link\ single\ first\ A$	D28
$= link\ solitary\ first\ A$	T47, T19
$= link\ list\ A$	since $tally\ A = 1$
$= link\ A$	T47

Otherwise, the result is immediate from T47.

T84 $EACH\ f\ suit\ A = suit\ EACH\ f\ A$

Proof: Immediate from D28, T33 and T30.

We define the concept of a **simple** array to explain some properties of *shape*. An array is simple if each of its items is atomic.

D29 *simple A* **for** $A = EACH\ single\ A$

We postulate that *shape* returns a simple array.

A30 *simple shape A*

T85 **if** *simple A* **then** *link A = list A*

Proof: By structural induction with T38. If *tally A = 0* then both sides reduce to *Null*. Assume *tally A > 0* and the theorem holds for *rest A*. Then

link A

= link EACH single A	condition, D29
= first EACH single A &link link EACH single rest A	D13
= single first A &link link rest EACH single A	T35, T32
= single first A &link link rest A	D29
= single first A &link list rest A	assumption
= first single first A hitch (rest single first A &link list rest A)	D12
= first A hitch (Null &link rest A)	T15, T20, A4
= first A hitch list rest A	D12
= list A	T11

We define the operation *grid* that generates the array of addresses for an array.

D30 *grid A* **for** *tell suit shape A*

We show that the grid for a list is a list of integers.

T86 *grid list A = tell tally A*

Proof: *grid list A*

= tell suit shape list A	D30
= tell suit solitary tally A	A17
= tell single first solitary tally A	T7, D28
= tell single tally A	T8
= tell tally A	T16

The array of addresses for an array has the array's shape.

T87 *shape grid A = shape A*

Proof: *shape grid A*

= shape tell suit shape A	D30

If *tally shape A = 1* then

shape grid A

= shape tell single first shape A	D28
= shape tell first shape A	A30, D29, T35
= shape &tell first shape A	D27
= list first shape A	T81
= shape A	assumption, A30

Otherwise

shape grid A

= *shape cart EACH &tell suit shape A*	D30
= *link EACH shape EACH &tell suit shape A*	T63
= *link EACH (shape &tell) suit shape A*	T39
= *link suit EACH (shape &tell) shape A*	T84
= *link EACH (shape &tell) shape A*	T83
= *link EACH list shape A*	T81, A30
= *link shape A*	T48
= *list shape A*	A30, T85
= *shape A*	A5

T88 *grid single A = single Null*

Proof: *grid single A*

= *tell suit shape single A*	D30
= *tell Null*	T17, D28, T4
= *cart EACH &tell Null*	D27
= *cart Null*	T29
= *single Null*	T65

We see that the array of addresses for an array *A* has the same shape as *A*. If *A* is a list the addresses are the items of *tell tally A*. If *A* is a single then the address of *A* is *Null*. If *A* has valence two or higher then the addresses are lists of nonnegative integers with their tally equal to the valence of *A*.

We introduce two operations for searching for an item in an array. The first operation finds the addresses of all occurrences, the second finds the address of the first occurrence or returns the suit of the shape.

D31 *A findall B* **for** *A EACHRIGHT equal B &sublist grid B*

D32 *A find B* **for** *first (A findall B)* **if** *A in B*
suit shape B **otherwise**

T89 **if** *tally A > 0* **then** *first A find list A = 0*

Proof: We have *first A in list A* by T76 and T77.

first A find list A

= *first (first A findall list A)*	D32
= *first (first A EACHRIGHT equal list A &sublist grid list A)*	D31
= *first (EACH (first A equal) list A &sublist tell tally A)*	T40
= *first (l hitch EACH (first A equal) &sublist tell tally A)*	T36, A27
= *first tell tally A*	T75
= *0*	T82

T90 **if** *A in B* **then** *A find B in grid B*

Proof: *A in B* implies *&or (A EACHRIGHT equal B)* holds. It also implies
A find B
= *first (A EACHRIGHT equal B &sublist grid B)* D32, D31
Since *tally (A EACHRIGHT equal B) = tally B* the conditions for T78 hold and
A find B is in *list grid B* and hence the theorem holds by T76.

We define an operation to select an item of an array by using its integer address in the corresponding list of items. We refer to the integer as the **index** of the item.

D33 *I &picklist A* **for** *first A* **if** *I = 0*
&pred I &picklist rest A **if** *0 < I < tally A*
?address **otherwise**

T91 *I &picklist A = I &picklist list A*

Proof: Immediate from A6, A7 and D33.

We show that *&picklist* corresponds to *first* and *second* with the left argument equal to *zero* and *one* respectively.

T92 *0 &picklist A = first A*

Proof: Immediate from D33.

T93 *1 &picklist A = second A*

Proof: If *tally A < 2* then both sides reduce to *?address*. Assume *tally A ≥ 2*.
1 &picklist A
= *&pred 1 &picklist rest A* D33
= *0 &picklist rest A* property of *&pred*
= *first rest A* T92
= *second A* D16

We prove some properties of *&picklist*.

T94 **if** *0 ≤ I < tally A* **then** *I &picklist A in A*

Proof: The condition implies *tally A > 0*. Proof by structural induction using T38.
I &picklist A in A
= *first A in A* D33
= *l* T77.
Assume *I >0* and that the theorem holds for *rest A*. Then
I &picklist A in A
= *&pred I &picklist rest A in A* D33
= *l* assumption

T95 **if** $0 \leq I < tally\ A$ **then** $I\ \&picklist\ EACH\ f\ A = f\ (I\ \&picklist\ A)$

Proof: By induction on I. Assume $I = 0$. Then

$I\ \&picklist\ EACH\ f\ A$	
$= 0\ \&picklist\ list\ EACH\ f\ A$	T91
$= first\ (f\ first\ A\ hitch\ EACH\ f\ rest\ A)$	T30, T36
$= f\ first\ A$	A1
$= f\ (0\ \&picklist\ A)$	T92

Assume I > 0 and the theorem holds for $J < I$. Then

$I\ \&picklist\ EACH\ f\ A$	
$= \&pred\ I\ \&picklist\ rest\ EACH\ f\ A$	D33
$= \&pred\ I\ \&picklist\ EACH\ f\ rest\ A$	T32
$= f\ (\&pred\ I\ \&picklist\ rest\ A)$	inductive assumption
$= f\ (I\ \&picklist\ A)$	D33

We prove a theorem that states an equivalence under which the lists of items of two arrays are identical.

T96 $list\ A = list\ B$ **iff**
$tally\ A = tally\ B$, and
$I\ \&picklist\ A = I\ \&picklist\ B$, for every $I\ in\ grid\ list\ A$

Proof: Assume $list\ A = list\ B$. Then

$tally\ A$	
$= first\ shape\ list\ A$	D1
$= first\ shape\ list\ B$	assumption
$= tally\ B$	D1

Let $I\ in\ grid\ list\ A$ hold. Then
If $I = 0$ then $tally\ A > 0$.

$0\ \&picklist\ A$	
$= first\ A$	T92
$= first\ list\ A$	A6
$= first\ list\ B$	assumption
$= 0\ \&\ picklist\ B$	T92

Assume $0 < I < tally\ A$.

$I\ \&picklist\ A$	
$= \&pred\ I\ \&picklist\ rest\ A$	D33
$= \&pred\ I\ \&picklist\ rest\ list\ A$	A7
$= \&pred\ I\ \&picklist\ rest\ list\ B$	assumption
$= I\ \&picklist\ B$	A7, D33

This completes the proof the first implication.

Let $tally\ A = tally\ B$ and $I\ picklist\ A = I\ picklist\ B$, for I in $grid\ list\ A$. Then
We prove the second implication by structural induction using T38.
If $tally\ A = 0$ then $list\ A = Null = list\ B$.
Assume that $tally\ A > 0$ and that the implication is true for $rest\ A$. Then

list A

= *first A hitch rest A*	T11
= *0 &picklist A hitch rest A*	T92
= *0 &picklist B hitch rest A*	condition
= *0 &picklist B hitch rest B*	inductive assumption
= *first B hitch rest B*	T92
= *list B*	T11

This completes the proof of the second implication.

T97 **if** $0 \leq I < tally\ A$ **then** *I &picklist tell tally A = I*

Proof: By induction. Assume *I = 0* then

I &picklist tell tally A

= *first tell tally A*	T92
= *0*	T82

Assume $I > 0$ and that *J &picklist tell tally A = J* for $J < I$. Then

I &picklist tell tally A

= *&pred I &picklist rest &tell tally A*	D33, D27
= *&pred I &picklist EACH &succ &tell &pred tally A*	D26, A2
= *&succ (&pred I &picklist &tell &pred tally A)*	T95
= *&succ &pred I*	assumption
= *I*	since $I > 0$

We introduce the operation *&atoi* that converts an address for a location in an array *A* into an index for the corresponding item in *list A*.

D34 *I &atoi A* **for** *I find list tell suit A*

T98 **if** *I in grid A* **then** *I &atoi shape A in grid list A*

Proof:

I &atoi shape A

= *I find list tell suit shape A*	D34
= *I find list grid A*	D30

Since *I in grid A* we have *I in list grid A* by T76 and by T90 we have

I find list grid A in grid list A

= *I &atoi shape A in grid list A*	above fact

T99 **if** $0 \leq I < tally\ A$ **then** *I find tell tally A = I*

Proof:

I find tell tally A

= *first (I findall tell tally A)*	D32
= *first (I EACHRIGHT equal tell tally A &sublist tell tally A)*	D31
= *first (EACH (I equal) tell tally A &sublist tell tally A)*	T40
= *first ((I equal first tell tally A hitch*	

EACH (I equal) rest tell tally A) &sublist tell tally A) T36

If *I = 0* then

I find tell tally A

= *first (l hitch EACH (0 equal) rest tell tally A) &sublist tell tally A)* above fact

= *first tell tally A* T75

= *0* T82

Assume *I > 0*. Then the first *I* items of *EACH (I equal) tell tally A* are *o* and the *&sublist* operation will apply *rest* to its arguments *I* times. We can prove by induction that *I* applications of *rest* to *tell tally A* results in a list with first item *I*, and hence the result is true.

We define the general selection operation *pick*.

D35 *I pick A* **for** *suit I &atoi shape A &picklist A*

We prove that *pick* behaves like *&picklist* on a list.

T100 **if** *0 ≤ I < tally A* **then** *I pick list A = I &picklist A*

Proof:

I pick list A

= *I &atoi shape list A & picklist list A* D35

= *I find list tell suit shape list A & picklist A* D34, T91

= *I find list grid list A &picklist A* D30

= *I find tell tally A &picklist A* T86, D27, T80

= *I &picklist A* T99

We prove the following result for picking with an address.

T101 **if** *I in grid A* **then** *I pick EACH f A = f (I pick A)*

Proof: Since *I in grid A* we know that *tally A > 0* and *suit I = I*. Then

I pick EACH f A

= *I &atoi shape A &picklist EACH f A* D35

= *f (I &atoi shape A &picklist A)* T95

= *f (I pick A)* D35

The relationship between *pick* and *find* is expressed by the following axioms:

A31 **if** *A in B* **then** *A find B pick B = A*

A32 **if** *0 ≤ I < tally A* **then** *I pick list grid A find list grid A = I*

We also introduce the operation *&itoa* that converts an index for an item in *list A* to the address for the corresponding item in *A*.

D36 *I &itoa A* **for** *I &picklist tell suit A*

T102 **if** *I in grid list A* **then** *I &itoa shape A in grid A*

Proof: By T86 and the condition we have $0 \le I < tally\ A$.

I &itoa shape A

= *I &picklist tell suit shape A*	D36
= *I &picklist grid A*	T91, D30

which by T94 is in *grid A*.

We show that converting an index to an address is invertible.

T103 **if** *I in grid list A* **then** *(I &itoa shape A) &atoi shape A = I*

Proof: By T86 and the condition we have $0 \le I < tally\ A$.

(I &itoa shape A) &atoi shape A

= *(I &picklist tell suit shape A) find list tell suit shape A*	D34, D36
= *(I &picklist grid A) find list grid A*	D30
= *(I pick list grid A) find list grid A*	T100
= *I*	A32

The addresses in *grid A* are in their own locations.

T104 **if** *suit I in grid A* **then** *I pick grid A = suit I*

Proof:

I pick grid A

= *suit I &atoi shape grid A &picklist grid A*	D35
= *suit I find list tell suit shape grid A &picklist grid A*	D34
= *suit I find list grid A pick list grid A*	T87, D30, T100
= *suit I*	A31

We now prove a theorem that states that two arrays are equal if and only if they have the same shape and the same items at each location. This theorem can be viewed as corresponding to the axiom of extensionality in set theory, which states that two sets are equal if they contain the same members.

T105 *A = B* **iff**

shape A = shape B, **and**

I pick A = I pick B for every *I in grid A*

Proof: Assume *A = B*. By T25 we have *shape A = shape B* and *list A = list B*. We also have *grid A = grid B* by D30. To complete the first implication we need to show that *I pick A = I pick B* for every *I in grid A*. For *I in grid A*

I pick A

= *suit I find list grid A &picklist list A*	D35
= *suit I find list grid B &picklist list B*	above facts

= *I pick B*	D35

Assume *shape A = shape B* and *I pick A = I pick B* for every *I in grid A*.

By T25 the proof will be complete if we prove *list A = list B*.

Since by A19 we also have *tally A = tally B*, by T96 the proof that *list A = list B* will be complete if we can show that *I pick A = I pick B* for *I in grid A* implies *J &picklist A = J &picklist B* for *J in grid list A*.

Let *I in grid A* hold and let *J = I &atoi shape A*. We have *I = suit I* and *J in grid list A* by T98. Then

J &picklist list A

= *suit I &atoi shape A &picklist list A*	defn of *J*, fact about *I*
= *I pick A*	D35
= *I pick B*	assumption
= *suit I &atoi shape A &picklist list B*	D35
= *J &picklist list B*	defn of *J*, fact about *I*

The proof is complete if we can show that for every *J* in *grid list A* there exists an *I* in *grid A* so that *J = I &atoi shape A*. Let *J* be in *grid list A* and let *I = J &itoa shape A*. Then *I* is in *grid A* by T102 and satisfies the requirement since

I &atoi shape A

= *J &itoa shape A &atoi shape A*	defn of *I*
= *J*	T103

We introduce an operation that selects multiple items using an array of addresses.

D37 *A choose B* **for** *A EACHLEFT pick B*

We can now establish that *grid A* is the array of addresses.

T106 *grid A choose A = A*

Proof: We prove the result by using theorem T105.

shape (grid A choose A)

= *shape (grid A EACHLEFT pick B)*	D37
= *shape grid A*	T27 for EACHLEFT
= *shape A*	T87

Let *I* be in *grid A*. Then

I pick (grid A choose A)

= *I pick (grid A EACHLEFT pick A)*	D37
= *I pick (EACH (A CONVERSE pick) grid A)*	T41
= *A CONVERSE pick (I pick grid A)*	T101
= *A CONVERSE pick I*	T104

= *I pick A* D11

In all of the above work, the ordering of the items of *grid A* has not been discussed. If *A* is a list then the items are the integers from *0* to *tally A* in increasing order. However, the order of the items of *cart EACH tell shape A* is controlled by the order in which *reshape* maps the items from the list of items. The choice is to use row major ordering, which is specified by the following axiom.

A33 The items of *list grid A* are in lexicographical order.

Theorem T105 can be used to prove that two array theory expressions are equivalent as was done in the proof of T106. It can also be used to define operations by stating the shape of the result and the items at each location in the result. For example, we can define the operation *reverse* as follows.

D38 *reverse* is the operation that satisfies:
shape reverse A = shape A
I pick reverse A = &pred tally A - (I &atoi shape A) &picklist A **if** *I in grid A*

We can prove that *reverse* is its own inverse.

T107 *reverse reverse A = A*

Proof: We use T105 to show both sides are equal. By D38 the shapes on both sides are equal.

I pick reverse reverse A
= *&pred tally reverse A - (I &atoi shape reverse A)*
&picklist reverse A D38
= *&pred tally A - (I &atoi shape A) pick list reverse A* D38, T100
= *&pred tally A - (&pred tally A - (I &atoi shape A)*
&atoi shape list A) &picklist A D38

We first prove a lemma.
Lemma: (*&pred tally A - (I &atoi shape A)) &atoi shape list A*
= &pred tally A - (I &atoi shape A)
Proof:
&pred tally A - (I &atoi shape A) &atoi shape list A
= *&pred tally A - (I &atoi shape A) find list tell suit shape list A* D34
= *&pred tally A - (I &atoi shape A) find list grid list A* D30
= *&pred tally A - (I &atoi shape A) find list tell tally A* T86
= *&pred tally A - (I &atoi shape A)* T80, T99

Continuing the proof of the main theorem,
I pick reverse reverse A
= *&pred tally A - (&pred tally A - (I &atoi shape A)) &picklist A* Lemma
= *I &atoi shape A &picklist A* arithmetic

= *I pick A* D35

Thus the conditions of T105 are satisfied and the proof is complete.

