UNIVERSITY OF JOENSUU
COMPUTER SCIENCE
DISSERTATIONS 6

Stephen Eriksson-Bique

AN ALGEBRAIC THEORY OF MULTIDIMENSIONAL ARRAYS

ACADEMIC DISSERTATION

To be presented, with the permission of the Faculty of Science of the University of Joensuu, for public criticism in Auditorium Louhela of the Carelian Science Park, Länsikatu 15, Joensuu, on April 12th, 2002, at 12 noon.

UNIVERSITY OF JOENSUU
2002

Julkaisija Joensuun yliopisto
Publisher University of Joensuu

Toimittaja Jussi Parkkinen
Editor

Vaihdot Joensuun yliopiston kirjasto, vaihdot
PL 107, 80101 JOENSUU
puh. (013) 251 2677, fax (013) 251 2691
email: vaihdot@joensuu.fi

Exchanges Joensuu University Library, Exchanges
P.O. Box 107, FIN-80101 Joensuu, FINLAND
tel. +358 13 251 2677, fax +358 13 251 2691
email: vaihdot@joensuu.fi

Myynti Joensuun yliopiston kirjasto, julkaisujen myynti
PL 107, 80101 JOENSUU
puh. (013) 251 2652, fax (013) 251 2691
email: joepub@joensuu.fi

Sales Joensuu University Library, Sale of Publications
P.O. Box 107, FIN-80101 Joensuu, FINLAND
tel. +358 13 251 2652, fax +358 13 251 2691
email: joepub@joensuu.fi

ISSN 1238-6944
ISBN 952-458-108-6
UDK 519.682
Computing Reviews (1998) Classification: D.1.1, D.3.1, D.3.2, D.3.3, E.1
Yliopistopaino
Joensuu 2002

AN ALGEBRAIC THEORY OF MULTIDIMENSIONAL ARRAYS

Stephen Eriksson-Bique

Department of Computer Science
University of Joensuu
P.O. Box 111, FIN-80101 Joensuu, FINLAND
Stephen.Eriksson-Bique@Joensuu.FI

University of Joensuu, Computer Science, Dissertations 6
Joensuu, April 2002, 278 pages
ISSN 1238-6944, ISBN 952-458-108-6

Keywords: applicative programming, categorical data type, homomorphism, primitive

An algebra of programming for multidimensional arrays is presented. This new calculus enhances software development as in the theory of lists and provides a complete theory for the data type as in More's Array Theory. An architecture-independent approach is taken that allows program derivation and optimization. Multidimensional arrays have a single type. Notation and terminology are introduced to facilitate reasoning about arrays. New definitions help to simplify some of the definitions and proofs. A set of primitive operations is defined. This set represents the data type as in abstract data types and includes new functions that account for common program structures. Arrays are equipped with all of their known properties and features, and with appropriate tools to take advantage of all of their dimensions. Yet, the theory is concise. A sound methodology is prescribed to define primitive operations instead of freely defining functions ad hoc. Mainly, operations are defined in a structured way without using indices. A constructive theory is developed which includes useful identities, properties and laws. The formulas lack many of the indices typically required. Array homomorphisms are explicitly classified. Generic programming is possible using templates for different computations. Programming techniques are explained. Case studies are done which show that programs can be written at a low-level using fine grain parallelism.

To Sirkka-Liisa, Anna-Maria, Sylvester and Linda

ACKNOWLEDGEMENTS

The Ph.D. candidate thanks Professor Elena Trichina for invaluable advice, guidance and support as supervisor, Professor Roope Kaivola for serving as reviewer, Professor Martti Penttonen for supervising the study program and serving as reviewer, Professor Jean-Luc Gaudiot for serving as opponent and Professor Jussi Parkkinen for serving as custodian.

The candidate acknowledges the Department of Computer Science at the University of Joensuu for a scholarship and for teaching and research positions, East Finland Graduate School in Computer Science and Engineering for a research post, Computer Science Research Foundation (Tietotekniikan tutkimussäätiö) for a grant, Professor Elena Trichina for providing full financial support to visit the University of South Australia in Adelaide and McQuarry University in Sydney, and the U.S. government for an American Fulbright Award.

CONTENTS

1 INTRODUCTION **1**
- 1.1 Conventions . . . 2
- 1.2 Data Types . . . 4
- 1.3 Mathematical Background . . . 5
- 1.4 A Review of Literature . . . 11
- 1.5 Approach and Goals . . . 26
- 1.6 Contributions of this Thesis . . . 28
- 1.7 Organization of the Text . . . 29

2 A SURVEY OF ARRAYS **30**
- 2.1 How to Define Primitives . . . 30
- 2.2 Elementary Functions . . . 34
- 2.3 Representation . . . 39
- 2.4 Implementation . . . 40
- 2.5 Summary of Elementary Notation . . . 42

3 A FOUNDATION FOR ARRAYS **43**
- 3.1 Array Model . . . 43
- 3.2 Constructors . . . 56
- 3.3 Destructors . . . 61
- 3.4 Abstractors . . . 74
- 3.5 Macros . . . 90
- 3.6 Examples . . . 91
- 3.7 Review . . . 92

4 A CONSTRUCTIVE THEORY **95**
- 4.1 Fundamental Identities . . . 95
- 4.2 Array Forms . . . 101
- 4.3 Homomorphisms . . . 107
- 4.4 Laws and Properties . . . 133
- 4.5 Expressiveness . . . 138
- 4.6 Summary . . . 139

5 PROGRAMMING EXAMPLES 143
5.1 Destructors . . . 144
5.2 Pairing Operations . . . 148
5.3 Partitioning Operations . . . 151
5.4 Transformations . . . 153
5.5 Higher-Order Functions . . . 158
5.6 Array Products . . . 159
5.7 Summary . . . 164

6 NEAR HOMOMORPHISMS 166
6.1 Map and Reduce . . . 166
6.2 Parentheses Matching . . . 168
6.3 Pattern Matching . . . 174
6.4 Digit Serial Computations . . . 178
6.5 Summary . . . 194

7 FUTURE WORK 195

REFERENCES 205

APPENDIX 215

INDEX 268

CHAPTER 1

INTRODUCTION

Useful models along with simple complete theories for array expressions are important since array data structures are popular with numerous applications in biology, economics, engineering, image processing, physics, probability and statistics. Kenneth Iverson demonstrated successfully how mathematical expressions for arrays, not simply expressions for variables as in FORTRAN, could be built into a programming language. John Backus advocated an algebra of programming. Trenchard More advocated developing complete and independent theories for data types. Several attempts have been made during the past two decades to develop suitable theories for arrays. T. More, J. Reynolds, L. Mullin, R. Bird, R. Miller, C. Banger and D. Skillicorn propose different models, primitive operations, notation, and terminology.

The main task is to construct a set of primitive operations for programming with multidimensional arrays using mainly structured operations to achieve an algebra of programming that permits the derivation and optimization of programs as in the theory of lists (Bird, 1990, 1987, Skillicorn, 1990), which enhances software development (Axford and Joy, 1993, Backus, 1978, Bird, 1989a, Manes and Arbib, 1986, More, 1979, Mou and Hudak, 1988, Skillicorn, 1990). Using a purely algebraic approach as in category theory (Spivey, 1989) allows only a few primitive operations to construct the data type. This work shows how to expand such a set of primitive operations in a concise, coherent and practical framework.

A primitive model, concise notation and descriptive terminology are introduced to facilitate reasoning about arrays. The data type is treated independently. The set of primitive operations represents the data type as in abstract data types. Moreover, this set includes new functions that account for common program structures. All features and properties of arrays are readily available via suitable access functions. Yet, the theory is concise. A sound methodology to define primitive operations is prescribed,

instead of freely defining functions ad hoc. Numerous examples are incorporated into the text.

This work balances abstraction and application. Such approaches are promising for future work. The approach taken is based on intuitive notions of arrays. It was decided to work with a familiar model rather than invent a new model of arrays. Although the model is not new, the calculus is new. This calculus permits the concise expression of mathematical algorithms for arrays as in APL. A concise constructive theory is presented that includes useful templates for generic programming in an application area. Formulas that lack many of the indices typically required are stated.

A collection of programs is written which exemplifies the applicability of the theory in programming practice. Cole has proposed a methodology for applications that are not genuine homomorphisms. Many more examples are needed to determine the feasibility of this methodology. Case studies show that programs can be written at a low-level using fine grain parallelism (Eriksson-Bique, 1998, Trichina and Eriksson-Bique, 1999).