8. Conclusion

The development of V6 array theory that we have presented here establishes all the properties of the primitive operations *shape*, *first* and *rest*, of *equal* with a single array argument and of *hitch* with a pair. We have deferred the complete specification of *A reshape B* as mentioned in section 2.3. In V6.21 Nial, if *B is* nonempty then the items of *B* are used cyclically to form a list with as many items as needed by *&prod A*. If *B* is empty then the fault value *?fill* is used for the items of the result. For both *hitch* and *reshape* if their argument is not a pair then the result is a fault. Thus, the specification of the six primitives as total operations is complete.

The above presentation deferred the proof of a few of the theorems concerning the operations *link* and *cart*. The theorems hold because of the geometry of the array data structures, but the proofs are quite complex using the formal definitions. Using T105, equivalent definitions of *link* and *cart* can be established and these may prove easier to establish the proofs of the theorems.

Some of the operations we have defined have not been specified completely. For example, *&prod* is only defined for arrays of nonnegative integers. However, we have only used the partially defined operations with arguments within the domains given in the definitions. In the full theory, all such operations are defined for all arrays by extending the definitions given here. Hence the use of T38 with the partially defined operations, as in the proof of T64, is justified.

The definition of many of the operations of array theory has not been established in this presentation. However, from the base of operations given here a sequence of definitions in Nial can be developed that establishes the semantics of the remaining array theoretic operations. Chapter 8 presents such a sequence for the V6 theory.

9. Appendix of definitions, axioms and theorems

Definitions

D0 *list A* **for** *rest (first A hitch A)*

D1 *tally A* **for** *first shape list A*

D2 *Null* **for** *shape tally A*

D3 *solitary A* **for** *A hitch Null*

D4 Let *A* be an array of nonnegative integers. We define

&prod A **for** *1* **if** *list A = Null*
*first A * &prod rest A* **otherwise**

D5 *valence A* **for** *tally shape A*

D6 *single A* **for** *Null reshape solitary A*

D7 *atomic A* **for** *single A = A*

D8 *EACH f A* **for** *shape A reshape (f first A hitch EACH f rest A)* **if** *tally A ≠ 0*
shape A reshape Null **otherwise**

D9 *A EACHLEFT f B* **for**
shape A reshape (first A f B hitch (rest A EACHLEFT f B)) **if** *tally A ≠ 0*
shape A reshape Null **otherwise**

D10 *A EACHRIGHT f B* **for**
shape B reshape (A f first B hitch (A EACHRIGHT f rest B)) **if** *tally B ≠ 0*
shape A reshape Null **otherwise**

D11 *A CONVERSE f B* **for** *B f A*

D12 *A &link B* **for** *list B* **if** *tally A = 0*
first A hitch (rest A &link B) **otherwise**

D13 *link A* **for** *Null* **if** *tally A = 0*
first A &link link rest A **otherwise**

D14 Let *A* be an array of nonnegative integers. Then

&sum A **for** *0* **if** *list A = Null*
first A + &sum rest A **otherwise**

D15 *A pair B* **for** *A hitch solitary B*

D16 *second A* **for** *first rest A*

D17 *A &cart B* **for**
shape A link shape B reshape link (A EACHLEFT EACHRIGHT pair B)

D18 *cart A* **for**
single A **if** *tally A = 0*
shape A EACHRIGHT reshape EACH hitch (first A &cart cart rest A) **otherwise**

D19 Let *A* be a Boolean value. Then
&not A **for** *l* **if** *A = o*
o **if** *A = l*

D20 Let *A* and *B* be Boolean values. Then
A &and B **for** *l* **if** *A = l* **and** *B = l*
o **otherwise**

D21 Let *A* and *B* be Boolean values. Then
A &or B **for** *o* **if** *A = o* **and** *B = o*
l **otherwise**

D22 Let *A* be an array with Boolean values as items.
&and A **for** *l* **if** *tally A = 0*
first A &and &and rest A **otherwise**

D23 Let *A* be an array with Boolean values as items.
&or A **for** *o* **if** *tally A = 0*
first A &or &or rest A **otherwise**

D24 Let *A* be an array with Boolean values as items and let *B* an array with *tally A = tally B*. We define
A &sublist B **for** *Null* **if** *tally B = 0*
first B hitch (rest A &sublist rest B) **if** *first A = l*
rest A &sublist rest B **if** *first A = o*

D25 *A in B* **for** *&or (A EACHRIGHT equal B)*

D26 Let *N* be a nonnegative integer. We define
&tell N **for** *Null* **if** *N = 0*
0 hitch EACH &succ &tell &pred N **otherwise**

D27 Let A be a nonnegative integer or a list of nonnegative integer. Then
tell A **for** *&tell A* **if** *A* is a nonnegative integer

cart EACH &tell A **if** *A* is a list of nonnegative integers

D28 *suit A* **for** *single first A* **if** *tally A = 1*
list A **otherwise**

D29 *simple A* **for** *A = EACH single A*

D30 *grid A* **for** *tell suit shape A*

D31 *A findall B* **for** *A EACHRIGHT equal B &sublist grid B*

D32 *A find B* **for** *first (A findall B)* **if** *A in B*
suit shape B **otherwise**

D33 *I &picklist A* **for** *first A* **if** *I = 0*
&pred I &picklist rest A **if** *0 < I < tally A*
?address **otherwise**

D34 *I &atoi A* **for** *I find list tell suit A*

D35 *I pick A* **for** *suit I &atoi shape A &picklist A*

D36 *I &itoa A* **for** *I &picklist tell suit A*

D37 *A choose B* **for** *A EACHLEFT pick B*

D38 *reverse* is the operation that satisfies:
shape reverse A = shape A
I pick reverse A = &pred tally A - (I &atoi shape A) &picklist A **if** *I in grid A*

Axioms

A0 *shape A reshape (first A hitch rest A) = A*

A1 *first (A hitch list B) = A*

A2 *rest (A hitch list B) = list B*

A3 *list (A hitch B) = A hitch B*

A4 *list rest A = rest A*

A5 *list shape A = shape A*

A6 *first list A = first A*

A7 *rest list A = rest A*

A8 *A hitch list B = A hitch B*

A9 *A reshape list B = A reshape B*

A10 *list A reshape B = A reshape B*

A11 *shape A reshape list A = A*

A12 *shape (shape A reshape B) = shape A*

A13 Either *list A = Null* or ∃ arrays *B, C* such that *list A = B hitch list C*

A14 *tally Null = 0*

A15 *tally (A hitch B) = &succ tally B*

A16 *rest Null = Null*

A17 *shape list A = solitary tally A*

A18 *list tally A = solitary tally A*

A19 *tally A = &prod shape A*

A20 **if** *&prod shape A > 0* **and** *tally B > 0* **then** *first (shape A reshape B) = first B*

A21 **if** *&prod shape A = tally B* **then** *rest (shape A reshape B) = rest B*

A22 **if** *&prod shape A ≤ &prod shape B* **then**

shape A reshape (shape B reshape C) = shape A reshape C

A23 *A hitch B ≠ A*

A24 *first Null = ?address*

A25 *&sum link EACH EACH tally A = &sum EACH &sum EACH EACH tally A*

A26 *&prod link EACH EACH tally A = &prod EACH &prod EACH EACH tally A*

A27 *A equal B = l* **if** *A = B*
o **if** *A ≠ B*

A28 *atomic equal A*

A29 *equal A = l* **if** *tally A ≤ 1*
first A equal second A &and equal rest rest A **otherwise**

A30 *simple shape A*

A31 **if** *A in B* **then** *A find B pick B = A*

A32 **if** *0 ≤ I < tally A* **then** *I pick list grid A find list grid A = A*

A33 The items of *list grid A* are in lexicographical order.

Theorems

T0 *A = B* **iff** (*shape A = shape B* **and** *first A = first B* **and** *rest A = rest B*)

T1 *list list A = list A*

T2 *shape A reshape A = A*

T3 *tally list A = tally A*

T4 *list Null = Null*

T5 *tally rest A = &pred tally A*

T6 *tally A* is a nonnegative integer.

T7 *tally solitary A = 1*

T8 *first solitary A = A*

T9 *list solitary A = solitary A*

T10 *tally A reshape A = list A*

T11 **if** *tally A = 0* **then** *list A = Null* **else** *list A = first A hitch rest A*

T12 *valence list A = 1*

T13 *first tally A = tally A*

T14 *rest tally A = Null*

T15 *first single A = A*

T16 *single tally A = tally A*

T17 *shape single A = Null*

T18 *tally single A = 1*

Cor. to T18 *tally tally A = 1*

T19 *solitary A = list single A*

T20 *rest single A = Null*

T21 *1 reshape single A = solitary A*

T22 *tally A ≠ Null*

Cor. to T22 *0 ≠ Null*

T23 *A hitch B ≠ B*

T24 **if** *&prod shape A = 0* **then** *shape A reshape B = shape A reshape Null*

T25 *A = B* **iff** (*shape A = shape B* **and** *list A = list B)*

T26 *EACH f A = shape A reshape (f first A hitch EACH f rest A)*

T27 *shape (EACH f A) = shape A*

T28 *tally EACH f A = tally A*

T29 *EACH f Null = Null*

T30 *EACH f list A = list EACH f A*

T31 *EACH f (A hitch B) = f A hitch EACH f B*

T32 *EACH f rest A = rest EACH f A*

T33 *EACH f single A = single f A*

T34 *EACH f A = shape A reshape EACH f list A*

T35 **if** *tally A > 0* **then** *first EACH f A = f first A*

T36 **if** *tally A > 0* **then** *EACH f list A = f first A hitch EACH f rest A*

T37 **if** *&prod shape A = tally B* **then**
EACH f (shape A reshape B) = shape A reshape EACH f B

T38 Let F be a formula where *A* is not free. **If** for every array A,
if *list A = Null* **then** F(*list A*), **and**
if *tally A > 0* **and** F(*rest A*) **then** F(*list A*),
then for every array *A*, F(*list A*).

T39 Let *f* and *g* be total operations. Then
EACH (f g) A = (EACH f) (EACH g) A

T40 *A EACHRIGHT f B = EACH (A f) B*

T41 *A EACHLEFT f B = EACH (B CONVERSE f) A*

T42 *list (A &link B) = (A &link B)*

T43 *A &link B = list A &link B = A &link list B*

T44 *tally (A &link B) = tally A + tally B*

T45 *A &link Null = list A*

T46 *list link A = link A*

T47 *link list A = link A*

T48 *link EACH list A = link A*

T49 *tally link A = &sum EACH tally A*

T50 *EACH f (A &link B) = EACH f A &link EACH f B*

T51 *EACH f link A = link EACH EACH f A*

T52 *tally link EACH link A = tally link link A*

T53 *link EACH link A = link link A*

T54 *tally (A pair B) = 2*

T55 *first (A pair B) = A*

T56 *rest (A pair B) = solitary B*

T57 *second (A pair B) = B*

T58 *A pair B = A hitch single B*

T59 *A link B = A &link B*

T60 *A hitch B = single A link B*

T61 *A pair B = single A link single B*

T62 *shape (A &cart B) = shape A link shape B*

T63 *shape cart A = link EACH shape A*

T64 *tally cart A = &prod EACH tally A*

T65 *cart Null = single Null*

T66 *cart single A = EACH single A*

Cor. to T66 *cart solitary A = EACH solitary A*

T67 *cart list A = EACH list cart A*

T68 **if** *tally cart A > 0* **then** *first cart A = EACH first A*

T69 *A cart B = A &cart B*

T70 *EACH EACH f cart A = cart EACH EACH f A*

T71 *tally cart link A = tally EACH link cart EACH cart A*

T72 *cart link A = EACH link cart EACH cart A*

T73 *equal A = equal list A*

T74 **if** *A* is an array with Boolean items **then** *&or (l hitch A) = l*

T75 **if** *tally A = tally B* **and** *first A = l* **then** *first (A &sublist B) = first B*

T76 *A in B = A in list B*

T77 **if** *tally A > 0* **then** *first A in A*

T78 **if** *tally A = tally B* **and** *&or A* **then** *first (A &sublist B) in list B*

T79 **if** *N* is a nonnegative integer **then** *tally &tell N = N*

T80 **if** *N* is a nonnegative integer **then** *list &tell N = &tell N*

T81 **if** *N* is a nonnegative integer **then** *shape &tell N = list N*

T82 **if** *tally A > 0* **then** *first tell tally A = 0*

T83 *link suit A = link A*

T84 *EACH f suit A = suit EACH f A*

T85 **if** *simple A* **then** *link A = list A*

T86 *grid list A = tell tally A*

T87 *shape grid A = shape A*

T88 *grid single A = single Null*

T89 **if** *tally A > 0* **then** *first A find list A = 0*

T90 **if** *A in B* **then** *A find B in grid B*

T91 *I &picklist A = I &picklist list A*

T92 *0 &picklist A = first A*

T93 *1 &picklist A = second A*

T94 **if** *0 ≤ I < tally A* **then** *I &picklist A in A*

T95 **if** *0 ≤ I < tally A* **then** *I &picklist EACH f A = f (I &picklist A)*

T96 *list A = list B* **iff**
tally A = tally B, and
I &picklist A = I &picklist B, for every *I in grid list A*

T97 **if** *0 ≤ I < tally A* **then** *I &picklist tell tally A = I*

T98 **if** *I in grid A* **then** *I &atoi shape A in grid list A*

T99 **if** *0 ≤ I < tally A* **then** *I find tell tally A = I*

T100 **if** *0 ≤ I < tally A* **then** *I pick list A = I &picklist A*

T101 **if** *I in grid A* **then** *I pick EACH f A = f (I pick A)*

T102 **if** *I in grid list A* **then** *I &itoa shape A in grid A*

T103 **if** *I in grid list A* **then** *(I &itoa shape A) &atoi shape A = I*

T104 **if** *suit I in grid A* **then** *I pick grid A = suit I*

T105 *A = B* **iff**
shape A = shape B, **and**
I pick A = I pick B for every *I in grid A*

T106 *grid A choose A = A*

T107 *reverse reverse A = A*

Chapter 7 Using the V4 Shell

A special version of Q'Nial has been constructed to permit experimentation with array theory and related topics. It exists in both a Console version and as a Q'Nial for Windows version. The latter is intended for development and experimentation, whereas the former can be used to do test runs and to obtain log files with examples shown in boxed characters.

The V4 Shell is like a normal version of Q'Nial except that it has all the array theoretic operations renamed to begin with an underscore. For example, *_first* implements the V4 operation *first*. The V4 semantics for array theory were chosen for the shell because they provide the most general set of arrays that have been considered in various versions of array theory and Nial. It is possible to simulate more restrictive universes of arrays by limiting the values that can be produced. The V4 Shell also supports the full expression syntax described in Chapter 3.

The instructions for using the graphical interface for the Windows version of the shell as the same as those for the Q'Nial for Windows product. Part 3 of the Nial Tools Manual [JeJe98] gives an overview of its usage, and the Help capability of the Windows version of the V4 Shell also provides assistance. (Note that the Help facility documents the V6 array theory operations and so should be used for assistance on systems operations and interface features and not to look up V4 array theoretic properties.)

The V4 Shell is designed to permit experimental study of array theory and related topics. We have used it to validate experimentally the theory presented in Chapter 6 and to study various choices in developing definition sequences for V4 and V6 Array Theory.

In this chapter we give details of the use of the V4 Shell in validating an executable theory and we discuss how it can be used to achieve effective implementation of an array theory variant. In later Chapters we present some of the definition sequences that have been studied.

The V4 Shell is distributed by NIAL Systems limited with the folders discussed in the manuscript provided as Nial Projects.

V4 objects

The V4 Shell supports the arrays of V4 array theory. That is, arrays are multidimensional collections of nested arrays, with empty arrays that can hide any other array as the virtual item. This is the most general form that arrays can take in the array theories that have been proposed and hence, it provides a basis for studying theories that have the same arrays or more restricted ones.