In this chapter, some conventions are stated and needed background material is presented. Some of the important theories on arrays and research on relevant and related approaches, especially all influential literature, are reviewed. Further, the methodolgy, goals and contributions are set forth. Lastly, the organization of the text is summarized.

1.1 CONVENTIONS

A conventional style of writing equations is used. Familiarity is beneficial as it is not possible to understand a statement unless the notation is understood in the first place. It is intended to avoid introducing notation. A goal is to promote the methodology, not any particular style.

It is convenient to use natural numbers whenever ranking or referring to a position in a sequence. In particular, indices will always be numbered starting from one. For example, the first item has index 1, the second item index 2 and so forth. Usually it is desired to list more than just a couple items to indicate a pattern, e.g., $2,3,5,7,11,13,17,\ldots$. However, such a general requirement is waived as frequently items are listed in consecutive order and it suffices to list only the first and last ones. For instance, a sequence

$$i_p,\ldots,i_q$$

implies the sequence

$$i_p, i_{p+1}, i_{p+2}, i_{p+3}, i_{p+4},\ldots, i_q.$$

To simplify definitions, a simple "implied" policy is adopted. Often it is desired to formulate a definition that is a function of a particular component of a list, say the kth component. Such a definition is always implicitly understood to hold for all possible values of k for which the definition makes sense. As an example, it is preferable to write one statement

$$f(i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_n)=g(k)$$

in which the values for k are implicitly understood as opposed to writing three statements

$$f(i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_n)=g(k) \text{ for } 2\le k\le n-1,$$
$$f(i_2,\ldots,i_n)=g(1),$$

and

$$f(i_1,\ldots,i_{n-1})=g(n).$$

In general, the initial, e.g., $i_1,\ldots,i_{k-1}$, or the tail part, e.g., $i_{k+1},\ldots,i_n$, of a sequence may be ignored whenever it is possible to interpret the definition accordingly. In the preceding example, a particular kind of sequence which is missing the kth component was specified using a specific function on subscripts. Any such function may be employed. For instance, if

$$f(i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_n)=g(p),$$

where

$$p=s(k)\ge k,$$

is given (e.g., $p=k+3$), then the function f is defined in particular for

$$p=s(1).$$

Any set of elements may be denoted by $\{x\,|\,S\}$, which reads "the set of all elements x such that the statements (one or more) S hold." The **empty set**, denoted by $\varnothing$, is the set $\{\ \}$ without any elements. Recall a few familiar sets of integers are

$$\mathbb{N}=\{1,2,3,4,\ldots\},$$
$$\mathbb{N}_0=\{0\}\cup\mathbb{N},$$

and

$$\mathbb{Z}=\{\ldots,-3,-2,-1,0,1,2,3,\ldots\}.$$

Mostly capital letters by convention denote the names of collections of objects (such as a set `S`) and lower case letters the names of functions (`f`), elements (`a`), and variables (`x`). A variable is sometimes considered a zero-dimensional array. The terms vector and matrix refer to familiar one-dimensional arrays and rectangular arrays with rows and columns, respectively. For example, a row vector, a column vector and a matrix are typically written in the following forms (with possibly different delimiters):

$$\underbrace{[a_1\ a_2\ \cdots\ a_n]}_{\text{row vector}}, \quad \underbrace{\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}}_{\text{column vector}}, \text{ and } \underbrace{\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}}_{\text{matrix}}.$$

1.2 DATA TYPES

Data types are a central theme in functional programming, which deals with functions from one data type to another. A rigorous treatment of data types is beyond the scope of this work. In general, an algebraic theory may be developed for an arbitrary data type.

A data type is a set of values and operations on them. A constant, an expression, a function or a variable, may assume any value from a data type. A (generic) **type variable** matches any valid data value. It is possible to define a data type in various ways. For example, by enumeration of the elements, by specifying the types of all elements, by giving a partial listing of elements and specifying generating functions or relations that constructively generate all other elements. An **abstract data type** is a collection of values that are defined mathematically in terms of operations.

Many familiar types and their properties may be taken for granted. For example, natural numbers possess many well-known properties. However, no attempt shall be made to define natural numbers axiomatically or to derive their properties. The main focus is on large collections of objects and operations on data aggregates. In particular, the **base types** or **primitive types**, i.e., the types of individual elements, are presumed to be known. Moreover, there are suitable (defined or built-in) operations for handling the (data) values.

Functions may be defined on data types using **pattern matching** by associating a prescribed value with each possible pattern of the argument. Consider the classical Fibonacci function `fib`. An argument may match a constant pattern:

$$\texttt{fib}(0) = 1,$$
$$\texttt{fib}(1) = 1.$$

An argument may also match a variable pattern:

$$\mathtt{fib}(n) = \mathtt{fib}(n-1) + \mathtt{fib}(n-2) \text{ for all } n > 1 .$$

For other functions, a pattern may involve a **constructor**. A list is defined using such a compound pattern later in Section 1.4.3.

1.3 MATHEMATICAL BACKGROUND

Recall the essential notion of function (see for example Spindler, 1994, p.701-708), which is an underlying theme in this work. Functions may be defined formally using set theoretic notions. Instead, another usual notion of function is adopted. A **function** f, (loosely synonymously) **operation** or **mapping**, is a rule that assigns to every element in a set $X \neq \varnothing$, which is the **domain**, a unique value. Thus, if f is a function with domain X and $x \in X$ with

$$f(x) = y \quad \text{and} \quad f(x) = z,$$

then

$$y = z .$$

Notice only the domain is specified. If S is any subset of the domain, then any function with that domain is said to be defined on S. If f is a function with domain $X \neq \varnothing$, then the set

$$R = \{ f(x) \mid x \in X \}$$

is called the **range** (or **codomain**). A function f with domain X and range R is usually denoted by

$$f : X \longrightarrow Y,$$

provided

$$R \subseteq Y .$$

The precise specification of the range is often omitted. Frequently, it suffices to specify only the types of the objects. A function with domain $\varnothing$ is ambiguous since there are no elements in $\varnothing$. Therefore, every function $f : \varnothing \to Y$ meets the stated requirements. For consistency, a function with domain $\varnothing$ is allowed provided that the range R is explicitly announced. For definiteness, there is exactly one function with domain $\varnothing$ and range R. A function is well-defined if every element has exactly one image. This means that if an element has more than one representation then the image is the same for all of the representations.

Sometimes the precise specification of the domain is not beneficial. In order to indicate the domain without specifying it exactly, employ a **partial function** $f : X \longrightarrow Y$, which is a function $f : S \longrightarrow Y$, provided $S \subsetneq X$. Mainly, a partial function is used to indicate the types of the arguments. A function f is **one-to-one** or **injective** if $f(x) = f(y)$ if and only if $x = y$. Any injective function has an inverse.

Functions may be defined on sets of values and on **tuples** in other less formal ways. It is convenient to define functions on tuples in a familiar style. A tuple allows components with different types (cf. Mou, 1990a, p. 452). Choose $n \in \mathbb{N}$ and let A_k be any nonempty set for every $k \in \iota n$. The **Cartesian product**

$$A_1 \times A_2 \times \cdots \times A_n$$

is the set of n-**dimensional tuples** with n components:

$$\{(a_1, a_2, \ldots, a_n) \mid a_k \in A_k \text{ for } 1 \leq k \leq n\}.$$

An important property of tuples is that the components are ordered within the tuples. The concept of tuple is a generalization of ordered pairs. Evidently two tuples $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_m)$ are equal if and only if $n = m$ and $a_k = b_k$ for all k with $1 \leq k \leq n$.

Any function $f(p_1, p_2, \ldots, p_{n-1}, p_n)$ defined on tuples of length n may be expressed as a function with up to $n-1$ **parameters,** which are also called arguments. The term parameter refers to an item which is not the main **argument** of the function. For example, in

$$f\, p_1\, p_2 \cdots p_{n-1}\, x = f(p_1, p_2, \ldots, p_{n-1}, x),$$

the parameters are $p_1, p_2, \ldots, p_{n-1}$ and the argument is x. It is sometimes convenient to write the first parameter as a subscript or a superscript:

$$f_{p_1}\, p_2 \cdots p_{n-1}\, x = f(p_1, p_2, \ldots, p_{n-1}, x)$$

or

$$f^{p_1}\, p_2 \cdots p_{n-1}\, x = f(p_1, p_2, \ldots, p_{n-1}, x).$$

A function on an object that is not a tuple is called an **unary function**. A function on pairs is called a **binary function**. It is sometimes convenient to rewrite binary operations which are usually written in **infix form** $a \oplus b$ in either **prefix form or** as unary operations:

$$\oplus(a, b) = a \oplus b = (a\oplus)\, b = (\oplus b)\, a.$$

Using two pairing functions

$$(\curlyvee\!\!\rightarrow \mathtt{x})\,\mathtt{y} = (\mathtt{x},\mathtt{y})$$

and

$$(\leftarrow\!\!\curlyvee \mathtt{x})\,\mathtt{y} = (\mathtt{y},\mathtt{x}),$$

it is possible to define

$$(\mathtt{a}\oplus)\,\mathtt{b} = \oplus \circ (\curlyvee\!\!\rightarrow \mathtt{a})\,\mathtt{b}$$

and

$$(\oplus\mathtt{a})\,\mathtt{b} = \oplus \circ (\leftarrow\!\!\curlyvee \mathtt{a})\,\mathtt{b}\,.$$

By convention, commas and parentheses may be omitted to simplify the appearance of an expression. For example, $\mathtt{f}(1,2)$ may be written instead of $\mathtt{f}((1,2))$ even though the latter expression is technically correct. For a given function such as

$$\mathtt{f} : \mathcal{A} \times \mathcal{B} \times \mathcal{C} \rightarrow \mathtt{Y},$$

the expression $\mathtt{f\ A\ B\ C}$ may be written instead of $\mathtt{f}(\mathtt{A},\mathtt{B},\mathtt{C})$. Notwithstanding, parentheses will be used liberally, preferring clarity over conciseness.

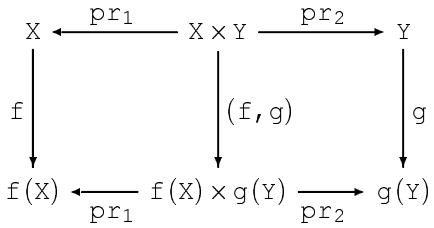
A **higher-order function** takes a function as one of its parameters. A related function **apply**, denoted by @, applies a given function to a given argument and is defined next.

DEFINITION 1.1 For any function $\mathtt{f}$ and any valid argument $\mathtt{x}$ of $\mathtt{f}$,

$$@\,\mathtt{f}\,\mathtt{x} = \mathtt{f}(\mathtt{x}).$$

EXAMPLE 1.1 $@(+2,3) = 5$.

A principal operation applies a pair of functions $(\mathtt{f},\mathtt{g})$ to a pair of objects $(\mathtt{x},\mathtt{y})$ as indicated by the following diagram



where the functions $\mathtt{pr}_1 : \mathtt{X} \times \mathtt{Y} \rightarrow \mathtt{X}$ and $\mathtt{pr}_2 : \mathtt{X} \times \mathtt{Y} \rightarrow \mathtt{Y}$, which map $(\mathtt{x},\mathtt{y})$ to $\mathtt{x}$ and $\mathtt{y}$, respectively, are the usual projections. In particular, $(\mathtt{f},\mathtt{g})\,(\mathtt{x},\mathtt{y}) = (\mathtt{f}(\mathtt{x}),\mathtt{g}(\mathtt{y}))$.

The history of algebra dates back to ancient times when the Babylonians and Egyptians solved linear and quadratic equations. The word algebra in common usage connotes elementary or high school algebra. Actually in this sense algebra is much too complicated a notion, being a mixture of numbers, linear and polynomial equations, variables, arithmetic rules and so forth. More formally, *linear*, *modern*, *higher*, *abstract* or *universal algebra* connotes the study of simple algebraic models and structures using general methods to discover properties not unlike usual addition and multiplication.

The relevant model is a *semigroup* and especially a *monoid*, which are among several classifications of algebraic models. Closely related models include *groups*, which impose additional requirements, and *vector spaces* (see for example Noble and Daniel, 1992, p. 181-186), which have not only additional requirements but also extra elements. Loosely, the word algebra in this work is synonymous with monoid.

A **semigroup** is a model, which is denoted by $(S, \oplus)$, where $S \neq \varnothing$ is any nonempty set and $\oplus$ is any associative binary operator that satisfies

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

whenever $a, b, c \in S$. By definition, whenever a binary operator combines two elements from a set, the result is always an element of that set. The key issue is to be able to determine when two distinct semigroups $(S, \oplus)$ and $(T, \otimes)$ look alike in the following sense: a binary operation in one semigroup corresponds to a binary operation in the other semigroup in such a way that there is an equivalence via some unary operation. In particular, any function $h : S \to T$ that preserves the structure by *distributing* over one of the operators

$$h(a \oplus b) = h(a) \otimes h(b)$$

is called a **homomorphism**. If h is also one-to-one, it is an **isomorphism** (S and T are isomorphic).

A **monoid** is a semigroup $(S, \oplus)$ equipped with an **identity** or **neutral element**, which is denoted by e, that satisfies

$$e \oplus a = a \oplus e = a$$

for all $a \in S$. An identity element is unique as, if e and $\hat{e}$ are identities, then plainly

$$e = e \oplus \hat{e} = \hat{e}.$$

Denote such a monoid by $(S, \oplus, e)$. An onto homomorphism h between two monoids $(S, \oplus, e)$ and $(T, \otimes, f)$ must preserve identity elements

$$h(e) = f.$$

This is evident as for any $g \in S$,

$$\begin{aligned} h(e) \otimes h(g) &= h(e \oplus g) \text{ since } h \text{ is a homomorphism} \\ &= h(g) \text{ since } e \text{ is an identity element.} \end{aligned}$$

A **group** is a monoid $(S, \oplus, e)$ for which every element has a unique inverse, i.e., for $s \in S$ there exists s^{-1} such that

$$s^{-1} \oplus s = s \oplus s^{-1} = e.$$

EXAMPLE 1.2 Consider the two monoids $(S, \oplus, e)$ and $(T, \otimes, f)$. The constant function $K_f : S \to T$ given by $K_f(s) = f$ satisfies

$$K_f(e) = f,$$

and

$$K_f(a \oplus b) = f = f \otimes f = K_f(a) \otimes K_f(b)$$

whence K_f is a homomorphism. In this example, an identity element is denoted by f, which is usually reserved for functions. A popular abstraction characterizes the elements as functions.

Consider next composition of functions. If

$$f : X \to Y \quad \text{and} \quad g : W \to X$$

are functions, then a composition of f and g is

$$f \circ g : W \to Y$$

where

$$(f \circ g)(w) = f(g(w)).$$

Composition is an associative operation. If $h : V \to W$, then

$$(f \circ g) \circ h = f \circ (g \circ h)$$

since

$$\begin{aligned} ((f \circ g) \circ h)(v) &= (f \circ g)(h(v)) \\ &= f(g(h(v))) \\ &= f \circ (g \circ h)(v). \end{aligned}$$

An important observation is that the composition of two homomorphisms is a homomorphism. Suppose $f : X \to Y$ and $g : W \to X$ are homomorphisms such that

$$f(x_1 \oplus x_2) = f(x_1) \ominus f(x_2)$$

and

$$g(w_1 \otimes w_2) = g(w_1) \oplus g(w_2).$$

Then

$$\begin{aligned}(f \circ g)(w_1 \otimes w_2) &= f(g(w_1) \oplus g(w_2)) \\ &= f(g(w_1)) \ominus f(g(w_2)) \\ &= (f \circ g)(w_1) \ominus (f \circ g)(w_2),\end{aligned}$$

which shows that $f \circ g$ is also a homomorphism.

EXAMPLE 1.3 Letting $S_X = \{f \mid f : X \to X \text{ is any function}\}$, $(S_X, \circ)$ is a semigroup.

EXAMPLE 1.4 In fact, $(S_X, \circ, id_X)$ is a monoid as there is always an identity mapping

$$id_X : X \to X$$

given by

$$id_X(x) = x \text{ for all } x \in X.$$

Trivially, id_{S_X} is a homomorphism from $(S_X, \circ, id_X)$ as

$$id_{S_X}(f \circ g) = f \circ g = id_{S_X}(f) \circ id_{S_X}(g).$$

EXAMPLE 1.5 Letting $\phi_X = \{f \mid f : X \to X \text{ is any one-to-one onto function}\}$,

$$(\phi_X, \circ, id_X)$$

is a group as every one-to-one onto function has an inverse.