The V4 Shell includes all the predefined objects of the V4 Q'Nial, but has renamed the ones that correspond to pre-defined array-theoretic objects. Only the identifier names are provided, and an underscore character precedes these. Thus, +, *sum* and *plus* have been

replaced by *_sum* and *_plus*. This permits the user to give definitions for +, *sum* and *plus* that correspond to the experiment being performed.

The system operations, such as *execute*, *write*, and *loaddefs* have been left with their names unmodified.

Experimenting with array theory

There are many ways that the V4 Shell can be used in experimenting with array theoretic concepts. One is to do a detailed study of a version of array theory. We present such a use in the example described in the next section.

Another use is in the development of a definition sequence for a partial theory starting with a certain set of primitives. Peter Falster has been investigating a number of such sequences for expository purposes. Chapter 10 provides a snapshot of his work in this area.

A third use is in developing an effective variant of Q'Nial that supports a modified theory. For example, consider an array theory similar to V4, which handles the pervasive operations in a modified manner. A set of definitions could be provided that utilize the existing implementation where possible, but construct the pervasive operations using ACROSS and DOWN as documented in Chapter 5 to get modified definitions that are reasonably efficient. We conclude this chapter with a discussion on how an effective variant could be achieved.

Validating a theory experimentally

We describe in this section how we have used the V4 Shell to validate the V6 theory presented in Chapter 6. The validation experiment is organized as a Nial Project with a project file and a set of definition files stored in a single folder called *V6inV4*. This folder is distributed with the V4 Shell as an example of its use.

In using the V4 shell for a validation experiment, it is important to adopt some naming conventions to assist in keeping track of the related objects that are in play. The following conventions have been adopted in *V6inV4*.

Meta-definitions are given names that both begin and end with an underscore. For example, *_and_* is used as the name of the logical connective in Boolean expressions that state conditions in definitions, axioms and theorems.

One exception to this naming convention is that the symbol "=" is used to denote meta-equality in the theory. This choice was made since meta-equality is used so frequently and its use adds to the readability of the definitions that state axioms or theorems.

Primitive objects are named as indicated in the target theory, but are defined using the names from the V4 Shell. Thus, in the example theory, the operation *shape*, which always returns a list in V6 array theory, is defined by

shape IS _list _shape

Defined objects are named as indicated in the target theory, but are defined using only meta-notation, primitive objects for the target theory, or objects defined earlier in the sequence.

An axiom or a theorem is stated as a defined object that tests the appropriate property of the primitive and defined objects as described in the axiom or theorem. The definition of an axiom or theorem is expressed in terms of meta-definitions, the names in the target theory and Nial control constructs. The latter are used to avoid evaluation of expressions where the value will be ignored. For example, a theorem in the form

if *Expr1* **then** *Expr2*

where *Expr1* and *Expr2* are arbitrary Boolean expressions, can be written as

IF Expr1 THEN Expr2 ELSE l ENDIF

which avoids evaluating *Expr2* when the condition fails.

In adding program code to do validation one must also avoid using the target language objects since they might not achieve the expected semantics. Thus, such code should be written using the predefined names provided with the V4 Shell.

The folder *V6inV4*

A brief description is given for each of the files provided in the folder. Where it seems helpful some additional commentary is given to point out some of the problems that have been encountered in using the V4 Shell on this example.

The theory files:

meta.ndf is the file defines the objects of the meta language. In the example, it gives definitions of operations for testing equality, Boolean operations and some domain testing operations.

The equality operation has to test equality of objects in the target theory. Unless the target theory contains all V4 arrays, the test should be explicitly programmed to descend through all levels. For V6 arrays, equality is tested by

```
_eq_ IS OP A B {
    IF _shape A _equal _shape B THEN
```

```
    IF _tally A _equal 0 THEN
      1
    ELSEIF _and _EACH _atomic A B THEN
      A _equal B
    ELSE
     _and (A _EACHBOTH _eq_ B)
    ENDIF
  ELSE
    o
  ENDIF }
```

The meta operation *_eq_* is renamed by

```
= IS _eq_
```

The Boolean connectives should be written so that the result is guaranteed to be a Boolean value in order to avoid having a condition in an axiom or theorem evaluate to a non-Boolean value. For example, *_and_* is defined by

```
 _and_ IS OP A B { (A _equal 1) _and (B _equal 1) }
```

primitives.ndf is the file that gives the definitions of the primitives chosen for the development.

Provided the target theory has data objects that are a subset of the data objects of V4 array theory, the data objects can be directly represented. Otherwise a representation has to be chosen, and all the primitives must take such representations as arguments and return them as results. In the latter case, an operation to convert the representation to the desired picture of the data object is needed to make it easier to visually check that definitions produce the expected result.

The implemented primitives must ensure that result is a data object in the target theory. For example, since the operation *rest* can create an arbitrary empty list in V4 array theory it must be defined to avoid this for V6.

```
rest IS OP A {  IF _tally A _lte 1 THEN _Null ELSE _rest A  ENDIF }
```

definitions.ndf is the file where the definitions of the theory are presented in the order that they are needed to achieve the expected semantics. In Chapter 6, definitions were stated as substitutions using a form

A **for** *B*

where *A* and *B* are array expressions. The definitions have been recast as Nial definitions of the appropriate functional order. For example the definition D15

A pair B **for** *A hitch solitary B*

becomes

```
pair IS OP A B { A hitch solitary B }
```

axioms.ndf, and **theorems.ndf** are the files of axioms and theorems of the V6 theory expressed as definitions. Each definition is expected to evaluate to *Truth* for every invocation.

If the axiom or theorem is a fact about constant arrays it is stated as a named expression. If it is a fact about operations in the theory that holds for all arrays it is stated as a named operation. If it is a fact about transformers in the theory that holds for all operations and all arrays then it is stated as a named transformer. Here are examples of each order.

```
A14 IS { tally Null = 0 }

A17 IS OP A { shape list A = solitary tally A }

T30 IS TR f OP A { EACH f list A = list EACH f A }
```

The axioms and theorems are almost direct copies of the statements in the mathematical paper. In some cases the syntactic dot is used after "=" to avoid putting parentheses since "=" is being used as a Nial operation and hence only applies to the simple expression on the right. The other change that is made is that most implications have been replaced by an *if-then-else* construct with *1* in the *else* part.

A validation experiment involves testing that the definitions behave as expected and that the stated axioms and theorems evaluate to *1* for all values that are tried. In order to test the V6 theory, we have developed a testing facility that systematically uses predefined arrays and randomly generated arrays to provide a substantial validation suite.

The testing with predefined arrays is controlled by the file **datatest.ndf**. It has routines to construct a variety of arrays from given lists of values, and a routine *testid* that runs a sequence of tests against an operation expression provided as a string or phrase. It is used with a specific name of an axiom or theorem stated as an operation, such as "*A17* above, or it is used with a transform expression such as *'T30 rest'* for testing an axiom or theorem stated as a transformer.

The testing with randomly generated arrays is controlled by the file **randtest.ndf**. It has a routine *randarr*, which recursively generates arrays with a limited extents, valence and depth. The routine *randtest* runs a sequence of tests on randomly generated arrays.

The actual tests run are controlled by **testing.ndf** and **tests.ndf**. The routine *dotests* uses a list of tests entries supplied as a parameter to run both the data driven and random testing. The number of random tests done and the seed random value are controlled by the other parameters to *dotests*. The file tests.ndf contains a number of lists of tests, so that the testing work is split into reasonable length tasks.

The folder *V6inV4* also contains the following files:

main.ndf is the script calls to *loaddefs* that loads the definition files in the correct order. It is the file that you must load manually when you start up the project. This can be achieved by right-clicking on its window and selecting the menu item *Loaddefs main.ndf*. The file **testing.ndf** is loaded first to ensure that it does not use any of the operations defined by the remaining files.

v6inv4.npj is the project file for the experiment. It records the information on the use of the V4 Shell so that at startup the window arrangement saved during a previous use of the V4 Shell for this project will be regenerated. To work with the project, start up the V4 Shell from the Windows Start menu or a shortcut, and then open the project file using Open under the Project menu. If you have used the project recently, it will be named in the lower part of the Project menu and can be selected directly.

eachdefs.ndf is a file containing more efficient definitions of *EACH*, *EACHLEFT* and *EACHRIGHT*. In doing the testing of the theorems T40 and above, it is convenient to load this file in order to speed up the execution of the remaining tests. If all the axioms and all the theorems below T40 have been validated, then these transformers behave as expected in the theory. The faster definitions take advantage of the fact that *_EACH* in the V4 Shell behaves exactly the same as *EACH* in V6 when an *EACH* transform is applied to a nonempty array.

V6inV4 Notes is a text file giving a brief overview of the experiment.

Achieving an effective array theory variant using the V4 Shell

The V4 Shell supports the development of a definition sequence that corresponds to a variant of array theory. However, because of the heavy layering of definitions, the variant may prove to be too slow for practical problem solving. In this section, we discuss how the problem of slow execution can be made less severe.

The most obvious way to speed up the variant is to identify the array theory operations and transformers that behave the same as their counterpart in V4 array theory. If one is defining a variant of V4 then many of the defined operations will correspond to the V4 counterpart. For all such objects, you can replace the definition in the sequence with a renaming of the V4 object. For example, if *reshape* is unchanged in the variant then define it by

```
reshape IS _reshape
```

The next speedup involves identifying the objects that are largely the same as the V4 counterpart, but require different handling in certain cases. The speedup used for the *EACH* family of transformers in testing the V6 theory is such an example.

```
EACH IS TR f OP A {
     IF _tally A = 0 THEN
       shape A reshape Null
     ELSE
       _EACH f A
```

```
  ENDIF }
```

Here the difference in the two definitions is quite small and the test to determine when to use the V4 definition can be made efficiently.

A more complex example is given by the pervasive operations. As noted in Chapter 5, the transformers *ACROSS* and *DOWN* can be used to achieve the pervasive operations starting with the definition for the atomic case.

We can express the property of the unary pervasive operation *abs* by

```
abs IS  DOWN [ atomic, &abs, pass, pass ]
```

Suppose that *&abs* is defined

```
&abs IS OP A {
   IF numeric A THEN
       _abs A
   ELSE
       A
   ENDIF }
```

This definition leaves the meaning of *abs* the same as V4 for numeric atoms, but changes it for literal atoms. The implementation of _abs in V4 explicitly handles the case where the argument is a homogeneous list of numeric atoms and hence on such an array _abs will be faster than the definition using *DOWN*.

Thus the above definition can be made more effective by testing for the special case. For example, the new definition becomes

```
abs IS OP A {
   IF allnumeric A THEN
       _abs A
   ELSE
      DOWN [ atomic, &abs, pass, pass ]
   ENDIF }
```

However, this only speeds up the case where *A* passes the test. If *A* is a list of all numeric arrays, then *DOWN* would still descend to the atomic level.

This can be avoided by the following alternative definition

```
abs IS DOWN [simple, FORK [allnumeric, _abs, EACH &abs], pass, pass]
```

Here, *DOWN* descends until a simple array is encountered and then the *FORK* transform is used to either apply the V4 routine to it or to use *EACH &abs*.

A similar situation holds for the other forms of pervasive operations. The definition of *minus* given in Chapter 5 as

```
minus IS DOWN [simple, &minus, pack, pass]
```

is replaced by

```
minus IS DOWN [and EACH simple,
     FORK [and EACH allnumeric, _minus, EACH &minus pack], pack, pass]
```

Here, *DOWN* descends until both items of the pair are simple arrays. The *FORK* transform applies the efficient V4 routine on a pair of all numeric arrays, but uses *EACH &minus pack* in the other cases. The above code assumes that *_minus* handles non-conforming arrays as intended in the variant being defined. If this is not the case, then

```
minus IS DOWN [and EACH simple,
     FORK [and link [EACH allnumeric, &conforms], _minus,
           EACH &minus pack], pack, pass]
```

where *&conforms* ensures that the pair of arrays will be processed correctly by *_minus*. This additional test would be needed if the variant uses a definition for *pack* that differs from that used in V4.

The operation *sum* defined in Chapter 5 as

```
sum IS DOWN [simple, ACROSS [0 first, pass, &plus], pack, pass]
```

can be optimized in two ways. First, if *A* is allnumeric the direct summation with *_sum* will be faster. This suggests the definition

```
sum IS DOWN [simple,
      FORK [allnumeric, _sum, ACROSS [0 first, pass, &plus]],
      pack, pass]
```

If A is a list of allnumeric arrays that conform then the summation by *_sum* in V4 avoids packing the arrays and is much faster. The following definition achieves both of these speedups.

```
sum IS  FORK [ allnumeric, _sum,
  DOWN [ and EACH simple, FORK [and EACH allnumeric , _sum,
              EACH ACROSS [ 0 first, pass, &plus ] pack ], pack, pass]]
```

Notice that the first speedup is programmed separately. While, it would also be caught in the second speedup, the tests inside the DOWN part would force a lot of extra work for a simple numeric array before *_sum* would be used.

Another way that definitions can be speeded up is by looking for equivalent definitions that will do less work. Consider the definition of *depth*. We know that the depth of an

atom is 0 and that of a non-atomic simple array is 1. Thus, we can define *depth* by the slightly more elaborate definition:

depth IS DOWN [_simple, FORK [_atomic, 0 first, 1 first] , pass, 1 plus max]

With this choice, the depth of large flat arrays is computed much more quickly because the computation of the depths of all the atoms in a simple array is avoided. The effective speedup is achieved only if the V4 Shell tests for simple and atomic arrays are used.

Chapter 8 Completing the Definition Sequence for V6

In Chapter 6 we presented a formal development of part of V6 Array Theory. This included the definition of many of the objects of the theory along with axioms and theorems that state the important properties of the objects in the theory.

In this chapter we continue the development of V6 array theory by presenting a definition sequence that defines most of the remaining objects of the theory. The definition sequence has been developed using the V4 Shell and is documented in the folder *V6seq*. We present the definition sequence as groups of related definitions along with a discussion of the design choices that have been made. For some groups of definitions we present equations that hold for the objects.

Additional primitive objects

In order to complete the definition sequence some additional primitive objects are required. These include the type testing predicates: *isboolean*, *isinteger* etc. and the atomic version of all the pervading operations that involve real numbers or literal data.

The file **moreprims.ndf** gives the definitions that are needed. Almost all of these are identical to the behavior of the corresponding V4 operations on atoms and hence are implemented by a simple renaming. For example,

&plus IS _plus.

However, there are two primitives that are treated differently. These are *&lte* and *&lt* used in comparisons. We discuss their design next.

A goal is to have the atoms of array theory form a lattice induced by the comparator *&lte*. For atoms of numeric type (Boolean, integer or real) the comparison is done on the numeric value, where the Boolean values “l” and “o” are treated as the integers “0” and “1”. For character atoms the comparison is based on a the position of the characters in a sorting sequence, and for phrases it is based on the lexicographic ordering of the phrases treated as character strings.

In order to complete the lattice, the special values *?I*, called the *Zenith* and *?O* called the *Nadir*, are treated as the highest and lowest values respectively. In comparing faults, the special cases of the *Nadir* and *Zenith* are treated separately. A pair of faults that are neither of these values are compared lexicographically according to their corresponding string. If two atoms of different type are compared and at least one is literal (character, phrase or fault) then the comparison is done on the basis of their type index as determined by *&typeindex* below. This latter decision is different than the choice in V4 Array Theory, where such comparisons always returned “o”.

In the V4 shell, the treatment of the comparators is similar to the above description except that comparisons with faults do not behave as described. The definitions below reflect the intended semantics making use of *_lte* where appropriate.

```
&lte IS OP A B {
   &typeindex IS OP A {
      _type A _find o 0 0. ` "" ?? };
   Nadir IS _max _Null;
   Zenith IS _min _Null;
   IF A _mate B THEN
     l
   ELSEIF A = Nadir _or_. B = Zenith THEN
     l
   ELSEIF A = Zenith _or_. B = Nadir THEN
     o
   ELSEIF _type A = _type B _or_. _numeric A _and _numeric B THEN
      A _lte B
   ELSE
      &typeindex A _lte &typeindex B
   ENDIF }

&lt IS OP A B {
   IF A _mate B THEN
     o
   ELSE
      A &lte B
   ENDIF }

&gte IS  &lte _reverse

&gt IS &lt _reverse
```

The pervasive operations

The file **pervdefs.ndf** holds the definitions of the pervasive operations of the theory. They are built up from the recursive transformers as discussed in Chapter 5. The file begins with the definitions of the identity operation *pass*, and the predicates *unequal* and *empty*.

```
pass IS OPERATION A ( A )

unequal IS OPERATION A { &not equal A }

empty IS OPERATION A { tally A equal 0 }
```

The next set of definitions introduce *FORK* as a selection transformer, and the three recursion transformers *RECUR*, *ACROSS* and *DOWN* of Chapter 5. The implemented version of *FORK* in Q'Nial is more general than the one given here in that it allows more than three operation arguments and corresponds to the general form of *if-then-else* expressions.