A semigroup $(S, \oplus)$ is said to be **closed** or to have the **closure property**, which simply means $a \oplus b \in S$ whenever $a, b \in S$. More loosely, an algebra consisting of a set of operations on types is said to be closed if no operation will produce an object that does not have the correct type. In particular, a function defined from one type to another is never ambiguous in a closed algebra.

1.4 A REVIEW OF LITERATURE

Kenneth Iverson demonstrated successfully how mathematical expressions for arrays, not simply expressions for variables as in FORTRAN, could be handled in a programming language. The idea to develop an algebra of programming was proposed by John Backus (1978). The idea to develop a complete theory for data types was proposed by Trenchard More (1979). More's eight principles of data organization are reviewed. It has been established that transformational programming based on structured higher-order functions such as homomorphisms enhances software development. Homomorphisms provide practical models for parallel implementation.

Many theories based on arrays have been proposed (Banger, 1994, Bird, 1989b, Jenkins et al., 1986, Miller, 1993, More, 1979, Mou, 1990a, Mullin, 1988, Reynolds, 1979, Tu, 1986). Existing theories employ different approaches, concepts and notation. Differences can be partly explained by different objectives, e.g., to develop a particular programming language (see for example Jenkins et al., 1986, Mou, 1990a, Tu, 1986) or to develop a programming approach (see for instance Bird, 1989b, Miller, 1993). Early work, such as More's Array Theory, was done before some methods were widely known. Some approaches have not gained wide acceptance probably because they were not well understood or shown to be sufficiently developed for general applications.

1.4.1 CALCULATOR LANGUAGES

An important landmark in the history of programming languages in connection with work on arrays was the introduction of APL (a programming language), which was the first array language (Iverson, 1962). In 1980, the American mathematician Kenneth Iverson received the Turing Award for his work. Definitions in APL did not depend on a technique known as the λ-calculus (lambda calculus) introduced by Alonzo Church in 1951 to abstract notions of functions.

By the time APL was implemented, it was possible to concisely express mathematical algorithms for homogeneous arrays, which could not be done in any other language. For example, arrays could be combined as in linear algebra, properties of arrays could be obtained, and arrays could be restructured without using control structures such as loops to process all of the elements. Many lines of code in nonsymbolic languages are often replaced by a couple symbols in APL.

Although APL has functional features, it has imperative features as well, e.g., indexed assignment. In some sense, APL might be regarded as a calculator for arrays. APL did not permit user-defined functions or even support them inside built-in operations. APL is commonly described as elegant and compact, which is not surprising

since the language was developed by a mathematician. The German writer Johann Wolfgang von Goethe [1749-1832] wrote:

> Mathematicians are like Frenchmen: whatever you say to them they translate into their own language and forthwith it is something entirely different.

An important landmark in the history of programming languages is relational databases that involve a set of tables and operations on (entire) tables (Codd, 1970). Tables, rows and columns are called relations, tuples and attributes, respectively. Such tables and operations are relevant for arrays. For instance, if the attributes consist of real numbers then they may be added to obtain the column sums.

Tu described a functional APL language called FAC (functional array calculator language) that was designed to calculate expressions and to be integrated with other functional languages (1986). Tu introduced functional semantics using the λ-calculus to allow abstract operations, partitioning operations and infinite arrays. FAC objects include arrays, functions, and operators. An array is a function that maps natural numbers to values. Tu also introduces *ragged arrays*, which are arrays of arrays that have the same dimension but not necessarily the same shape.

EXAMPLE 1.6 A three-dimensional ragged array may consist of two arrays. The first array may have two rows:

```
1 2 3
4 5
```

However, the second array may have more columns and one row:

```
6 7 8 9.
```

Lin and Snyder (1993) describe ZPL, an array sublanguage suitable for the MIMD programming model. Many array operations that may be applied to conformable arrays such as addition of two arrays in APL are available in ZPL. Functions can easily be promoted or mapped to the values of the array. Total reductions (e.g., `\`) and scans (e.g., `+\\`) are included as common operators. Partial reductions and scans may be computed using "dimensions." Arrays may be declared and operations may be applied using specified "regions," a representation of index sets which makes it possible to avoid explicit array indexing. Chamberlain et al. (1999) discuss the semantics and benefits of regions in the context of an idealized array language called RL. "Directions" are used to access array values and to define new regions. Chamberlain et al. (1998) explain the benefits for portability and performance provided by ZPL.

Restifo Mullin's dissertation, *A theory of arrays* (1988), is based on the notions of shape, dimensionality, and an indexing operation for arrays. The theory allows expressions and substitutions. The notation closely resembles APL. Hains and Mullin (1991) also present an architecture-independent theory for multidimensional arrays. Mullin and Jenkins (1991) compare this calculus of arrays and Array Theory, which is discussed next.

1.4.2 ARRAY THEORY

Array Theory was developed by Trenchard More (1973, 1979, 1986a, 1986b). More (1973) presented an axiomatization of nested arrays which generalizes the data structure used in APL. More (1979) described an organizational model for nested arrays. Both properties of arrays and operations on them are discussed in detail. *More observed that it is equally plausible to develop a complete theory for arrays as any other theory may be developed such as a theory for sets or a theory of linear algebra.*

More developed a "one-sorted, standard, closed theory of arrays" which basically means there is only one kind of array, i.e., a single array type. The type of elements (monotypic versus polytypic) is also considered. An empty array is "typical," which means that an array inherits its type. More looks at operations that map arrays into arrays as well as operations that transform one operation into another one. Operations are defined "freely" with numerous references to APL. In particular, operations need not be defined in terms of certain primitive operations. More observed that an array may be viewed as

> ... a function that maps an index set of primitive addresses onto an indexed set that contains the items of the array (1979, p. 61).

In 1979, Reynolds also presents the view of arrays as functions:

> ...that an array of real numbers is a variable whose value is a function from subscripts into real numbers (1979, p. 292).

Reynolds discusses mostly one-dimensional arrays. The multidimensional case is mentioned via the Cartesian product.

What does the notion of **data structure** incorporate? The data structure reveals how the data is organized. More's eight principles of data organization (1979, p. 57) are:

- aggregation
- nesting
- well-ordering
- repetition
- smoothness
- valency
- arrangement
- orientation

Next each of these concepts is interpreted. Loosely, an array is a collection of elements. This description adheres to More's principle of aggregation:

> The aggregation of objects into a collection is described by saying that the collection holds the objects as items (1979, p. 57).

An array has a fixed number of axes. This requirement is More's principle of valency:

> The number of axes for an array is an intrinsic property that is called the axis valency, or simply the valency (1979, p. 58).

Along any particular axis, there are a fixed number of elements. This fundamental law is More's principle of smoothness:

> ...all rows (in the general sense) that are parallel to the same axis have the same extent. For example, when boxes of the same shape are stacked evenly on a loading pallet so that the sides and top of the stack are smooth, the result is an array having three axes (1979, p.58).

In addition, elements are located at all possible *nodes* where axes intersect in the regular network of axes. This fundamental law is More's principle of arrangement:

> The same system of distinct, directed axes is perceived at every location in an array....There is no coordinate system of separate axes existing outside the arrangement of items.
>
> The valency of an array is discovered by counting the axes at any location in the array. A location is determined by a position on each axis. Each item of an array occurs in as many different rows as there axes for the array (1979, p. 58).

Identical objects may be stored at different locations in an array. Therefore, it is not merely the value but also the placement that comprises an item of an array. Such useful duplication is characterized by More's principle of repetition.

Axes are ordered, i.e., the axes are ordered by assignment of a unique ordinal number to each axis. Thus, it is possible to look at the array from different viewpoints or orientations. This rule is More's principle of orientation.

In addition, locations along an axis are ordered sequentially. It follows that all locations may be ordered using both the location and the ordinal number assigned to the axis. Furthermore, the elements themselves may be ordered using the order imposed on the locations. Moreover, after removing some "rows," the remaining elements are still ordered. This basic law is More's principle of well-ordering:

> The kinds of collections that are most often used to organize objects are well-ordered in the sense that every nonempty subcollection has, according to some sort of convention, a first item. This principle of well-ordering prevents a collection from being dense or continuous. The process of striking or deleting zero or more items from a well-ordered collection and then closing up the gaps without reordering the remaining items results in a subcollection that is well-ordered (1979, p. 57).

Arrays may be nested, which is More's principle of nesting:

> A collection is an object just as its items are objects. Hence the items of a collection may themselves be collections (1979, p. 57).

Array Theory was used to develop NIAL, a high-level programming language based on nested arrays. NIAL supports several programming paradigms (Jenkins et al., 1986).

Mou and Hudak (1990b, 1988) proposed an algebraic model and notation, which included higher-order functions, for divide-and-conquer algorithms. Mou (1990a) presented DIVACON, a functional language for parallel computation based on divide-and-conquer. Mou described a binary division of arrays along each dimension.

VEC is a functional language of nested arrays. A sublanguage called SIZE requires that all operations are *shapely*, i.e., the shape of the input determines the shape of the result (Jay et al., 1997). This work attempts to recast the theory of arrays using both the syntax of lists and shape analysis. A *cost algebra*, which explains how to combine execution costs, is used to construct a compositional cost calculus.

1.4.3 ALGEBRA OF PROGRAMMING

In his 1977 ACM Turing Award Lecture, John Backus (1978, 1981) advocated a functional programming style over traditional languages. Although the paper has led to more evolutionary than revolutionary changes, one of the fathers of "von Neumann languages" certainly had an impact on the scientific community with this visionary paper. Backus headed the group that produced FORTRAN, undoubtedly a highly successful precursor of imperative programming languages. Backus proposed to separate the data and program structures from the von Neumann machine (Campbell, 1997) for which everything had to be translated into words that would have to pass through the "bottleneck." In von Neumann computers, not only the data but also the commands to access and process the data had to go through the central processing unit (CPU) of the computer a "word at a time."

Like APL, Backus' FP (functional programming) systems were based on functional forms that mapped entire objects into entire objects. These FP systems did not depend on lambda expressions or substitution rules for variables. Backus used sequences, which may be thought of as arrays with only one axis. The algebra of programs that Backus proposed advanced the applicative (functional) programming style over conventional languages that rely on state transitions. Algebraic approaches have been applied to other programming paradigms including object and logic paradigms as well as to composite software systems (Malcolm, 1997).

John McCarthy demonstrated the importance of recursion and the close connection between the program structures and the data structure of lists in LISP (list processing language). An applicative style of programming using recursion is presented in (Wadler, 1981). Transformational programming and applications were discussed in (Paige, 1983). Strategies in transformational programming to manipulate algorithmic expressions in order to rewrite them in a more efficient form starting from a suitable specification of a problem are discussed in (Bird, 1984, 1986). A functional approach based on recursive equations and sequences, which are like lists (or one-dimensional arrays), was employed (Bird, 1984). All functions were defined in terms of a (sequence) relation. In particular, definitions were not made freely as in Array Theory.

The theory of lists is a calculus for lists (Bird, 1988, 1990, Bird and Meertens, 1987, Bird, 1987, 1989a,b, Meertens, 1986, Skillicorn, 1990, Spivey, 1989). Parallel programs are sometimes "optimized" into sequential programs (see for example Bird, 1989a). The theory relies heavily on homomorphisms. Next a summary of some of the main constituents of the theory is given. What is germane is the calculus, not the style

or the notation which is slightly modified in this review.

Let α be any data type. The theory adds the new data type $[\alpha]$. It is the data type for lists of elements of homogeneous type, i.e., all elements have the same type α. In other words, a list has type $[\alpha]$ if all elements in the list have type α.

Given two lists of the same type, "concatenate" or join them together via a **type constructor**, which is denoted by $+\!\!+$. For example,

$$[a_1, \ldots, a_n] +\!\!+ [b_1, \ldots, b_m] = [a_1, \ldots, a_n, b_1, \ldots, b_m].$$

An empty list $[\,]$ has no elements. For any list A,

$$[\,] +\!\!+ A = A +\!\!+ [\,] = A.$$

Evidently the empty list $[\,]$ is the identity element for the binary operator $+\!\!+$. Moreover, the binary operator $+\!\!+$ is associative. Hence, $([\alpha], +\!\!+, [\,])$ is a monoid.

An important higher-order function (**HOF**) for lists is `map` which is denoted (in this Chapter) by $*$. The function `map` applies a given (parameter) function

$$f : \alpha \to \beta$$

to every element of a given list $[a_1, \ldots, a_n]$ whose elements have type α:

$$\star f([a_1, \ldots, a_n]) = [f(a_1), \ldots, f(a_n)].$$

Thus, $\star$ takes a function f defined on the data type α and produces a new function

$$\star f$$

which is defined on the new data type $[\alpha]$. It is a **polymorphic** function as it is defined for a list of elements of any homogeneous type.

Parametric polymorphism concerns functions that are defined for various types. Studying such functions makes it possible to study laws and properties that hold more generally. Bird (1987) defines $\star$ via the equations

$$\star f([\,]) = [\,],$$
$$\star f(X +\!\!+ Y) = (\star f X) +\!\!+ (\star f Y).$$

The higher-order function $\star f$ takes the monoid

$$([\alpha], +\!\!+, [\,]) \qquad \text{to the monoid} \qquad ([\beta], +\!\!+, [\,]).$$

An important identity is **map distributivity**:

$$\star(f \circ g) = (\star f) \circ (\star g).$$

Another important polymorphic function is reduce which is denoted by $/$ and which inserts a given associative binary operator $\oplus$ between all of the elements in a given list $[a_1, \ldots, a_n]$ whose elements have type α:

$$/\oplus\,[a_1, \ldots, a_n] = a_1 \oplus \cdots \oplus a_n.$$

If $\oplus$ has an identity element e then $(\alpha, \oplus, e)$ is a monoid. In this case,

$$/\oplus e\,[\,] = e,$$
$$/\oplus e\,[a] = a,$$

and

$$/\oplus e\,(X{+\!+}Y) = (/\oplus e\,X) \oplus (/\oplus e\,Y).$$

The function $/$ lifts $\oplus$ from the monoid

$$(\alpha, \oplus, e) \quad \text{to the monoid} \quad ([\alpha], {+\!+}, [\,]).$$

A fundamental theorem is the **homomorphism lemma** which states that every homomorphism on a monoid $([\alpha], {+\!+}, [\,])$ is a composition of a map and a reduce operation. In particular, any homomorphism may be computed by applying some reduce operation after applying a map. Another important fact is the **unique extension property** which asserts that whenever the type constructor $+\!+$ is replaced by an associative operator $\oplus$ with an identity element e, the operation is completely determined up to some function on the elements. This means that a homomorphism h between the monoids $([\alpha], {+\!+}, [\,])$ and $(S, \oplus, e)$ is completely specified whenever the operation h is defined for all lists containing only one element. For example, if for all elements a with type α,

$$h\,[a] = u\,(a),$$

where u is a well-defined function, then h is uniquely determined.

An important function is **catamorphism**, which is a unique homomorphism from one data type to another that distributes through the type constructors. Programming using catamorphisms is discussed in (Hughes, 1991). Functions may be defined in a "standard way" as homomorphisms. Some examples from the theory of lists are given next.

EXAMPLE 1.7 The number of elements in any list is given by the homomorphism

$$\# = (/+) \circ (\star K_1),$$

where the constant function $K_1\,(x) = 1$.

EXAMPLE 1.8 The filter function $\lhd$ selectively filters out items and is given by

$$\lhd \mathtt{p} = (/+\!\!+) \circ (\star \mathtt{f_p}).$$

where

$$\mathtt{f_p}(\mathtt{x}) = \begin{cases} [\mathtt{x}] & \text{if } \mathtt{p}(\mathtt{x}) \\ [\ ] & \text{otherwise} \end{cases}.$$

EXAMPLE 1.9 The function that reverses a list is given by

$$\mathtt{reverse} = (/\oplus) \circ (\star [\cdot])$$

where

$$\mathtt{a} \oplus \mathtt{b} = \mathtt{b} +\!\!+ \mathtt{a}$$

and

$$[\cdot](\mathtt{a}) = [\mathtt{a}].$$

Promotion theorems reveal important properties and laws of catamorphisms. The following promotion laws hold:

$$\begin{aligned} (\star \mathtt{f}) \circ (/+\!\!+) &= (/+\!\!+) \circ (\star(\star \mathtt{f})), \\ (\lhd \mathtt{p}) \circ (/+\!\!+) &= (/+\!\!+) \circ (\star(\lhd \mathtt{p})), \\ (/\oplus) \circ (/+\!\!+) &= (/\oplus) \circ (\star(/\oplus)). \end{aligned}$$