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

```
RECUR IS TRANSFORMER test endf parta joinf partb
  (FORK [test, endf,
         joinf [parta, RECUR [test, endf, parta, joinf, partb] partb] ]
)

ACROSS IS TRANSFORMER endf parta joinf (
       RECUR [empty, endf, parta first, joinf, rest] list)

DOWN IS TRANSFORMER test endf structf joinf (
    FORK [test, endf, joinf EACH (DOWN [test, endf, structf, joinf])
structf] )
```

The unary pervasive operations are all based on descending an array to the atomic leaves and then applying the atomic version of the operation. The descent is controlled by the transformer *LEAF*, defined by

```
LEAF IS TRANSFORMER f ( DOWN [atomic, f, pass, pass] )
```

Each unary pervasive operation is then defined from the corresponding operation on an atom.

```
abs IS LEAF &abs

not IS LEAF &not

type IS LEAF &type

floor IS LEAF &floor

opposite IS LEAF (0 &minus)

reciprocal IS LEAF (1 &divide)
```

In the above definitions we could have taken *&opposite* as primitive and defined *&minus* from it using

```
&minus IS OPERATION A B { A &plus &opposite B }.
```

Similarly, we could have chosen *&reciprocal* and defined *&divide*.

We have omitted the definition of the scientific and trigonometric unary operations since they are all extended in the same manner from their atomic versions using *LEAF*.

The definition of the remaining pervasive operations need the operation *pack*, whose definition in turn needs to be able to compute the maximum of a list of integers. We define the following two preliminary versions of *max*.

```
&max IS OP A B {
  IF A &gte B THEN
     A
  ELSE
     B
```

```
  ENDIF }

&&max IS ACROSS [??O first, pass, &max]
```

The definition of *pack* requires a number of design choices. Its role in the definition of pervasive functions is to bring corresponding items of items together. For example,

```
pack [2 3 4, 8 9 10]
ÚÄÄÄÂÄÄÄÂÄÄÄ¿
³2 8³3 9³4 10³
ÀÄÄÄÁÄÄÄÁÄÄÄÙ
```

If the argument to *pack* has items of equal shape (is **equishaped**), then the effect is to interchange the top two levels of the array, which can be done by

grid first A EACHLEFT EACHRIGHT pick A.

However, if *A* is not equishaped then decisions have to be made. If an item of *A* is an atom then it is appropriate to replicate it in *pack*. What should be done in the other cases? In V6.21 Nial, it was decided to replicate an item of *A* that has exactly one item. Thus an item of *A* that is a single or a solitary is replicated.

If there are two or more items of *A* of different shape each with other than one item, then the fault *?conform* is returned by *pack*. The assumption is that such a situation is likely due to a programming error and returning a fault will assist in debugging. These decisions are a change from the V4 semantics of *pack* where only singles are replicated and an operation *trim* is used to construct an equishaped array otherwise.

In the V6 version, there is also the decision of what valence to use if all the items of *A* have one item. The maximum valence is used to avoid loss of information. (The same decision was used in the corresponding feature in APL.) These decisions are expressed in the following definition of *pack*.

```
pack IS OPERATION A {
  Shps := EACH shape A;
  Pattern := EACH tally A EACHLEFT unequal 1;
  Shps1 := Pattern &sublist Shps;
  IF unequal Shps1 THEN
     ??conform
  ELSE
     IF empty Shps1 THEN
       J := &&max EACH valence A find EACH valence A;
       Shp := J pick Shps;
     ELSE
       Shp := first Shps1;
     ENDIF;
     B := Shp EACHRIGHT reshape A;
     grid first B EACHLEFT EACHRIGHT pick B
  ENDIF }
```

Before continuing we introduce the definition of *&min*. The above definition of *&max* has the property that if exactly one of *A* and *B* is a fault then the result is that fault unless it is the *Nadir*. This is a consequence of using the type index to determine the result of *&lte* if different types are compared. If we make the corresponding definition for *&min* using *&lte*, then, if exactly one of *A* and *B* is a fault, then the fault is not returned unless it is the *Nadir*. In order to get the desired fault behavior, we must explicitly program it into *&min*. We define *&min* by:

```
&min IS OP A B {
  Zenith := ??I;
  IF A equal Zenith THEN
     B
  ELSEIF B equal Zenith THEN
     A
  ELSEIF isfault A &and &not isfault B THEN
   A
 ELSEIF isfault B &and &not isfault A THEN
   B
 ELSEIF A &lte B THEN
     A
  ELSE
     B
  ENDIF }
```

The consequence of this definition is that the predicate on simple arrays

and (min A EACHLEFT lte A)

fails for some simple arrays where some of the items are faults. The corresponding predicate for *&max*

and (max A EACHLEFT gte A)

holds for all simple arrays.

The binary pervasive operations and multi-pervasive operations are all based on using *pack* to restructure the argument while descending a simple array then applying the atomic version of the operation. The descent to simple arrays is controlled by the transformer *DIG* defined by:

```
DIG IS TRANSFORMER f ( DOWN [simple, f, pack, pass] )
```

Each binary pervasive operation is then defined from its corresponding one on atomic pairs. We give the definition for *plus*. The definitions for the remaining binary pervasive operations have the same pattern. We omit them here for brevity. In the definitions a test is made that the operation is being applied to a pair in order to catch accidental misuses.

```
plus IS OPERATION A {
  IF tally A equal 2 THEN
     DIG &plus A
  ELSE
     fault '?plus expects a pair'
  ENDIF }
```

The multi-pervasive operations are reductive as well as pervasive. The reductive effect is achieved using ACROSS with corresponding binary operation as the joining operation. Thus the definitions are:

```
sum IS DIG ACROSS [0 first, pass, &plus]

product IS DIG ACROSS [1 first, pass, &times]

min IS DIG ACROSS [??I first, pass, &min]

max IS DIG ACROSS [??O first, pass, &max]

and IS DIG ACROSS [l first, pass, &and]

or IS DIG ACROSS [o first, pass, &or]
```

The three types of pervasive operations obey recursive equations involving the distributive transformers. We define

```
EACHBOTH IS TRANSFORMER f OPERATION A B { EACH f pack A B }

EACHALL IS TRANSFORMER f (EACH f pack)
```

The recursive equations are:

Unary Pervasive *f*: $f\,A = EACH\,f\,A$

Binary Pervasive *f*: $A\,f\,B = A\;EACHBOTH\,f\,B$

Multi-Pervasive *f*: $f\,A = EACHALL\,f\,A$

Since *EACHBOTH* is a special case of *EACHALL*, the binary pervasive operations also obey the third equation.

Selection and structure operations

In this section we present the definition of many of the selection and restructuring operations not defined in Chapter 6. Many of the definitions have tests to check for valid arguments in order to assist in debugging.

We begin by extending *&sublist* to the definition of *sublist* used in Version 6 Array Theory.

```
sublist IS OPERATION Arg {
  IF tally Arg equal 2 THEN
    A B := Arg;
    IF and EACH isboolean A THEN
       IF tally A gt 0 THEN
          tally B reshape A &sublist list B
```

```
      ELSEIF tally B = 0 THEN
         Null
      ELSE
         fault '?first arg of sublist is empty'
      ENDIF
    ELSE
      fault '?first arg of sublist not boolean'
    ENDIF
  ELSE
    fault '?argument of sublist must be a pair'
  ENDIF
}
```

The following equations hold for *sublist* provided *A* has Boolean items and *B* is not empty.

tally (A sublist B) = sum (tally B reshape A)
A sublist B = A sublist list B
l sublist B = list B
o sublist B = Null

The next three operations are based on *pick*. The third one is a recursive operation used to select an array along a path into a given array *A*.

```
third IS OP A { 2 pick list A }

last IS OPERATION A {tally A minus 1 pick list A}

reach IS OP Arg {
  IF tally Arg equal 2 THEN
     Path A := Arg;
     IF empty Path THEN
        A
     ELSEIF suit first Path in grid A THEN
        rest Path reach (first Path pick A)
     ELSE
        ??path
     ENDIF
  ELSE
     fault '?argument of reach must be a pair'
  ENDIF }
```

This completes the definition of the operations based on *pick*. The following additional equations for *pick* and *choose* hold provided their left argument is in range.

*0 * shape A pick A = first A*
I pick A = list I pick A
I choose (J choose A) = (I choose J) choose A
I pick (J choose A) = (I pick J) pick A

We can also describe the items of *cart A* using *pick*. Let *J* be an address of *A* and *link I* be an address of *cart A*, where *I* has an item for each item of *A* of the same length as the valence of that item. Then

J pick (link I pick cart A) = (J pick I) pick (J pick A)

The next two operations are reshapings of an array. The operation *post* returns the table with one column of the items of the argument. The operation *front* returns all but the last item of A as a list.

```
post IS OPERATION A { [tally A, 1] reshape A }

front IS OPERATION A {
   IF empty A THEN
      list A
   ELSE
      tally A minus 1 reshape A
   ENDIF }
```

The following equations related to *reshape* hold.

solitary A = 1 reshape single A
post post A = post A
pair A = 2 reshape A
pair pair A = pair A
if *not empty A* **then** *Null reshape A = single first A*
0 reshape A = Null

We now define some operations related to *link*. The operation *content* is a recursive version of *link* that gathers the leaves of an array as a list. The operation *append* is analogous to *hitch*; its selectors are *front* and *last*.

```
content IS OPERATION A {
  IF simple A THEN
     list A
  ELSE
     link EACH content A
  ENDIF }

append IS OP A B { A link single B }
```

The following equations involving *link*, *content* or *append* hold.

if *and EACH empty A* **then** *link A = Null*
if *and EACH simple A* **then** *link A = content A*
content A = link EACH content A
content A = content list A
list content A = content A
link solitary A = list A
shape A reshape (front A append last A) = A

The next two definitions use *tell* and are given for convenience since these operations are used frequently in practical work.

```
axes IS OPERATION A { tell valence A }

count IS OPERATION A { 1 plus tell A }
```

The following equations hold for *tell*-related operations.

if *diverse A* **then** *grid A = A EACHLEFT find A*
tell shape A = cart EACH tell shape A
tell Null = single Null
tell tally A choose list A = list A
if *isinteger A* **and** *A >= 0* **then** *tell (A plus 1) = tell A append A*

The operation *in* is analogous to set membership in set theory. Arrays are different from sets in that arrays can contain duplicate items and the order of items in the array is specifically determined. The following operations correspond to set operations at the top level of the array. The correspondences are

notin	not element of
allin	subset
like	set equality (at top level)
except	set difference
cull	remove duplicates
diverse	has no duplicates

```
notin IS OPERATION A B { not (A in B) }

allin IS OPERATION A B { and (A EACHLEFT in B) }

like IS OPERATION A B { A allin B and (B allin A) }

except IS OPERATION Arg {
  IF tally Arg equal 2 THEN
     A B := Arg;
     A EACHLEFT notin B sublist A
  ELSE
     fault '?argument of except must be a pair'
  ENDIF }

cull IS OPERATION A {
        grid A EACHLEFT in (A EACHLEFT find A) sublist A }

diverse IS OPERATION A { cull A equal list A }
```

The two set operations that are missing are those that correspond to set union and set intersection. In array theory, the operation that corresponds to set union is *link*. For set intersection we can define an operation *intersect* by

```
intersect IS OPERATION A {
   IF empty A THEN
```

```
      Null
   ELSE
      EACH and ( first A EACHLEFT EACHRIGHT in A ) sublist first A
   ENDIF }
```

Trenchard More calls this operation *lap*. However, this definition is not included in the predefined set of operations in Q'Nial because of its inherent inefficiency. Many of the above definitions are implemented internally using smart algorithms, and perhaps *intersect* should be added with an appropriate fast algorithm.

The next definition is for the operation *rotate*. It expects the first argument to be an integer and uses it modulo the tally of the second argument to rotate the items of the second argument that many positions to the left if positive or to the right if negative.

```
rotate IS OPERATION N A {
  IF isinteger N THEN
     Ta := tally A;
     shape A reshape (Ta + N + tell Ta mod Ta choose list A)
  ELSE
     fault '?first arg of rotate not an integer'
  ENDIF }
```

Equations for *rotate*, where *N* is an integer, are

shape (N rotate A) = shape A
N rotate A = shape A reshape (N rotate list A)
N rotate A = N mod tally A rotate A
N rotate A = opposite (tally A - N) rotate A

The operations *take* and *drop* select contiguous parts of an array starting from an end of a list or a corner of a higher dimensional array using a list of integers to determine the size of the result along each axis. With *take*, the number indicates the size to be selected, with *drop*, it indicates the amount to discard. If the an item of the first argument is negative then the amount taken or dropped is from the end of the axis rather than the beginning. For *take*, if the size is larger than the extent, then the type of the first item is used to fill the positions past the end. If the second argument is empty the fault *?fill* is used as the fill item.

```
take IS OPERATION Arg {
  &idx IS OP N Lim {
     IF N gte 0 THEN
        tell N
     ELSE
        reverse (Lim minus count abs N)
     ENDIF };
  &overpick IS OP I B {
     IF empty B THEN
       ??fill
     ELSEIF suit I notin grid B THEN
       type first B
     ELSE
```

```
      I pick B
    ENDIF };
  IF tally Arg equal 2 THEN
    A B := Arg;
    IF and EACH isinteger A THEN
      IF tally A equal valence B THEN
        cart EACH &idx (A pack shape B) EACHLEFT &overpick B
      ELSE
        fault '?valence error in take'
      ENDIF
    ELSE
      fault '?left argument in take must be integers'
    ENDIF
  ELSE
    fault '?argument of take must be a pair'
  ENDIF }

drop IS OPERATION Arg {
  &idx IS OP N Lim {
    IF N gte 0 THEN
      N plus tell (Lim minus N max 0)
    ELSE
      tell (Lim minus abs N max 0)
    ENDIF };
  IF tally Arg equal 2 THEN
    A B := Arg;
    IF and EACH isinteger A THEN
      IF tally A equal valence B THEN
        cart EACH &idx (A pack shape B) EACHLEFT pick B
      ELSE
        fault '?valence error in drop'
      ENDIF
    ELSE
      fault '?left argument in drop must be integers'
    ENDIF
  ELSE
    fault '?argument of take must be a pair'
  ENDIF }

takeright IS OPERATION A B { opposite A take B }

dropright IS OPERATION A B { opposite A drop B }
```

Equations for *take* and *drop* are

shape B take B = B
0 times shape B drop B = B
if *A in count shape B* **then** *A take B = (tell A choose B)*

The next pair of definitions are for operations that use a Boolean pattern to split a list into parts that become items of the result. The operation *cutall* includes the items that correspond to the occurrences of the “l”s as the first items of the result items, whereas *cut* omits these items. For both operations, if there are leading “o”s, the corresponding items are made into the first item of the result.

```
cutall IS OPERATION Arg {
  IF tally Arg equal 2 THEN
    A B := Arg;
    IF and EACH isboolean A THEN
       IF tally A gt 0 THEN
          B := list B;
          A := tally B reshape A;
          Starts := 1 findall A;
          Lengths := rest Starts append tally A minus Starts;
          Items := Starts EACHBOTH plus EACH tell Lengths
             EACHLEFT choose B;
          IF first Starts unequal 0 THEN
             Items := tell first Starts choose B hitch Items;
          ENDIF;
          Items
       ELSE
          fault '?first arg of cutall is empty'
       ENDIF
    ELSE
       fault '?first arg of cutall is not boolean'
    ENDIF
  ELSE
    fault '?argument of cutall must be a pair'
  ENDIF }

cut IS OPERATION Arg {
  IF tally Arg equal 2 THEN
    A B := Arg;
    IF and EACH isboolean A THEN
       IF tally A gt 0 THEN
          B := list B;
          A := tally B reshape A;
          Starts := 1 findall A;
          Lengths := rest Starts append tally A minus Starts;
          Items := Starts plus 1 EACHBOTH plus EACH tell
             (Lengths minus 1) EACHLEFT choose B;
          IF first Starts unequal 0 THEN
             Items := tell first Starts choose B hitch Items;
          ENDIF;
          Items
       ELSE
          fault '?first arg of cut is empty'
       ENDIF
    ELSE
       fault '?first arg of cut is not boolean'
    ENDIF
  ELSE
    fault '?argument of cut must be a pair'
  ENDIF }
```

Sorting transformers and operations

The sorting of data is an important capability in data processing. Array theory provides a generic sorting capability as a pair of transformers *SORT* and *GRADE*. They take as their

argument the comparator to be used in doing the ordering. In most cases the comparator *lte* is appropriate for simple data. However, for more complex data structures the user may wish to provide a comparator specific to the application.