Next, look at a few polymorphic functions to get a sample of other functions from the theory of lists. The function $\twoheadrightarrow$ is called a **left reduce** or **foldl** operation that groups from left to right using a supplied binary operator $\oplus$ and element $\mathtt{e}$:

$$\twoheadrightarrow \oplus \mathtt{e}\, [\mathtt{a}_1, \ldots, \mathtt{a}_n] = (\cdots((\mathtt{e} \oplus \mathtt{a}_1) \oplus \mathtt{a}_2) \oplus \mathtt{a}_3 \cdots) \oplus \mathtt{a}_n.$$

The function $\twoheadleftarrow$ is called a **right reduce** or **foldr** operation which groups from right to left using a furnished binary operator $\oplus$ with value $\mathtt{e}$:

$$\twoheadleftarrow \oplus \mathtt{e}\, [\mathtt{a}_1, \ldots, \mathtt{a}_n] = \mathtt{e} \oplus (\mathtt{a}_1 \oplus (\cdots \mathtt{a}_{n-2} \oplus (\mathtt{a}_{n-1} \oplus \mathtt{a}_n) \cdots)).$$

Whenever $\oplus$ is associative,

$$\twoheadrightarrow \oplus \mathtt{e} = \twoheadleftarrow \oplus \mathtt{e},$$

which is the first duality theorem. Two functions that compute segments of lists are given by:

$$\mathtt{inits}\, [\mathtt{a}_1, \ldots, \mathtt{a}_n] = [[\ ], [\mathtt{a}_1], [\mathtt{a}_1, \mathtt{a}_2], \ldots, [\mathtt{a}_1, \ldots, \mathtt{a}_n]],$$

and

$$\begin{aligned}&\mathtt{tails}\,[a_1,\ldots,a_n]\\ &\qquad = [[a_1,\ldots,a_n],[a_2,\ldots,a_n],\ldots,[a_{n-1},a_n],[a_n],[\,]].\end{aligned}$$

Principle uses of folds are discussed in (Hutton, 1999).

To compute prefix sums using a left reduce operation, use the following **left accumulate** (or **scanl**) function:

$$/\!\!\!\!\rightarrow \oplus\, e\,[a_1,\ldots,a_n] = [e, e\oplus a_1,\ldots,(\cdots((e\oplus a_1)\oplus a_2)\oplus\cdots)\oplus a_n].$$

The following **accumulation lemma** is well-known:

$$/\!\!\!\!\rightarrow \oplus\, e = \star(\rightarrow \oplus e)\circ \mathtt{inits}.$$

Consider how to construct the data type $[\alpha]$ and classify operations in a more complete way (Skillicorn, 1993, 1994a,b, 1992, Spivey, 1989). The constructor $+\!\!+$ by itself cannot be used to build up all objects with type $[\alpha]$. So add another generating relation

$$[\cdot] : \alpha \rightarrow [\alpha]$$

which builds singletons by enclosing the argument in brackets $[\cdot]\,(a) = [a]$.

The following list includes the generator $[\,]$ which is the empty list and all other constructors:

$$\begin{aligned}&[\,],\\ &[\cdot] : \alpha \rightarrow [\alpha],\\ &+\!\!+ : [\alpha]\times[\alpha] \rightarrow [\alpha].\end{aligned}$$

Any list may be generated using only these constructors. It turns out that this "construction" is sufficient to define homomorphisms on lists. In particular, an operation $h : [\alpha] \rightarrow Y$ is a list homomorphism if it can be defined via the equations

$$\begin{aligned}h[\,] &= e,\\ h\circ[\cdot] &= u,\\ h\circ +\!\!+ &= \oplus\circ(h,h),\end{aligned}$$

for some value e, unary function $u : \alpha \rightarrow Y$ and associative binary operator $\oplus : Y\times Y \rightarrow Y$. The equations may be dispensed with to obtain the basic structure:

$$\begin{aligned}&e,\\ &u : \alpha \rightarrow Y,\\ &\oplus : Y\times Y \rightarrow Y.\end{aligned}$$

This structure has a similar form as the list of constructors.

To show that a map $\star \mathtt{f}$ is a list homomorphism, present the structure thusly:

$$[\,],$$
$$[\cdot] \circ \mathtt{f} : \alpha \to [\beta],$$
$$+\!\!+ : [\beta] \times [\beta] \to [\beta],$$

where $\mathtt{f} : \alpha \to \beta$ is any function. Letting $\mathtt{e}$ be the identity element for the associative binary operator $\oplus$, a reduction $/\oplus$ is also a list homomorphism:

$$\mathtt{e},$$
$$\mathtt{id} : \alpha \to \alpha,$$
$$\oplus : \alpha \times \alpha \to \alpha.$$

Bird (1989b) presented a calculus for various data types such as arrays, lists, and trees. In this paper, two-dimensional arrays with elements in $\{0, 1\}$ are considered. Bird employs a couple constructors which resemble the concatenation operator in the sense that they function basically in the usual way of adjoining matrices. A particular algebraic structure (binoid) is classified and the notion of homomorphism is "weakened." In addition, several operations and laws are developed. In Richard Miller's thesis *A Constructive Theory of Multidimensional Arrays* (1993), the "join operators" are generalized for n-dimensions and the data type is defined using axioms.

1.4.4 ALGEBRAIC METHODS FOR DIFFERENT DATA TYPES

The terms "squiggol" and "Bird-Meertens Formalism" (**BMF**) have been used to characterize programming paradigms based on type constructors and higher-order functions, especially homomorphisms and catamorphisms (Gorlatch and Lenguaer, 1995, Hu et al., 1998b, Skillicorn, 1990, 1993, 1992). *The Squiggolist* (J. Jeuring, editor), last published in 1991 by CWI Amsterdam, was a journal for works in "Constructive Algorithmics" or BMF. The abbreviation BMF may have originated in a joke in April 1988 comparing BMF with BNF (Backhouse and Hoogendijk, 1993).

A BMF may be regarded as either a calculus in transformational programming for a *particular* data type or a more general theory on structured data types which uses basic concepts from category theory and universal algebra to unify sets of equations for *different* data types (Campbell, 1997, Cole, 1993, Hu et al., 1994, Jansson, 2000, Malcolm, 1989, Meertens, 1996). The algebraic approach was developed long ago (Burstall and Landin, 1969, Fokkinga, 1996, Goguen, 1977, Hagino, 1987, Hughes,

1991, Lambeck, 1968, Lehmann and Smyth, 1981, Malcolm, 1990a,b, Manes and Arbib, 1986, Smith and Plotkin, 1982, Walters, 1991). The theory of lists has been explored in the context of category theory (Spivey, 1989). Skillicorn provides an introduction to **categorical data types** which are a generalization of abstract data types. A general textbook describing an algebraic approach to programming appeared more recently (Bird and de Moor, 1997).

Roland Backhouse showed that notation in a calculational theory on structured data types is compact and expressive (Backhouse, 1989). He compared such notation and the Einhoven quantifier notation, which are not equally expressive. He discussed **lists**, **trees**, **bags** that are unordered lists, and **sets** that are bags without duplicates or repetitions. He also considered a "lifting" mechanism to raise functions to higher-order functions. A relational theory of data types appeared in (Backhouse and Hoogendijk, 1993).

Malcolm (1990a, 1990b) presented an algebraic theory of data types that consists of a calculus for program transformations. Notation was introduced to define data types via initial algebras and final coalgebras. Malcolm advocated both notation that facilitates formal reasoning and a methodology to define functions on data structures. Malcolm introduced notation for catamorphisms and discussed ideas related to **polytypic** functions, which are defined by induction on the structure of data types (Hinze, 1999, Jansson, 2000).

Many problems can be expressed without specifying a particular data type. A polymorphic calculational theory in a relational framework is discussed in (Backhouse et al., 1991). **Shape polymorphism** and **shapely types** are presented in (Jay and Cockett, 1994). Shapely types are data types that consist of separate data and shape parts. Alexander et al. (1995) describe ADL, a polymorphic non-recursive language designed for high-order operations on aggregate structures.

The notion of "monadic folds" for structuring programs using a system of data types was proposed in (Meijer and Jeuring, 1995). The derivation of "generic" functional programs for different data types is explored in Bird et al. (1996), which generalizes the theory of segments. Jansson introduced polytypic programming using a functional language extension, PolyP, which includes a construct for user-defined data types (Jansson, 2000).

Bird and Meertens (1998) discussed a reduction that can be applied to "nested" data types and a semantics for them in a calculational theory. Hinze (1999) proposed an alternative approach to extend polytypism to arbitrary data types. Okasaki (1998) used nested data types to devise purely functional data structures. A method to construct

generalized folds for nested data types that satisfy certain properties was introduced in (Bird and Paterson, 1999b) and de Bruijn notation for nested data types was considered in (Bird and Paterson, 1999a).

Banger and Skillicorn (1993) presented a foundation for "flat arrays," which was influential. There are two properties of flat arrays: (1) the shape and (2) the content as an infinite sequence. In this theory, indexing is not considered to be a primitive operation. There are two generating relations: **scalarize** that builds "0-dimensional arrays" and **add_dimension** that adds a dimension. Flat arrays were recast using a categorical data type construction (Banger, 1992, 1994, Banger and Skillicorn, 1992). A category is comprised of **objects**. For every pair of objects, there is a class of **morphisms** or **arrows** (maps or transformations) (Pierce, 1991). Often the objects are taken to be some mathematical model such as monoids. The arrows are usually required to meet some natural conditions (or axioms). For instance, there is always an identity mapping from any object to itself, compositions are defined and the associative law holds for compositions. As the name suggests, interesting morphisms preserve the mathematical structure of the objects.

1.4.5 PARALLEL COMPUTATION

The sequential model based on the von Neumann machine has remained highly successful. The diversity of popular parallel models has made it difficult to achieve the same degree of success. For example, sequential programs are portable, whereas parallel programs are not. Survey of models of parallel computation are given in (Campbell, 1997, Skillicorn and Talia, 1998). Skillicorn and Talia (1998) discuss the benefits of "homomorphic skeletons."

Formal methods have been shown to enhance software development. Parallel computation based on structured higher-order functions has been studied extensively (see for instance Banger and Skillicorn, 1993, Bratvold, 1994, 1992, Cole, 1995, Duff and Harrison, 1996, Gorlatch, 1996a,b, Gorlatch and Lenguaer, 1995, Hu and Takeichi, 1999, Hu et al., 1998a, 1997, Jay et al., 1997, Kumar and Skillicorn, 1995, Skillicorn, 1990, 1993, 1994a,b,c, 1995, Skillicorn and Cai, 1994, 1995, Skillicorn, 1992). Skillicorn (1990) argued that a formalism which is based on homomorphisms permits an architecture-independent complexity theory over four classes of parallel architectures, namely SIMD (single instruction, multiple data), tightly coupled MIMD (multiple instruction, multiple data), hypercuboid and constant valence MIMD computers. He showed that such a formalism addresses many problems that plague parallel computing. Programs have a longer life span and are portable.

Bratvold (1994) showed how to develop a skeleton-based functional program from a problem specification using the homomorphism lemma. Bratvold (1992) also outlined compilation stages for higher-order functions in functional languages. The functional paradigm is well suited to parallel processing as there are no side effects. A framework based on polytypic programming to develop efficient parallel programs was given in (Hu et al., 1998b).

Arvind et al. (1989) and Barth et al. (1991) added programming constructs to ID to make programs more "declarative," parallel and storage efficient. ID is a functional language which includes algebraic types and definitions. ID allows for the construction of functional arrays using an "accumulator."

1.4.6 CRITICAL ANALYSIS

It is evident from the literature that many questions on developing a theory of arrays have remained unanswered. Many of the existing theories are based on solid mathematical foundations. None of the theories that utilize algebraic structures seem to provide a complete treatment for arrays as Trenchard More proposed or a general programming language as John Backus advocated. When homomorphisms are employed, only a couple primitive operations are used no guidelines are provided on how to extend the set of primitive operations in a coherent and concise framework.

More's organizational model (1979) is sound. However, operations are defined freely without employing algebraic structures. Mullin's theory (1988) defines a primitive indexing operation, which seems appropriate for arrays. Although Mullin's theory is based on a sound architecture-independent approach, operations should not be defined in terms of an indexing operation. A more satisfactory theory incorporates algebraic structures in an algebra of programming.

Banger and Skillicorn (1993) model arrays using an infinite sequence, which is not always needed or beneficial. The operations (as they are programmed in Banger and Skillicorn, 1993) do not seem to be characteristic of arrays. For instance, indexing an array is not a primitive operation. Too few program structures are discussed. Reductions are defined via a single operation applied to each dimension recursively, which is not surprising since homomorphisms are only defined in terms of add_dimension. Other possible definitions of homomorphisms are not discussed. The set of primitive operations provided does not seem to adequately represent the data type. The construction of the set of primitives seems to be based solely on algebraic considerations without regard to programming and how the data type is often used. It is not explained how to extend the theory in order to develop a complete programming language that

provides adequate tools so that programming various operations on a specific model of arrays is not too difficult. In essence, the algebra is more important than the data type.

Banger and Skillicorn (1992) state a theorem, a lemma and a couple properties. Banger and Skillicorn (1993) describe the same model with nesting. These statements reformulate existing and well understood theoretical results. The basis for the data type construction and the programming methodology using catamorphisms and higher-order functions, such as map and reduction, were already known (see for example Malcolm, 1989, 1990a,b).

Banger and Skillicorn (1992) describe only a couple programs and basic operations as in APL. These programs are presented without derivation or optimization, and do not include complete definitions. The theory should provide more useful programming templates to account for common program structures.

Bird (1989b) deals only with two-dimensional arrays with homogeneous elements. Notation for handling multidimensional arrays is not presented. Only a couple primitive operations are defined. All operations which are not needed to construct the date type must be defined in terms of the constructors, including operations which help to define the objects and which are usually regarded as primitive operations. For instance, operations cannot be applied axially and instead must be applied to the planes of the array, as if it were only constructed of a list of planes, which implies the data type is inferior to lists. In practice, it is difficult or impossible to write all programs in terms of a couple primitive operations, which is a serious shortcoming and justifies why a larger set of primitive operations is needed. Bird's work pertains mainly to list operations. Since arrays can only be partitioned along rows or columns, operations can only be applied to rows or columns.

Miller's generalization (1993) of Bird's theory for multidimensional arrays does not seem to treat the set of constructors or the algebra in a fundamentally different way. The algebraic treatment of the data type is sound. The scope of the work is nonetheless narrow dealing mainly with a small set of definitions. Different data structures allow for different program structures. Generalizations of other theories is not necessarily sufficient to build up a completely independent data type.

Polytypic programming (discussed in Hinze, 1999, Jansson, 2000, Meertens, 1996, Pfeifer and Rueá, 1999) attempts to answer different questions. Methods to install different data types in general algorithms are not investigated. Instead, the approach taken is to focus on a specific data type for particular applications.

1.5 APPROACH AND GOALS

Data structures for arrays are popular with numerous applications; consequently, useful models for multidimensional arrays along with simple complete theories for array expressions are important. Software development is costly. Structured higher-order functions and in particular homomorphisms have been shown to enhance software development. Mathematical approaches that focus on essential details of computations instead of specifying unnecessary implementation details foster optimization. The benefits of algebraic approaches have been well established in the literature (see for example Axford and Joy, 1993, Backus, 1978, Bird, 1989a, Manes and Arbib, 1986, More, 1979, Mou and Hudak, 1988, Skillicorn, 1990).

The main task is to construct a set of primitive operations for programming with multidimensional arrays which does not depend on a particular architecture and which fosters software development. Ideally, programs could be derived starting from a specification and could be optimized for a particular implementation. A specific model of arrays should be given which is a primitive concept and operations should be defined in a structured way without using indices as in Mullin's theory of arrays. Notation should be provided which makes it easier to develop the theory and guidelines on how to define primitive operations should be stated (as advocated in Malcolm, 1990b). In particular, definitions should not be made freely or ad hoc so that the language does not become unwieldly. A sufficient number of primitive operations should be provided to represent the data type as in abstract data types so that programming templates can be reused to solve different problems in an application area. All parts of the theory should be verified formally correct.

Constructors and higher-order functions should be utilized to achieve an algebra of programming that permits the derivation and optimization of programs as in the theory of lists. The theory should include useful homomorphisms, laws and properties. A theory is sought which permits the concise expression of mathematical algorithms for arrays as in APL. Formulas should be provided for doing basic computations. LISP and recursion fit together due to uniformity in the data structure and the program structure. Arrays have a special data structure and a goal is to reach for a similar uniformity in the data structure for arrays and common program structures.

In functional programming, arrays have been treated as essentially complicated lists. Arrays should be treated as an independent data type as More advocated. It is not satisfactory to provide only a couple primitive operations and a couple definitions of homomorphisms for every data type. While algebraic structures are important, the data type is paramount. The theory should account for common program structures.

The theory should not be limited to some minimal set of operations, regardless of the completeness or elegance of the algebra; otherwise, programming is severely limited.