In order to provide a total ordering for arrays, the comparator *up* is given below. Internally it is used to provide an ordering to speed up some operations, such as *cull* and *except*, that have quadratic time if done by the formal definition. For atomic arrays *up* corresponds to *lte* except for faults.

We begin by defining *GRADE*, which returns the permutation that reorders the array according to the comparator. The algorithm presented here corresponds to the one used in internally in Q'Nial. A much shorter formal definition could be given, but the one below documents the smart algorithm used for most examples. It is based on the list merge sort algorithm (Algorithm 5.2.4-L) given in [Knuth Vol. 3] improved according to exercise 12 (but not exercise 15). Exercise 12 uses initial runs to speed up the algorithm and hence if *A* is already sorted the algorithm does minimal work. (A radix sort algorithm does sorting of lists of integers and characters when *SORT* or *GRADE* is used with the comparator *lte* in Q'Nial.)

```
GRADE IS TRANSFORMER comp OPERATION A {
    N := tally A;
    Shpa := shape A;
    Va := valence A;
    Ga := list grid A;
    N2 := N + 2;
    Li := N2 reshape 0;
    Li@0 := 1;
    T := N + 1;
    IF N gte 2 THEN
      FOR P WITH count (N minus 1) DO
         IF A@(P minus 1) comp A@P THEN
           Li@P := P plus 1;
         ELSE
           Li@T := opp (P plus 1);
           T := P;
         ENDIF;
      ENDFOR;
      Li@T := 0;
      Li@N := 0;
      Li@(N plus 1) := opp Li@(N plus 1);
      WHILE Li@(N plus 1) unequal 0 DO
        S := 0;
        T := N plus 1;
        Q := Li@T;
        P := Li@S;
        REPEAT
           IF A@(P minus 1) comp A@(Q minus 1) THEN
              Li@S := IF Li@S lt 0 THEN opp P else P ENDIF;
              S := P;
              P := Li@P;
              IF P lte 0 THEN
                 Li@S := Q;
                 S := T;
```

```
                REPEAT
                   T := Q;
                   Q := Li@Q;
                UNTIL Q lte 0 ENDREPEAT;
             ENDIF
          ELSE
             Li@S := IF Li@S lt 0 THEN opp Q else Q ENDIF;
             S := Q;
             Q := Li@Q;
             IF Q lte 0 THEN
                Li@S := P;
                S := T;
                REPEAT
                   T := P;
                   P := Li@P;
                UNTIL P lte 0 ENDREPEAT;
             ENDIF;
          ENDIF;
          IF P lt 0 THEN P := opp P; ENDIF;
          IF Q lt 0 THEN Q := opp Q; ENDIF;
       UNTIL Q = 0 ENDREPEAT;
       Li@S := IF Li@S lt 0 THEN opp P ELSE P ENDIF;
       Li@T := 0;
     ENDWHILE;
  ENDIF;
% second part: reconstitute the result following the links in Li;
  Res := tally A reshape 0;
  T := 0;
  FOR S WITH tell N DO
    T := Li@T;
    Res@S := IF Va equal 1 THEN T minus 1 ELSE T minus 1 pick Ga
ENDIF;
  ENDFOR;
  Shpa reshape Res }
```

The transformer *SORT* is easily defined from *GRADE*.

```
SORT IS TRANSFORMER f OPERATION A { GRADE f A choose A }
```

The comparator *up* provides a strict ordering of arrays. Atomic items are compared by within each type if both are the same type, or by type index otherwise. Non-atomic arrays are compared by a lexicographic comparison, like dictionary sorting, making the decision on the first item that differs. If two arrays have the same items then they the decision is made by comparing their shape with *up*. The comparator is used to define *gradeup* and *sortup*, which are used internally to give fast algorithms for operations such as *cull*, *diverse* and *except*. In addition, arrays that have been sorted are marked internally so that the searching operations *find* and *in* can use binary search.

```
up IS OPERATION A B {
   &typeindex IS OP A {
      _type A _find o 0 0. ` "" ?? };
  IF A equal B THEN
    l
  ELSEIF and EACH atomic A B THEN
     IF type A = type B THEN
```

```
        IF isfault A THEN
           phrase string A lt phrase string B
        ELSE
           A lt B
        ENDIF
     ELSE
        &typeindex A lt &typeindex B
     ENDIF
  ELSE
     Ta Tb := EACH tally A B;
     La Lb := EACH list A B;
     T := min Ta Tb;
     I := 0;
     Cont := l;
     WHILE Cont and (I lt T) DO
        Cont := La@I equal Lb@I;
        IF Cont THEN I := I plus 1 ENDIF;
     ENDWHILE;
     IF not Cont THEN
        La@I up Lb@I
     ELSEIF Ta unequal Tb THEN
        Ta lt Tb
     ELSE
        shape A up shape B
     ENDIF
   ENDIF }

sortup IS SORT up

gradeup IS GRADE up
```

Equations for the sorting operations are

sortup sortup A = sortup A
sortup cull A = cull sortup A
sortup (A except B) = sortup A except sortup B

Axis restructuring operations

The next pair of operations involves axis rearrangements of the items of the array at the same level. The operation *fuse* can be used in two ways. If the first argument is a permutation of the axis numbers, then the axes are rearranged according to that permutation. If it is a nested array then items with two or more items indicate that the diagonal of the corresponding axes should be selected.

The operation *transpose* is the special case of *fuse* in which the axes are reversed.

```
fuse IS OPERATION Arg {
  invpr IS OP I {
    Patterns := tell tally link I EACHLEFT EACHRIGHT in I;
    Patterns EACHLEFT sublist tell tally I except Null };
  IF tally Arg equal 2 THEN
     I A := Arg;
```

```
    IF not (axes A allin link I) or
         (tally link I unequal valence A) THEN
      fault '?left arg of fuse should contain axes of right arg'
    ELSE
      J gets invpr I;
      Shp gets EACH min (I EACHLEFT choose shape A);
      EACH (J choose) tell Shp choose A
    ENDIF
  ELSE
     fault '?argument of fuse must be a pair'
  ENDIF }

transpose IS OPERATION A { reverse axes A fuse A }
```

Equations for these operations are

transpose transpose A = A
shape transpose A = reverse shape A
transpose single A = single A
transpose list A = list A
axes A fuse A = A

The next definitions are for operations that add a level to an array by splitting the axes at the top level. The operations *raise* and *lower* use an integer to indicate how many axes to move; *raise* moves the first *N* axes up, *lower* moves the last *N* axes down. The operation *split* selects an arbitrary set of axes to move down. The operations *rows* and *cols* move the last and second last axes down respectively.

```
raise IS OPERATION Arg {
   IF tally Arg equal 2 THEN
      N A := Arg;
      IF isinteger N and (N gte 0 equal 1) THEN
         Shp Ishp := N [take, drop] shape A;
         Pattern := 1 hitch (product Ishp - 1 reshape 0);
         Items := Ishp EACHRIGHT reshape (Pattern cutall A);
         Shp reshape Items
      ELSE
         fault '?first arg of raise not an axis number'
      ENDIF
   ELSE
      fault '?argument of raise must be a pair'
   ENDIF }

lower IS OPERATION N A {
 IF isinteger N THEN
   IF N lt 0 or (N gt valence A) THEN
     fault '?left arg of lower out of range'
   ELSE
     valence A minus N raise A
   ENDIF
 ELSE
   fault '?left arg of lower not an integer'
 ENDIF }

split IS OPERATION I A {
```

```
   IF empty A THEN
      fault '?empty right arg in split'
   ELSEIF not (I allin axes A &and diverse I) THEN
      fault '?invalid left arg in split'
   ELSE
      J gets axes A except link I;
      tally J raise (J link I fuse A)
   ENDIF }

rows IS OPERATION A { valence A min 1 lower A }


cols IS OPERATION A {
  IF valence A equal 0 or (A equal Null) THEN
    single A
  ELSE
    valence A minus 2 max 0 split A
  ENDIF }
```

The next pair of definitions are for axis restructuring operations that combine the axes at the top two levels. The operation *mix* is the inverse for *raise*, *lower* and *rows* and *blend* is the inverse of *split*.

```
mix IS OPERATION A {
   IF empty A THEN
     shape A append 0 reshape A
   ELSEIF unequal EACH shape A THEN
     ??conform
   ELSE
     shape A link shape first A reshape link A
   ENDIF }

blend IS OPERATION I A {
   IF empty A THEN
      A
   ELSE
     J gets tell (valence A plus valence first A) except I;
     GRADE <= (J link I) fuse mix A
   ENDIF }
```

The axis restructuring operations of the last two sets of definitions obey the following equations provided *A* is not empty, *I* is a subset of the axes of *A*, and *0 <= N <= tally A*.

shape (I split A) = axes A except I choose shape A
shape first (I split A) = I choose shape A
I blend (I split A) = A
mix (N raise A) = A
mix (N lower A) = A
mix rows A = A
shape (N raise A) = I take shape A
shape first (N raise A) = I drop shape A
N raise A = N drop axes A split A

The axis restructuring operations described above, all affect either the top level or the top two levels of the array. Ole Franksen has shown that there is a general operation *pierce* that can be used to define this entire family of operations. See [FrJe92, Frank96] for details.

The final two definitions of this group use the axis restructuring operations to achieve operations that behave like the indexed catenation of APL. The first argument gives an axis position and the second is the list of arrays to be combined.

```
catenate IS OPERATION I A {
   B := EACH ( I split ) A ;
   IF equal EACH shape B THEN
      I blend EACH link pack B
   ELSE
      fault '?conform error in catenate'
   ENDIF }

 laminate IS OPERATION I A {
   IF equal EACH shape A THEN
      Axesofitems := axes first A;
      link (I take Axesofitems) (I drop Axesofitems plus 1) blend A
   ELSE
      fault '?conform error in laminate'
   ENDIF }
```

Additional transformers

We define two transformers that generalize concepts from linear algebra. *OUTER* does a generalized outer product. *INNER* does a general inner product equivalent to both matrix multiplication and vector dot product. The definition of *INNER [f, g]* assumes that *f* is reductive and that *g* distributes across arrays of the same shape.

```
OUTER IS TRANSFORMER f (EACH f cart)

INNER IS TRANSFORMER f g OPERATION A B {
  IF valence A lte 2 THEN
     AA := rows A;
  ELSE
     AA := 1 lower A;
  ENDIF;
  IF valence B lte 2 THEN
     BB := cols B;
  ELSE
     BB := 1 raise B;
  ENDIF;
  EACH f (AA OUTER g BB) }
```

The transformers *RANK*, *BYROWS*, *BYCOLS* and *PARTITION* apply an operation to subarrays of the argument, forming an array from the result. With *RANK*, the subarrays are formed by lowering the last *N* axes. *BYROWS* and *BYCOLS* are special cases of

RANK intended for use on tables. *PARTITION* uses *split* to form the subarrays. For all these transformers it is assumed that *f* produces arrays of the same shape, so that the use of *mix* or *blend* on the array of results succeeds.

```
RANK IS TRANSFORMER f OPERATION N A {
   IF empty A THEN
      A
   ELSEIF not isinteger N THEN
      fault '?left arg to RANK transform not an integer'
   ELSEIF N lt 0 or (N gt valence A) THEN
      fault 'left arg to RANK transform out of range'
   ELSEIF N equal 0 and (valence A equal 0) THEN
      EACH f A
   ELSE
      mix EACH f (N lower A)
   ENDIF }

BYROWS IS TRANSFORMER f OPERATION A {
       1 RANK f A }

BYCOLS IS TRANSFORMER f OPERATION A {
       transpose (1 RANK f transpose A) }

PARTITION IS TRANSFORMER f OPERATION Ij A {
  IF empty Ij or (tally Ij gt 2) THEN
    fault 'invalid left arg of PARTITION transform'
  ELSEIF empty A THEN
    A
  ELSE
    IF tally Ij equal 1 THEN
       I := J := first Ij;
    ELSEIF tally Ij equal 2 THEN
       I J := Ij;
    ENDIF;
    II gets axes A except I link I;
    B := tally I RANK f (II fuse A);
    IF J = Null and (shape B equal Null) THEN
      first B
    ELSEIF tally J equal (valence B minus (valence A minus tally I))
THEN
      JJ gets axes B except J link J;
      GRADE lte JJ fuse B
    ELSE
      fault 'left arg incompatible with function in PARTITION
transform'
    ENDIF
  ENDIF }
```

The major reductive transformers are *REDUCE* and *ACCUMULATE*. There is little need for reduction in array theory since the multi-pervasive operations and *link* have it built in. For these operations *ACCUMULATE* is an important transformer and internally the corresponding transforms are done efficiently taking advantage of the associativity of the reductive operations. *REDUCEROWS* and *REDUCECOLS* are added for convenience of doing reductions on tables.

```
REDUCE IS TRANSFORMER f OPERATION A {
  IF empty A THEN
      ??identity
  ELSE
    Res := last A;
    FOR B WITH reverse front A DO
      Res := B f Res
    ENDFOR
  ENDIF }

ACCUMULATE IS TRANSFORMER f OPERATION A {
  Heads := count tally A EACHLEFT take list A;
  EACH REDUCE f Heads }

REDUCEROWS IS TRANSFORMER f OPERATION A {
   BYROWS (REDUCE f) A }

REDUCECOLS IS TRANSFORMER f OPERATION A {
   [valence A minus 2 max 0, axes first A] PARTITION (REDUCE f) A }
```

The remaining transformers have been provided to capture frequently used idioms. *N FOLD f A* applies *f* to *A N* times. *FILTER f A* uses the predicate *f* to select items from the list of items of *A*. *ITERATE f A* applies the operation *f* to the items of *A* in order, returning the result of the last application. It is useful for issuing a sequence of outputs with operation *writescreen*. *A BYKEY f B* uses *A* as a list of keys to group the items of *B* and then *f* is applied to each of the groups.

```
FOLD IS TRANSFORMER f OPERATION N A {
  IF isinteger N THEN
   IF N > 0 THEN
     N minus 1 FOLD f (f A)
   ELSE
     A
   ENDIF
  ELSE
    fault '?first argument of FOLD must be an integer'
  ENDIF }

FILTER IS TRANSFORMER f OPERATION A { EACH f A sublist A }

ITERATE IS TRANSFORMER f OPERATION A {
          FOR X WITH A DO f X ENDFOR }

BYKEY IS TRANSFORMER f OPERATION A B {
  A := list A ;
  Indices gets gradeup A;
  EACH f (cull A EACHLEFT findall sortup A
       EACHLEFT choose (Indices choose B)) }
```

The V4 Shell model

In order to make the tests of the definitions in the sequence more efficient, the file **defns.ndf** has been rewritten to use faster versions of the objects defined in Chapter 6 by

taking advantage of the V4 operations where possible. The file **newids.ndf** contains definitions that test the sets of equations presented in this chapter. The file **tests.ndf** provides tests for the equations.

In doing the testing several minor problems were noticed in the first attempt at the definitions in this chapter. For example, the definition

E45 IS OP A { axes A fuse A = A }

failed on singles. In looking at the problem, we discovered that it was due to the definition for *except* failing on the case where it reduced to

Null sublist Null.

This was due to the way *sublist* was originally programmed in **moredefns.ndf**.

Doing the exercise of gathering all the definitions for V6 Array Theory objects in one place and attempting to have them work exactly as they do in Q'Nial V6.21 has pointed out a number of minor inconsistencies, especially in the handling of faults. For the operations written at the C level in Q'Nial there is no consistent style for expressing the fault conditions. For the operations written in Nial and stored in the **defs.ndf** file built into Q'Nial, there is considerably less error checking and there is also an inconsistent style for faults.

A more serious problem was encountered with the definition of *rotate* stored in **defs.ndf**. Although the core algorithm is designed to work for both positive and negative integers, when a fault test was added, it ruled out the use of negative integers. This programming error was discovered when definition E34 failed in the testing.

While the testing has validated that the definitions presented in this chapter satisfy the given equations, no check has been done to ensure that the definitions provided in the *V6seq* folder actually match the Q'Nial V6.21 semantics at all the boundary conditions.

A useful next step will be to bring the model built with the V4 Shell and the production version of Q'Nial into complete agreement with cleaned up fault conditions. Then both should be tested rigorously to ensure that they match and satisfy all the equations.

Chapter 9 Towards a Formal Treatment of V4 Array Theory

In Chapter 4, we discussed the fact that there can be many theories of arrays based on the fundamental assumptions we presented. The choices are of two major kinds: the choice of equations that one wants to be universal, and the choices for the detailed semantics of each primitive operation and transformer. Much of chapter 4 is devoted to showing that three desirable universal equations cannot simultaneously hold. One choice, used in Version 4.1 Nial, involves having empty lists with any array as the virtual item. We call this choice V4 Array Theory in this document. The other choice, used in Version 6.21 Nial, has a unique empty list. In both choices, the transformer EACH distributes across operation composition.

In Chapter 6, a formal development of V6 array theory is presented. The purpose of this chapter is to outline how a development like that in Chapter 6 could be done for V4 array theory.

We make the following assumptions in the development:

- there are five structural primitive operations: *first*, *rest*, and *hitch* corresponding to *car*, *cdr*, and *cons* from Lisp, and *shape* and *reshape* from APL,
- an operation *equal* for array equality,
- nonnegative integers, Boolean values and fault values are atomic arrays
- the arithmetic (+, -, *) and comparison (≤, <, ≥, >) operations for the *nonnegative* integers and the successor and predecessor functions (*&succ, &pred*) are primitive,
- there are an arbitrary number of empty lists, and
- shapes are suits.

We use the same meta-syntax as in Chapter 6 to express definitions, axioms and theorems. We proceed by discussing the definitions, axioms and theorems from each section of Chapter 6, indicating what would change, but omitting detailed proofs.

1. Fundamental Axioms and properties from Lisp and APL

The first part of the section is unchanged. A0, T0 and D0 are valid in V4 as stated.

1.1 Axioms corresponding to list properties

In this section A2 is incorrect due to the existence of voids. In order to state it we must define *void* and *Null*.

D0.5 *Null* **for** *0 reshape 0*

D0.6 *void A* **for** *rest (A hitch Null)*

Then we postulate

A2 *rest (A hitch list B)* = *list B* **if** *tally B > 0*
= *void A* **otherwise**

A5 must also be restated due to the change in treatment of *shape*, but this is deferred since we cannot define *suit* at this point in the development. A10 is valid since in V4 array theory *reshape* uses just the content of the left argument.

Theorem T1 is valid since its proof just depends on axioms and definitions that are unchanged.

1.2 Axioms corresponding to APL properties of *shape* and *reshape*

Although *shape* has changed, the axioms and theorem of this section are unaffected.

2.1 Properties of *tally* and *Null*

The definition of *tally* becomes

D1 *tally A* **for** *shape list A*

D2 has been replaced by D0.5, but we must now add the fact that the shape of an atom is *Null* by the axiom

A12.5 *shape tally A = Null*

A13, A14 and A16 are changed to reflect the existence of voids.

A13 **Either** ∃ array *D* such that *list A = void D*, **or**
∃ arrays *B*, *C* such that *list A = B hitch list C*

A14 *tally void A = 0*

A16 *rest void A = void A*

The proof of T5 has to be modified by the change in A13, but goes through in much the same fashion.

A17 is no longer true and there is no need for a replacement since D1 has the corresponding fact. However, A18 is still required.

2.2 The relationship between *shape* and *tally*

In this section theorem T10 is valid, but the proof is simpler. T11 must be restated as

T11 **if** *tally A = 0* **then** *list A = void first A* **else** *list A = first A hitch rest A*

and the proof modified for the void case.

D4 and A19 remain as stated.

2.3 Axioms for other properties of reshape

In V4 array theory, *reshape* uses the virtual item of the right argument if it is empty and hence A20 simplifies to

A20 *first (shape A reshape B) = first B*

One difference in V4 array theory is that *reshape* uses a coercion operation, *gage* to change any left argument it is given into a shape. We introduce D4.5 to express this.

D4.5 *gage A* **for** *shape (A reshape 0)*

The details of the coercion are not important to the development other than the fact that *gage* returns a suit of nonnegative integers.

The remainder of this section is valid in V4 array theory.

2.4 Properties of atoms and singles

The changes to this section are minor. T14 becomes

T14 *rest tally A = void tally A*

and

T20 *rest single A = void A*

With *single* defined, we can define

D28 *suit A* **for** *single first A* **if** *tally A = 1*
list A **otherwise**

and state

A5 *suit shape A = shape A*

2.5 Some non-equality theorems about lists

This section is unchanged.

2.6 Empty arrays

This section has to be reworked completely since this is where V4 differs from V6 substantially. A24 becomes

A24 *first void A = A*

Theorem T24 has to be replaced by a similar equation, which we state as an Axiom since it is not clear it can be proved here (left as an exercise).

A24.5 **if** *&prod shape A = 0* **then**
shape A reshape B = shape A reshape void first A

T25 holds but the proof has to be modified slightly to use *list A = void first A* in the empty case.

3. The replacement transformer *EACH*

The definition of *EACH* differs in V4 on empty arrays. It becomes

D8 *EACH f A* **for** *shape A reshape (f first A hitch EACH f rest A)* **if** *tally A ≠ 0*
shape A reshape void f first A **otherwise**

The following theorems about *EACH* are different in this section.

T29 *EACH f void A = void f A*

T35 *first EACH f A = f first A*

T37 *EACH f (shape A reshape B) = shape A reshape EACH f B*

A similar induction theorem holds.

T38 Let F be a formula where *A* is not free. **If** for every array A,
if *list A = void first A* **then** F(*list A*), **and**
if *tally A > 0* **and** F(*rest A*) **then** F(*list A*),
then for every array *A*, F(*list A*).

3.2 The replacement transforms *EACHLEFT* and *EACHRIGHT*

The definitions D9 and D10 must be modified in the analogous way as D8. The theorems hold as stated.

4. Construction of lists

The definition of *&link* and *link* have to be modified to support voids. They become

D12 *A &link B* **for** *list A* **if** *tally A = 0* **and** *tally B = 0*
list B **if** *tally A = 0* **and** *tally B > 0*
first A hitch (rest A &link B) **otherwise**

D13 *link A* **for** *void first first A* **if** *tally A = 0*
first A &link link rest A **otherwise**

The theorems are largely the same with changes to only T45 and T53. Many of the proofs will have to be adapted to allow for voids.

T45 *A &link void first B = list A*

T53 **if** *tally A > 0* **then** *link EACH link A = link link A*

4.1 Properties of pairing

The theorems and definitions of this section are unchanged in V4 array theory.

5. Construction of multidimensional arrays by Cartesian products

The definitions of *&cart* carries through to V4 array theory, however *cart* has to be modified to get the correct virtual item.

D18 *cart A* **for**
single EACH first A **if** *tally A = 0*
shape A EACHRIGHT reshape EACH hitch (first A &cart cart rest A) **otherwise**

The change in the result of *shape* changes T62 and T63 to

T62 *shape (A &cart B) = suit (shape A link shape B)*

T63 *shape cart A = gage link EACH shape A*

The proof of T63 may need T83 and T84, but they can be moved forward easily.

The generalization of T66 is

T65 *cart void A = single void first A*

T68 simplifies to

T68 *first cart A = EACH first A*

6. Boolean operations and membership

The definitions, axioms and theorems concerning equality and the Boolean connectives remains the same in V4 array theory. The definition of *&sublist* is modified to allow for voids.

D24 Let *A* be an array with Boolean values as items and let *B* an array with *tally A = tally B*. We define

A &sublist B **for** *list B* **if** *tally B = 0*
first B hitch (rest A &sublist rest B) **if** *first A = l*
rest A &sublist rest B **if** *first A = o*

All of the theorems are unchanged although some of the proofs may need minor changes.

7. Addressing and selection operations

The definitions and theorems in this section are changed considerably in V4 array theory due to the change in *shape* and a more general definition of *tell*. Also the selection operation *pick* reflects the choice made in Chapter 4, where picking from an empty returns the virtual item.

Theorem T81 is invalid since *shape* returns an integer on a list.

D27 is modified to be recursive and uses *gage* to coerce the argument. This has little impact on the development since the definition is unchanged for the cases used.

D27 Let A be a nonnegative integer or a list of nonnegative integer. Then

tell A **for** *&tell gage A* **if** *gage A* is a nonnegative integer
cart EACH tell gage A **otherwise**

The definition for *grid* is simplified by the choice of result for *shape*. In fact, the desire to have this definition was a motivation to choose the result of *shape* to be a suit in More's original work.

D30 *grid A* **for** *tell shape A*

Similarly, D32 is modified.

D32 *A find B* **for** *first (A findall B)* **if** *A in B*
shape B **otherwise**

The definition of *&picklist* changes to

D33 *I &picklist A* **for** *first A* **if** *I <= 0*
&pred I &picklist rest A **if** *0 < I < tally A*
first A **otherwise**

The first case is changed to select the virtual item. The last case is changed to support the behavior of picking out of range in V4 array theory, where the first item is always taken. Trenchard More has explored other choices for this case, including the choice where the given address is taken modulo the shape.

Theorem T95 is simplified by the change in *&picklist*. It becomes

T95 *I &picklist EACH f A = f (I &picklist A)*

The existence of virtual items and the new behavior of *&picklist* changes T96 as follows.

T96 *list A = list B* **iff**
tally A = tally B, and
I &picklist A = I &picklist B, for every array *I*

The proof for T96 should go through, but it has not been checked in detail.

D34 to D36 and T104 change to

D34 *I &atoi A* **for** *I find list tell A*

D35 *I pick A* **for** *gage I &atoi shape A &picklist A*

D36 *I &itoa A* **for** *I &picklist tell A*

Notice that *pick* gages its left argument, so that it is always a suit of nonnegative integers.

T104 **if** *I in grid A* **then** *I pick grid A = I*

The proofs of the theorems in this section should all go through with these systematic changes.

As discussed in Chapter 4, the theorem that corresponds to the axiom of extensionality for V4 array theory becomes

T105 *A = B* **iff**
shape A = shape B, **and**
I pick A = I pick B for every array *I*

The proof of T105 needs to be examined carefully and modified to meet the changes.

The remainder of the section is unchanged.

8. Conclusion

The sketch for the development of V4 array theory we have presented parallels that in Chapter 6 for the V6 theory. Not all the details have been worked out. There may be some gaps that need to be closed to get all the proofs to go through. However, we hope the reader is convinced that a completely formal development of the V4 theory is possible along these lines.

As in Chapter 6, we have not fully defined all the properties of the operations. In Chapter 10, we discuss the development of V4 definition sequences that start from a different premise. However, it is intended that the operations used there as primitives correspond to the ones of the same name defined in this chapter.

9. Using the V4shell to validate the V4 development

We have done a preliminary test of the development of this chapter as documented in the folder *V4inV4*. The files are named in the same way as in the folder *V6inV4* described in Chapter 7. The testing routines have been modified to generate empty arrays with arbitrary virtual items.

Chapter 10 Definition sequences for V4 Array Theory

1. Introduction

This chapter is a snapshot of on-going research on the development of definition sequences for V4 array theory being carried out by Peter Falster. The aims and scope of the research are:

1. To define a set of basic structural functions of array theory, i.e. those functions dealing with the reference frame, the address structure and only considering the domains of Boolean values and natural numbers and their associated operations from a purely geometrical viewpoint.

2. To study the behavior of these functions in the light of applications of tensor and matrix algebra.

3. To develop the set of functions as a definition sequence from a small number of primitive operations and transformers, and

4. To extend the definition sequence to support pervasive arithmetic and Boolean operations.

In other words by first considering the geometrical structure of arrays disregarding the substance, we develop the arithmetic functions considering numbers as "marks in the sand". This approach goes back to the Egyptian way of counting and trading, i.e. looking at numbers from a purely geometrical viewpoint.

At first we concern ourselves with structural functions that are either limited to simple arrays (arrays which items are atoms) or that are defined over the top two levels of an array. Then we generalize these ideas to pervading and pervasive operations, i.e. incorporating all the properties of the measurement scale functions.

The syntactic form of the definition sequence in the first part is based on a purely functional approach with all bound variables eliminated as originally developed by Schönfinkel [Schö24]. In the second part some use is made of the operation and transformer forms that are based on a lambda calculus approach with simple substitution rules for the bound variables.

We present the research on the V4 definition sequence in the following steps:

1. An overview of the functional syntax used in the definition sequence.
2. A discussion of the choice of different structural functions as primitives with a selection made.
3. A sequence of definitions in the variable-free functional form with related definitions gathered into groups,

4. An extension of the definition sequence of step 3 with recursive transformers and primitive scalar unary and dyadic operations, and making use of the parameterized operation and transformer forms.
5. A revision of some of the structural functions based on the arithmetic defined in step 4.
6. A discussion of the choice of primitive functions used in steps 2 and 4 in order to show how the number of primitives can be reduced and to relate it to the choice of primitives in the definition sequences in chapters 6 and 8.

Establishing a definition sequence is an engineering design process in the choice of primitive functions and defined functions. It is an iterative process where the choice of different set of primitives gives different definition sequences.

Trenchard More argues that his choice can be related to Zermelo-Fraenkel-Skolem's axiomatic approach to set theory [More79]. Mike Jenkins has chosen as his primitives the ones that form a basic equation that combines APL and Lisp concepts (Chapters 6 and 9).

One "optimization" criteria could be to select a minimum number of primitives. Another one could be to choose the set of primitives that gives the most effective definition sequence with respect to computer resources (storage and time). But, it should be stressed there is no best definition sequence.

Accordingly, we have been experimenting with different definition sequences; the one presented here is one among several we have studied over the past three years.

The primitive and defined functions discussed in the chapter are documented in the appendices at the end of the chapter. The folder *Version81* contains the definitions presented here.

2. Basic assumptions about syntax

This section summarizes the functional subset of Nial and is based on chapter 3. The three kinds of objects in array theory can be viewed as functions in mathematics with the following interpretations: a 0^{th} order function is an array *A*, a 1^{st} order function is an operation *f* and a 2^{nd} order function is a transformer *T*.

We assume the following syntax capabilities:

The form

<name> IS <ATexpr>

associates the meaning of the <ATexpr> with <name> and fixes the role of the name to be that of the <ATexpr>. Thus, if <ATexpr> is an operation expression then <name> will denote an operation.

Juxtaposition of two array expressions

A B

denotes a a one-dimensional array with two items. The notation is called a **strand** and it is generalized to

A B ... C

where A, B, ...,C are array expressions and denotes a one-dimensional array. Strand formation is given higher precedence when interpreting an array-theoretic expression.

The corresponding list notation

[A, B, ..., C]

where A, B, ..., C are array expressions, also denotes a one-dimensional array (a list) with items given by the corresponding array expressions. The special cases [A] and [] are also defined as the solitary, i.e. a list of length one, and the empty, i.e. a list of length zero, respectively.

The notation (juxtaposition)

f A

where f is an operation expression and A is an array expression, denotes the array formed by applying the operation denoted by f to the array denoted by A.

The notation

[f, g, .., h]

where f, g, ..., h are operation expressions, denotes the operation that when applied to an array expression A results in the same array as formed by

[f, g, ..., h] A = [f A, g A, .., h A]

The above equal sign means “equal by definition”.

The notation

T f

where T is a transformer expression and f is an operation expression, denotes the operation formed by applying the transformer denoted by T to the operation denoted by A.

The notation

[T, U, ..., V]

where T, U, ..., V are transformer expressions, denotes the transformer that when applied to an operation expression f results in the same operation as formed by

[T, U, ..., V] f = [T f , U f, ..., V f]

The notation

A f

where A is an array expression and f is an operation expression denotes the operation that when applied to the array expression B has the meaning of

(A f) B = f [A, B]

The notation

f g

denotes the operation formed by the composition of the operations denoted by f and g. i.e.

(f g) A = f (g A)

The notation

T U

denotes the transformer formed by the composition of the transformers denoted by T and U. i.e.

(T U) f = T (U f)

The notations

A T, f T, and T A

where A is an array expression, f is an operation expression, and T is a transformer expression denote transformers that are given meanings using the meanings of the above constructs by the equations

(A T) g = A (T g)
(f T) g = f (T g)
(T A) g = T (A g)

where g is an operation expression. This has been called the associative syntax for composition.

If three or more expressions appear in a sequence the resulting <ATexpr> is given meaning by grouping the expressions pairwise from the left after gathering the strands. Thus,

T f A

means

(T f) A

We name functions that are internal to the definition sequence with an identifier beginning with '&' to indicate they are not intended to be part of the set of operations in array theory. The internal functions may either be limited in scope or may just be means to define other more general functions.

The functions of the V4 Shell are denoted by underscored operation _f, and underscored transformer _T and are used in the following to define and implement the primitive functions. Thus, the primitives are *modeled* on the implemented V4 Shell functions.

The syntax for the operation and transformer forms is

<name> IS OP A B...(....)

<name> IS TR f g...(....)

The lambda calculus forms are only used initially as a means to define the primitives, but later are used explicitly in the extended definition sequence.

The above syntax has the following implications:

- the meaning of a pair of two array expressions (A B) = [A, B] has been defined
- it facilitates a functional approach.

The definition of the identity or neutral function *pass*, as well as the definitions of *Null*, *solo*, *twin*, etc. are on the boundary between what can be defined by the list syntax and the operation and transformer forms. We propose here to take *solo* as primitive and define *pass*, *Null* and *twin* in the functional style.

In order to develop the initial definition sequence in a pure variable-free style, we need the following three transformers as primitives:

NIX IS TR f (_pass)

ONCE IS TR f (f)

COM IS TR f g (f g)

The definition sequence is given in a functional style using the above three transformers in the sense that all the definitions in the initial sequence of structural functions are constructed variable-free, i.e. without using the lambda calculus forms.

3. Choice of primitive functions

Trenchard More has presented the arguments behind his choice of primitive functions [More79]. The choice of primitives is based on Zermelo-Fraenkel-Skolem's nine axioms for set-theory plus one additional primitive to measure the geometry of an array. The relationship of the axioms to the primitives is stated in table 1.

Axiom no.	**Set theory**	**Array Theory**
1	extensionality	equal
2	unordered pair	pair
3	subset	sublist
4	powers	tell, cart
5	union	link
6	choice	first
7	infinity	-
8	replacement	EACH, reshape
9	foundation	-
	-	shape

Table 1 Zermelo-Fraenkel-Skolem set theory and More's primitive functions

In chapter 6, Jenkins based his choice of primitives on the fundamental axiom that combines the array properties of APL with the list properties of Lisp:

shape A reshape (first A hitch rest A) = A

This gives six primitives, namely *shape*, *reshape*, *first*, *hitch, rest* and *equal*. In addition his approach assumes the arithmetic operations of Peano arithmetic.

In the definition sequences we have been studying, the choice of primitives is related to the concept of measurements and measurement scales.

Measurement is the assignment of numerals to observable properties of objects, events or situations according to rule.

We discuss two aspects of this definition:

- the measurement itself, which can be considered as the *substance* of an atomic array,
- the objects, events or situations which constitute the *axes* or *reference* system of the array.

For example, we could represent the height of a set of persons by a list considering the persons as objects or by indices on the axis of the list and the measurement of heights by the items of the list.

Both aspects of measurements are related to the classification of measurements according to four scale forms described in Table 2. (See also [Fran84, Fran85]).

	Operation or relation	**Binary operation**	**Transformations**	**Group structure**
Nominal	=		Permutations	Symmetric
Ordinal	<	min	Reversal and cyclic shifts	Monotonic
Interval	-	opp, +	x'=ax+b; a>0	Affine
Ratio	/	recip, *	x'=ax; a>0	Similarity

Table 2 Classification of measurement scales

The properties of the scales given in the table are cumulative from the Nominal scale down to the Ratio scale. For example, *equal* is applicable to measurements on all the scales.

The first column gives a classification of each measurement scale by stating the characteristic relation or operation of the scale form. The second column states in addition to this the binary associative and commutative operations *min*, *sum* and *product*, and the unary operations *opposite* and *reciprocal*, which are the natural choice as primitives for the ordinal, interval and ratio scale. The focus of the first two columns is on the substance of atomic data and on the pervasive properties of the relations and operations. This interpretation means that the characteristic operation for the nominal scale, =, is not the array-theoretic *equal* but the pervading operations *match* or *mate* (see appendix A).

The only operation given for the ordinal scale is *min*. The table does not describe all the operations that are defined for data on the ordinal scale. The theory includes all the Boolean operations that can be derived from the operation *equal* (see section 4) and all the operations from multi-valued logic.

The third column describes the transformations on measurements that leaves them invariant. The last column gives the group structure that corresponds to the invariant transformations.

The choice of structural primitives is based on the fundamental idea of considering an array as a geometrical nested object of orthogonal axes:

1. introduce transformations under which each axis of the reference system is invariant, i.e. reversal (*reverse* or *rev*) and cyclic shift (*cycle*),
2. consider setting the axes after each other and orthogonal to each other by linking (*link*) and Cartesian product (*cart*), respectively,
3. mark the geometrical space by ordering the points in the space in principal order by the natural numbers (*mark*),
4. introduce nesting of a reference system by making a list with one item increasing depth (*solo*), and removing nesting and decreasing depth by selecting the item (*first*),
5. introduce a universal mapping function as a homomorphism with replacement of items over one level of nesting (*EACH* or *EA*),
6. introduce transformations from geometrical form to natural numbers (*size*) and from Boolean values to natural numbers (*toint*),
7. introduce a universal identity operation of the items of an array (*equal*) by mapping to the atoms *Truth* if all the items are identical and *Falsehood*, otherwise.

The following assumptions are made regarding the initial definition sequence:

- the allowed atomic domains are only Boolean values, the natural numbers and the characters,
- operations are monadic (although they may be applied to a pair) and are totally defined,
- no recursion and branching are allowed,
- definitions are variable-free.

The ordering and accessing of points in a nested space are based on the following principles:

- main order (or principal order) and the points/items accessed by *indexing*,
- Cartesian product and the points/items accessed by *addressing,*

Table 3 gives the set of primitive functions we propose as a foundation for the definition sequence along with a short explanation of each primitive.

equal	Universal identity of items of an array by mapping to Truth or Falsehood

EACH (EA)	Universal transformer mapping (homomorphism), replacement of items over one level
reverse (rev)	places the items of an array in reverse order
cycle	rotates the elements of the second item according to the number of elements of the first item
first	decrease depth by picking the first item of an array
solo	increase depth by making a list of one item
link	the list of the items of the items and "aussonderung" of the empties
cart	Cartesian product of the items of an array
size	measure the extent along each axis
mark	generates the list of primary indices from a natural number in principal order
toint	map truth and falsehood to 0 and 1, respectively, otherwise passing

Table 3 Primitive functions

The "Aussonderung" principle [More73] is implicitly part of the operation *link* for the following reason. Since *link A* is the same as *link main A*, where *main A* removes the empty items of *A*, we can control whether items in a list are selected by enclosing them in a solo or an empty prior to linking the modified list. In More's set of primitives Aussondering is achieved with *sublist* using a Boolean array to indicate which items to select. In order to achieve the Boolean Aussunderung of *sublist*, we need to have a mapping from the Booleans to the natural numbers 0 and 1. This is provided by the primitive *toint*.

Table 4 gives the corresponding set of implemented primitive functions based on the internal implemented functions in the V4 Shell.

```
eq IS _equal
EA IS _EACH
rev IS _reverse
cycle IS _reshape [ _shape _second , _FOLD ( _link [ _rest , _solitary _first ])
                [_tally _first, _second] ]
link IS _link
cart IS _cart
first IS _first
solo IS _solitary
size IS _shape
mark IS _reshape[_pass, _tell _product]
```

```
        _abs _sublist[_EACH _isinteger, _pass] _content
toint IS _FORK[ _isboolean, _abs, _pass]
```

Table 4 Implementation of primitive functions

The basic assumptions and characteristic properties on which the primitives are based must be defined in some way. In Chapters 6 and 9 first order predicate calculus is used as a meta language to state the properties of the primitives as axioms. Two fundamental properties of a function's behavior must be stated, namely how the resulting array is determined with regards to form and with regards to substance. One way to do this is by specifying the form and substance at the top level by using *size* and *pick*, respectively as discussed in Chapter 4.

We return to the problem in section 6, where we state some of the equations (axioms and theorems) for the primitive functions.

The properties of the above primitives are determined in either of two ways, based on equations or based on design choices. A more in–depth discussion of the primitive functions and their properties based on equations can be found in [More73, More79, More81 and More93]

It should be stressed that all that is known from this point forward in the initial definition sequence is entirely determined by the eleven primitive functions and the syntax rules. The V4 Shell with all its functions is only used as a means for adding definitions and from now on it is only the parser we use and the few C-routines that implements the primitives.

4. Defined structural functions

Starting with the selected primitives we define a set of structural functions, i.e. a set of functions which gives meaning to useful concepts as outer product, inner product, transpose, etc. We can study the functionality of these functions, in particular, how they behave for the special cases of *single*, *solo* and *void*. All the behavior of the defined functions can be traced back to the basic assumptions behind the primitives and their definitions.

The definition sequence constructed here follows the approach taken by More in [More81]. The definitions are also are found in the folder *v4seqs* provided with the V4 Shell. Most of the functions follows the definitions originally defined in the Q'Nial Dictionary [Jenk85], which contains detailed explanations. Changes to the original definitions are not documented here but can be investigated by the reader through experimentation with the V4 Shell.

In the remainder of this section we present groups of definitions organized into tables. The groups illustrate just one possible way of organizing the definition sequence.

All Boolean functions (connectives) dealing with measurements on the *nominal scale* can be defined simply from *equal* and one array, e.g. the atomic character ‘a’, denoted in Nial by *`a*.

```
pass IS first solo
twin IS [pass, pass]
TWICE IS COM[ONCE,ONCE]
Truth IS eq `a
verity IS eq eq
Falsehood IS Truth eq `a
Null IS size Truth
&not IS (Falsehood eq) (Truth eq)
&uneq IS &not eq
~= IS &uneq
falsity IS &not verity
&and IS eq (l link)
&or IS &not &and EA &not
boolean IS &or [l eq, o eq]
```

We have used *equal* (*eq*) on a character atom to define the atomic truth value *Truth*, which is then utilized in the definition of the empty list *Null*. Any atom could be used in place of *`a*. An alternative would be to assume *Null* and define *Truth* from it using *eq*.

The basic structural functions

```
last IS first rev
void IS link cart [solo, Null first]
vacate IS link void
lshift IS Truth cycle
rshift IS rev lshift rev
second IS first lshift
pair IS [first, second]
copair IS [second, first]
CONVERSE IS COM[ONCE, rev  NIX]
CON IS CONVERSE
single IS EA first cart void
```

The constant arrays

```
Zero IS size Null
One IS size solo Zero
Two IS size 0 0
Bull IS vacate Falsehood
Nil IS vacate `
```

The list constructing functions

```
list IS link solo
append IS link[first, solo second]
hitch IS link[solo first, second]
```

Three fundamental predicate functions

```
        NEUTRAL IS eq [NIX, ONCE]
atomic IS NEUTRAL single
simple IS NEUTRAL (EA single)
```

The EACHRIGHT and EACHLEFT transformers

```
EACHRIGHT IS COM[EA, cart[single first, second]NIX]
EACHLEFT IS COM[EA, cart [first, single second]NIX]
```

The functions related to counting, i.e. number of elements and number of axes

```
tally IS size list
valence IS tally size
empty IS 0 eq tally
isone IS 1 eq tally
```

The array structure of primary addresses and indices, integer coercing and tell of an array

```
grid IS cart EA mark size
grill IS mark size
gage IS size mark
&tell IS cart EA mark gage
```

Shape, scale and axes

```
shape IS list size
scale IS EA mark size
axes IS mark valence
```

Rotating and indexing

```
rotate IS cycle[mark first, second]
picklist IS first rotate
```

Reshaping, interchanging the top two levels and fitting

```
reshape IS EL picklist [mark first, second]
re IS reshape
```

```
pin IS EL ER picklist [grill first, pass]
conform IS ER EL picklist [grill first, pass]
fit IS second conform
```

Selecting from a list

```
sublist IS EA second link EA (pin copair)
        pin [EA solo second, EA (mark &toint (Truth eq)) first]
SIFT IS sublist [EA,NIX]
rest IS sublist[link[o first,EA verity],pass]
front IS sublist[rest link[EA verity, o first],pass]
rear IS sublist[rest link[EA falsity, l first],pass]
main IS sublist [EA((o eq) (0 eq tally)),pass]
oneof IS eq [1 first, tally sublist twin]
```

Some reshaping functions

```
post IS reshape [[tally,1 first],pass]
dwarf IS re [SIFT (1 &uneq) size, pass]
suit IS dwarf list
```

The Boolean membership functions

```
in IS &or (ER eq)
membership IS EL in
allin IS &and (EL in)
lap IS sublist [EA &and EL ER in[first, pass],first]
except IS sublist[EL ((o eq) in), first]
like IS &and [allin, CON allin]
```

The functions related to selecting from and searching in an array

```
&pick IS first sublist [ER eq
        [gage first,grid second],second]
findall IS sublist [ER eq,grid second]
find IS first append [findall,size second]
seek IS [in,find]
indexof IS EL find
```

The operation *&pick* is total but if the address in the first item is out of range for the second item it selects the first item. Basically, the following principles can be applied for picking out of range and coercing the argument:

- apply the address modulo the shape to select the item (cyclic)
- select the first item

- return a fault (faults are not included in the assumptions for the initial definition sequence)
- apply *gage* to the address argument to convert to a suit of numbers

Applying modulo and trimming the address follows later in the definition sequence.

The functions related to set theory

```
members IS EL in [grid, EL find twin]
cull IS sublist [members, pass]
diverse IS eq [ cull , list]
join IS cull link
```

The *ordinal scale* functions, geometrical determined

```
&lower IS allin EA mark
&below IS &and [&lower, diverse]
&max IS tally cull link EA mark
&min IS tally lap EA mark
&incr IS tally (0 hitch) link mark
&decr IS tally front link EA mark
&glb IS rear EA &incr lap EA mark
&lub IS rear EA &incr cull link EA mark
```

The distinction between *&min* and *&max* and *&glb* (greatest lower bound) and *&lub* (least upper bound) are that the first pair of operations deal with measurements on a ratio scale while the last pair deal with measurement on the interval scale. Another distinction is that *&glb Null = &lub Null = Null*, i.e. the latter pair of operations pass on the empty array. This property is utilized in the colligation function *diag*.

The family of Peircean functions

```
transpose IS EL &pick [(EA rev) (cart EA mark) rev size, pass]
trp IS transpose
&mix IS re[link [size, size first] , link]
rows IS EL EL &pick [EL ER link EA (cart EA mark)
                    [front, rear] size, pass]
cols IS trp rows trp
&align IS &mix pin
diag IS EL &pick [pin re [valence, solo mark &glb size],pass]
```

The *OUTER* and *INNER* product transformers

```
          OUTER IS COM[EA, cart NIX]
INNER IS COM[OUTER, [rows first, cols second]NIX]
```

The *interval* and *ratio scale* functions, geometrical determined

```
&sum IS tally link EA mark
&minus IS tally except EA mark
&prod IS tally cart EA mark
&mark IS re[pass, mark &prod]
&modus IS find[&prod first
        ,&mark hitch [&incr first, second]]
&modular IS second &modus
&quotient IS first &modus
```

Taking, dropping and placing from/in an array

```
&take IS EL &pick[cart EA mark first, second]
&takeright IS rev &take
            [first, rev second]
third IS first (ool sublist)
&TIE IS COM[EA, pin NIX]
&drop IS EL &pick [cart &TIE except
       [second, &TIE take] [first
       , EA mark size second],second]
placeall   IS &TIE &pick [ EL find [ grid first , second ]
           , ER append [ reshape
           [ tally second , third ] , first ] ]
           [ first , EL &pick [ first second
           , grid first ] , second second ]
place IS placeall [ first , EA solo second ]
```

We define below the operation *trim* that trims the argument to the pervasive operations and the operation *&trimaddr* that trims addresses to the more general *pick* operation. Note that these operations are completely defined by means of the structural functions only.

```
&trimaddr IS suit rev (&mod pin) ER re[&glb EA tally, pass]
                [link[rev first, 0 CON re valence second], rev size second]

pick IS &pick [trimaddr, second]
```

Some further Boolean functions

```
&imply IS &and EA &or pin[EA &not front,rest]
&iff IS &or [&and, &and EA &not]
&xor IS &and [&or, &or EA &not]
&lte IS eq [EA &min pin [front, rest], front]
&gte IS &lte rev
```

5. Extending the definition sequence to pervasive operations

The next step is to extend the operations associated with the measurement scales to be pervasive. This requires some means to achieve descent into an array to an arbitrary depth. We now extend the allowed syntax to permit operation forms, however, *IS* is still used only for non-recursive definitions. We add conditional branching using *FORK* and the recursive transformers of Chapter 5 as primitives and we extend the atomic domains to include faults.

```
FORK IS _FORK
RECUR IS _RECUR
ACROSS IS _ACROSS
DOWN IS _DOWN
```

The form of the pervasive *unary* operations is defined in two steps.

1. The definition on atomic domains is given as *&f*.

&f IS FORK [is_domain, _f, pass]

where *is_domain* is a predicate that test for the proper atomic domain. For example, *numeric* tests if the argument is Boolean, integer or real. If the argument is outside the domain the defined operation passes.

2. Then the definition is extended using *LEAF* to descend to the atomic level.

f IS LEAF &f

where *LEAF* is defined by

LEAF IS TR f (DOWN [atomic, f, pass, pass])

The primitive unary operations on atoms and there extensions are

```
&opp IS  _FORK[0 0. _CONVERSE _in _type, _opposite, _pass]
&abs IS  _FORK[0 0. _CONVERSE _in _type, _abs, _pass]
&floor IS _FORK[ 0. _CONVERSE _in _type, _floor, _pass]
&recip IS _FORK[0 0. _CONVERSE _in _type, _recip, _pass]
opp IS LEAF &opp
abs IS LEAF &abs
floor IS LEAF &floor
recip IS LEAF &recip
type IS LEAF _type
```

Before dealing with the measurement scale binary functions we must define two additional structural functions *&trim* and *pack*, using an operation form to define the former.

```
&trim IS OP A {
  V:= &lub EA valence A;
  E:= &lub link EA size A;
  Oldshapes:= EA size A;
  Newshapes:= EA V &takeright  (V re E ER link oldshapes);
  Cutshape:= EA &glb pin Newshapes;
  Trimmed:= EA valence A EL &takeright Cutshape &TIE &take A;
  Cutshape ER re Trimmed }
pack is pin &trim
```

The form of the *binary* operations is developed in three steps.

1. Each binary operation is adjusted according to how we want to treat values outside the proper domain (scale form). The problem of defining special result tables if one or both arguments are outside domain is left for further research. To simplify matters we therefore map to a fault value if outside its proper domain. Notice that the primitive function *_f* already performs this mapping. But for clarity we have explicitly mentioned *&f* such that it later can be redefined if necessary. Thus, the form for the first step of defining each binary pervasive operation is

&f IS FORK[and EACH is_domain, _f, ??&f]

2. Define the boundary case when the given argument is empty (the identity for the operation) and use *ACROSS* as described in Chapter 5 to apply the binary operation to reduce a simple array. The form is:

&&f IS ACROSS[Identity first, pass, &f]

3. Pervade down the array using *DIG* until it meets a simple array.

f IS DIG &&f

where *DIG* is defined by

DIG IS TR f (DOWN[simple, f, pack, pass])

The primitive binary operations on atoms are:

```
 &min IS _min
&sum IS _sum
&prod IS _product
&quot IS _quotient
```

We illustrate the definition process for pervasive operations based on binary measurement functions by defining *sum*.

```
 &&sum IS ACROSS [0 first, pass, &sum]
sum IS DIG &sum
```

A more complete set of the defined functions is documented in appendix A.

6. Equations (axioms and theorems)

The research on definitions sequences discussed in this chapter has not focussed on a formal axiomatic development. As a result, our interest has been to achieve definitions that satisfied many equations or universal laws. In this section we state the equations that hold for the primitive structural functions. They capture the key properties of the primitives, but may not specify them completely. There is no distinction made here between axioms and theorems. Which of the equations should be taken as axioms and which should be theorems that are proved from the axioms has not yet been determined.

equal

```
eq = eq[re[size, single first],pass]
eq A = eq list A
tally eq A = 1 first A
size eq A = Null first A
eq single A = Truth
eq solo A = Truth
eq void A = Truth
```

EACH

```
size EA f A = size A
f first A = first EA f A
f second A =  second EA f A
f last A = last EA f
f (I pick) A = (I pick) EA f A
EA f rest A = rest EA f A
single f A = EA f single A
solo f A = EA f solo A
void f A = EA f void A
A is an atom => A = single A => EA f A = single f A
size A reshape list A = A
EA(f g) A = (EA f) (EA g) A
```

reverse

```
reverse reverse A = A
```

```
size reverse A = size A
tally reverse A = tally A
```

cycle

```
cycle = rotate [tally first, second]
Null cycle A = pass
A cycle Null = Null
cycle A = cycle pair A
cycle single A = pass A
```

link

```
link list A = link A
tally link A = sum EA tally A
link A = list link A
link A = link EA list A
link A = link list A
link void A = void first A
void A link B = list B
A link void B = list A
cart link A = EA link cart EA cart A
link A = EA list link A
link A = link main A
link void A = void first A
EA f link A = link EA EA f A
f is associative if it satisfies the associative law: f EA f = f link
For sum and prod: f EA f EA tally A = f link EA tally A
```

cart

```
size cart A = link EA size A
tally A = prod size A
tally cart A = prod EA tally A
prod link EA tally A = prod EA prod EA tally A
first cart A = EA first A
cart void A = single void first A
cart single A = EA single A
cart solo A = EA solo A
EA first A = first cart A
cart EA single A = single A
EA EA f cart A = cart EA EA f A
cart link A = EA link cart EA cart
cart A = EL TIE EL &pick [&tell EA size, pass] A
```

first

```
A = A first B
first (A B) = AB
first solo A =A
```

first void A = A first single A = A first list A = second A

solo
first solo A = A tally solo A = 1 first A suit size solo A = 1 first A solo = OP A ([A]) solo = [pass]

size
size A = suit size A and EA (isordinal) size A = Truth first A isordinal tally A = Truth first A tally A = prod size A tally A = size list A size single A = Null first A size void A = 0 first A size solo A = 1 first A

mark
mark Null = 0 mark A = re [pass, tell tally tell] A mark A = mark gage A size mark A = size tell A

toint
toint Falsehood = Zero toint Truth = One not boolean: toint A = pass A

NIX, ONCE and **COM**
NIX f A = pass A ONCE f A = f A COM [f,g] = f g

In complete analogy we can define universal laws for the defined functions. Many of those laws are found in the Q'Nial Reference manual and therefore not included here.

7. Experimenting with different definition sequences

Our research has involved the investigation of many definition sequences based on alternative choices of the primitive functions. In this section we discuss some of our conclusions about alternative choices. We present the conclusions in terms of Tables 5a

and 5b below. Table 5a is a summary of experiments related to the choice of structural primitives. Table 5b is a summary of experiments where we extend the structural part with branching and recursive transformers. The tables are to be considered together in the sense that the correspondingly titled columns correspond to the same experiment.

Structural & Recursive Transformers		Structural & Recursive Transformers (revised)	Structural & Recursive Definitions
equal	equal	equal	equal
EACH	EACH	-	-
reverse	reverse-	-	-
cycle	-	-	-
link	link	link	hitch
cart	cart	cart	-
first	-	-	first
-	second	second	rest
solo (single, void)	-	-	-
size	size	size	shape
mark	mark	-	-
-	pin	pin	reshape
-	SIFT	-	-
to_integer	-	-	-
NIX		-	-
ONCE		-	-
COM		-	-

Table 5a Different choices of primitives

Structural & Recursive Transformers	Structural & Recursive Transformers (revised)	Structural & Recursive Definitions
FORK	FORK	FORK
RECUR	RECUR	-
ACROSS	ACROSS	-
DOWN	DOWN	-
abs, opp, floor, recip, type	abs, opp, floor, recip, type	abs, opp, floor, recip, type, pred, succ
min, sum, prod, quot	min, sum, prod, quot	min, sum, prod, quot

Table 5b Different choices of primitives

The first two columns of Table 5a give choices for the initial definition sequence for the structural functions. The first column of Table 5b is the corresponding extension. In the first column we indicate that we could equally well have taken either *single* or *void* as

primitives instead of *solo* as the primitive that increases depth. If *single* is chosen, it has only minor impact on the overall definition sequence, but it makes the definitions of *solo* and *void* more transparent than those of *void* and *single* in the definition sequence we have presented.

In the second column we have replaced the primitives *cycle*, *solo* and *first* by *second* and *pin*. *Pin* incorporates the idea of cycling, reshaping and interchanging the top two levels in one operation. Starting with *second* we can define both *first* and *pair*. The operation *pair* was chosen as primitive by More in one of his early definition sequences (cf. Table 1).

The functions *first* and *second* have the following properties:

The operation *first* corresponds to Schönfinkel's "Konstanz" function when it is used with currying. That is, whatever argument *B* we apply the curried operation *A first* to, it always maps it to *A*. This is stated by the equation

A first B = A

Basically, if *A* is a constant array, *A first* is the corresponding constant function. We can utilize this fact in a definition sequence, for example, by defining *verity* by

verity IS Truth first.

The operation *second* satisfies the equation

B =A second B

which states that whatever argument *B* we apply the curried operation *A second* to, we always get *B*, that is we *pass*. This allows *pass* to be defined by

pass IS Null second.

Both definitions originate in the fact that at the outset we have defined a pair and the interpretation of currying by the equation:

(A f) B = f (A B)

Our research indicates that we can choose either *first* or *second* as the primitive that reduces depth. However, we have discovered out that *pin* and *second* complement each other in an elegant way.

The second column of Table 5a also indicates that we can replace *toint* with the transformer *SIFT* which directly implements the Aussonderung principle. We have found that using *SIFT* together with *pin* can simplify the definition sequence considerably. To illustrate these ideas consider the following sequence.

```
pass IS 'a' second
twin IS [pass,pass]
verity IS eq eq
falsity IS (Falsehood eq) verity
list IS SIFT verity
```

```
vacate IS SIFT falsity
void IS vacate twin
first IS second vacate
fit IS second pin pin
sublist IS EA first SIFT second pin [second, first]
re IS fit [mark first, second]
```

The third column indicates that if we include the recursive transformers of Chapter 5 then fewer primitives are needed since several of the primitives we have selected in the second column, such as *EACH*, *mark* and *SIFT* can be defined.

The fourth column indicates the primitives used in Chapter 9, which requires that *IS* be used to make recursive definitions directly. We note that *hitch* combines *link* and *solo* and *reshape* replaces *pin*. With the availability of recursive definitions we can even remove *cart*, *EACH* and *mark*.

8. Conclusion

We have illustrated how from a set of 11 primitive functions you can define almost all the structural functions in V4 array theory and thereby also in the Nial language. The simple Boolean and arithmetic functions are defined from geometrical considerations. However, measurement scale functions dealing with substance are, in general, defined on the domains of integers and reals (and eventually the literal domains), and must start with those functions as primitives. Generalizing these measurement scale functions to have pervasive properties requires recursion and can be based on the recursive transformers (ACROSS and DOWN) or on direct recursive definitions.

The definitions given here have been developed using the V4 Shell. However, we have not yet done a validation of the definitions using the techniques described in Chapter 7. This task is left for the next stage of the research.

Appendix A Definition sequence for pervasive measurement scale functions

```
# EACH f A : EACH is the replacment transformer that results in an array
        of the same shape as A, with each item of the result formed
        by applying f to the corresponding item of A. If A is empty
        the archetype of the result is f applied to the archetype of A.
        EA is an abbreviation for EACH.

# EACH IS TR f ( &fit [pass, ACROSS [void f first, f, hitch] ] )

# reverse IS &fit [pass , ACROSS [ pass, pass, append copair ] ]

isinteger IS FORK [atomic, (0 eq) &type, o first ]

&incr IS (One &sum)

&decr IS (&opp One &sum)

&lte IS eq [&min, first]

# FOLD f [N,A] : If N is an ordinal then FOLD is the transformer that
            applies the operation argument f to A, N times. If
            N is a negative number then f is applied to the reverse
            of A, abs N times, and the result is reversed. If N is
            not an integer the result is A. If the argument to FOLD f
            is not a pair then it is treated as the pair.

&FOLD IS TR f (RECUR [0 eq first, second, pass,
                f second, [ &decr first, second] ])

FOLD IS TR f (FORK [ isinteger first,
              FORK [0 &lte first,
                 &FOLD f,
                 rev &FOLD f [&abs first, rev second] ],
             second ] pair)

# NIX IS 0 FOLD

# ONCE IS 1 FOLD

&vote IS FORK [first, solo rest, void rest]

# content A : the list of atoms held in A at any level.
```

```
content IS DOWN [ simple, list, pass, link ]

reach IS FORK[empty first, second, reach[rest first, &pick[first, second]]]

# LEAF f A : LEAF is a recursive (pervasive)transformer that descends an array A
        until it reaches an atom. It then applies f to the atom.
        If f maps an atom to an atom then the result has the same
        structure as A.

LEAF IS TR f ( DOWN [atomic, f, pass, pass ] )

# Pervasive functions

type IS LEAF &type

&&not IS FORK[o CON in type, &not, pass]

not IS LEAF &&not

opp IS LEAF &opp

floor IS LEAF &floor

ceiling IS opp floor opp

recip IS LEAF &recip

abs IS LEAF &abs;

# Pervading functions:

# Functions related to ordinal, interval and ratio scale on simple arrays

# Here follows a conversion to reals!

&&min IS ACROSS [1E350 first, pass, &min]

&max IS opp &&min opp

&&sum IS ACROSS [0 first, pass, &sum]

&minus IS &&sum[first, &opp second]

&&prod IS ACROSS [1 first, pass, &prod]

&mod IS &minus[first, &&prod[&quot, second]]
```

```
&trim IS OP A {
  V:= &lub EA valence A;
  E:= &lub link EA size A;
  Oldshapes:= EA size A;
  Newshapes:= EA V &takeright  (V re E ER link oldshapes);
  Cutshape:= EA &glb pin Newshapes;
  Trimmed:= EA valence A EL &takeright Cutshape &TIE &take A;
  Cutshape ER re Trimmed }

pack is pin &trim

TIE IS COM[(EA , pack NIX]

# DIG f A : DIG is a recursive transformer used to descend an array
        A until it finds a simple array. It then applies f to the
        simple array. During the descent it "packs" the arrays so
        the items have the same shape and corresponding items are
        brought together. Thus, DIG f applied to a pair of lists of
        the same shape will result in a list with f applied to each
        corresponding pair.

DIG IS TR f ( DOWN [simple, f, pack, pass] )

match IS DIG eq

toreal IS DIG FORK [ o 0 CONVERSE in &type , 1.0 &prod, pass ]

mate IS match toreal

&&and IS &and

and IS DIG &&and

or IS not and not

imply IS and or [not front, rest]

iff IS or [and, and not]

xor IS and [or, or not]

min IS DIG &&min

max IS opp min opp

lte IS and mate [min[front,rest],front]
```

```
gte IS lte rev

lt IS and [lte, not mate]

gt is lt rev

sum IS DIG &&sum

minus IS sum EA FOLD opp pin[0 1 CON re size, pass]

prod IS DIG &&prod

div IS prod EA FOLD recip pin[0 1 CON re size, pass]

quot IS DIG (&quot pair)

mod IS minus[first, prod[quot, second]]

depth IS DOWN [atomic, 0 first, pass, 1 sum max ]

RUN IS TR f (EA f EL sublist [EL CON lte EA mark tally, pass])

SWEEP IS TR f (EA f EL sublist
    [EL CON lt EA mark(1 0 EL &&sum) tally , pass])

BIN IS TR f (EA f pin [front, rest])

# tell A : The operation tell produces an array frame based on the
         integers that form the content of A. Each item of the result
         has the same structure as A and contains integers less than the
         corresponding items of abs A. The result consists of all possible
         such arrays organized by as many axes as there are integers
         in the content of A. If the content of A is not all integers
         the items that are not integers are replicated in each result
         item. If an integer in the content of A is negative, the
         corresponding entries are enumerated in reverse order.

&tell IS RECUR [Zero eq, Null first, &decr, append copair, &decr ] ;

# &tell =  mark

tell IS DOWN [ atomic , FORK [ isinteger,
                        FORK [0 &lte, mark, opp (1 sum) rev mark &abs],
                       pass],pass, cart]
```

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.

[JeJe98]

Jenkins M. A., Jenkins W. H., "The Nial Tools Manual", available by download from the NIAL Systems Limited Home Page at *http://www.nial.com*, under Free Software.

[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.

[MaWa85]
Manna Z., Waldinger R., *The Logical Basis for Computer Programming*, Addison-Wesley, Reading, Mass. 1985.

[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.

[Schö24]
Schönfinkel M., "On the building blocks of mathematical logic", *From Frege to Gödel – A source Book in Mathematical Logic, 1879-1931* (edited by Jean van Heijenoort), Harvard University Press, Cambridge, Massachusetts, 1977.