Arrays are so natural that it might seem new notation is not needed. After all, they have been around long enough and are certainly quite popular data structures. There are seemingly few if any absolutes and perhaps one of them has been provided by the mathematician and philosopher Alfred North Whitehead [1861-1947]:

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race (Davis and Hersh, 1981).

Existing notation for arrays is not entirely satisfactory. It is frequently tedious to write formulas and especially prove properties and laws using existing notation. Everyone has an intuitive understanding of arrays. Yet, there does not seem to be a universally accepted notational system for working with multidimensional arrays that does not require reading and writing numerous indices. It seems that in many cases, many more indices are used than are actually needed.

EXAMPLE 1.10 Consider a $n \times n \times n \times n \times n$ matrix. It is possible to write formulas and programs using several indices $(i_1, i_2, i_3, i_4, i_5)$. Proving the program is correct using so many indices is tiresome and even if someone would successfully undertake the task, it would require considerable stamina to verify the proof itself is correct, which diminishes the accomplishment. There are many well-known problems dealing with so many indices. While many indices may be needed to define general operations, to define a particular operation, often much less information is needed. For instance, applying a row operation involves only elements in a row and so only one index is changing.

An obstacle is the encumbrance due to the number of indices. This obstacle can be overcome since only a relatively small number of indices are actually needed to define operations. This fact follows since common operations possess a highly regular structure.

A satisfactory solution does not eliminate the indices or treat arrays as an inferior data type. The indices are useful and give purpose to the data type. A typical treatment of arrays reduces them to essentially complicated lists with some extraneous features. Regrettably, this solution reduces the effectiveness of the data type. Even when the objects are constructed so that they resemble arrays, the operations do not seem to be characteristic of arrays.

Dissatisfaction arises partly due to the adaptation of the data type to fit into a theory instead of developing a theory to suit the data type. A goal is to raise arrays to their full potential by taking advantage of all of the indices without having to handle them excessively. To this end, new notation is presented.

A formalism is sought which is a basis to develop theory and applications. The theory should be adequate at least to develop an application. At the highest level of achievement, a theory is a calculus which expounds an independent data type.

1.6 CONTRIBUTIONS OF THIS THESIS

The data type is treated in an architecture-independent way for portability and optimization. A set of primitive operations is constructed and a methodology to construct such a set is prescribed. New definitions that simplify the theory (definitions and proofs) and novel primitive operations that account for common program structures are introduced. A constructive theory is developed which provides programming templates for generic programming. Explicit formulas are given that lack many of the indices typically required. The applicability of the theory to programming is illustrated by examples. Practical programming techniques are explained. The use of homomorphisms as programming templates is demonstrated. In summary, a concise and coherent architecture-independent theory of multidimensional arrays is presented which

- defines the data type independently,
- prescribes a methodology to define all primitive operations,
- furnishes a calculus for defining operations,
- classifies “array” homomorphisms,
- states concise formulas for common mathematical algorithms,
- explains practical programming techniques, and
- permits the derivation and optimization of programs.

This work balances abstraction and application. The calculus is useful for both research and programming. It is not necessary to translate an algorithm from a theory into a programming language and to prove the translation is correct. A goal is to enhance not only software development but also research. This goal is reached by providing a calculus, which makes it easier to reason about the data type, focuses on essential details and properties of computations and aids in programming.

1.7 ORGANIZATION OF THE TEXT

In the sequel a fairly complete and concise theory for the data type is developed. First, notation and terminology are introduced to define and reason about the data type. Second, a set of primitive operations is constructed to represent the objects and to recognize common program structures. Third, homomorphisms are classified and laws, properties, identities and sets of equations are stated. Fourth, the applicability in programming practice is demonstrated.

In Chapter 1, relevant literature is reviewed and the task is explained. In Chapter 2, elementary concepts and informal notions of arrays are introduced and a methodology to define primitive operations is prescribed. In Chapter 3, the foundation is established by presenting formal concepts of arrays and defining all primitive operations. In Chapter 4, the consequences of the definitions are formally investigated and a constructive theory of arrays is developed. In Chapter 5, programs are written to confirm the applicability of the theory in practice. In Chapter 6, case studies are done to demonstrate the usefulness of programming templates. Finally, in Chapter 7, a new programming template is presented and future research is discussed.

CHAPTER 2

A SURVEY OF ARRAYS

Some basic concepts, functions and symbols which are used in the sequel are introduced. Elementary functions help to simplify formal definitions and proofs. A methodology to define primitive operations is prescribed. In addition, the following topics are discussed informally: type and representations of an array, implementation and feasibility. According to Webster's dictionary, an array is a regular arrangement, or an orderly listing, or a number of mathematical elements arranged in rows and columns. In conventional programming, arrays are collections of homogeneous items, i.e., elements with the same type, which are differentiated via indices. It seems arrays are natural objects even if they are devices of humans as Leopold Kronecker [1823-1891] wrote:

> The whole numbers has the Dear God made; all else is man's work.

2.1 HOW TO DEFINE PRIMITIVES

Consider the notion of data structure for multidimensional arrays. What is data? It is something known or assumed. For example, real numbers may be assumed to exist by axiom and represented by some useful means. Data would be useless if it could not be processed in some way (without some functions defined on that data). However, do not consider such functions or such data. Instead, focus on large data aggregates consisting of data and possessing some peculiar structure, which is the data structure.

Is there a set of basic operations on such data aggregates? Objects are partly defined by the operations that can be performed on them, much like abstract data types (see for example Aho, 1982). Values like Booleans simply cannot be properly interpreted without operations such as logical functions. Defining an object without operations is like finding a marvelous tool without knowing how to use it.

What primitive operations can be performed on arrays? It should be possible to access any item of an array directly. Such an indexing operation sets arrays apart from other models such as lists, sets, and trees. Especially it is not necessary to process an array starting from some node and proceeding in some systematic fashion, which is appropriate for lists, sets and trees.

It should be possible to extract blocks such as rows or columns as they are used often in programs. Fundamental properties of an array should be readily available via suitable access functions. It should not be a programming exercise to obtain such information as the number of axes, the number of elements, or the extent of an axis. In short, it should be possible to access any information about data organization.

It seems easy to see how to extend operations like addition but it is less clear how to extend operations like multiplication. Should multiplication for matrices be defined in terms of an inner product? It is a reasonable question since every element in the resulting array is obtained via an inner product. Anyway, multiplication is a special application. A key observation is that typical operations are defined axially.

Presumptively, operations are defined for the types of elements in an array, e.g., if elements are integers then presumably there are usual operations defined (built-in) on integers. It should be possible to say, for example, perform this given operation along this axis. For instance, sum the columns or rows of an array.

It is also possible to think about arrays as objects that may be manipulated as in elementary algebra. Any array is a single value that can be manipulated like other familiar values such as integers or reals, although the operations may be less intuitive or more complicated. For example, operations might resemble those of a vector space, e.g., addition of "vectors" is both commutative and associative, there is a zero element, multiplication of vectors by scalars is defined and there are distributive laws. Nevertheless, the algebra is closed in the sense that operations on objects always yield objects of an explicit type in the algebra.

Much is known about matrices. This knowledge is applicable to higher dimensions. To give an analogy, in real analysis (a branch of mathematics), $\mathbb{R}$ is like $\mathbb{R}^n$ in the sense that important concepts have counterparts in both spaces, e.g., open sets, convergence, differentiation, ... For example, it seems evident that it is possible to perform operations along any axis, not just rows.

Primitive operations should not be freely defined as in Array Theory. A number of constraints are placed on the primitives. It is proposed to develop a theory for particular applications so that the number of primitive operations can be kept small. It is recommended to arrange three classes of primitive operations:

- One class of functions, namely the **constructors** (creation), build up the data type. The purpose of the constructors is to induce algebra into programming.
- Another class of functions is the **destructors** (selection). These functions complement the constructors in order to define all other features provided by the data type. Together, the constructors and destructors replace any representation of the data type, i.e., form the access functions.
- An important class is the **abstractors**, which are abstract operations. These functions are defining operations for the data type. They give a purpose for the data type and explain the data structure. Abstractors include important higher-order functions and partitioning operations. These primitives are essential tools to build up the theory. They facilitate programming by introducing recognizable patterns of computations.

The intention is to replace the model by primitive operations (see for example Mou, 1990a, p. 452). The model is needed not only to specify the objects but also to define the primitive operations. A primitive operation is intrinsically related to the data structure. A primitive operation is defined for all objects and does not depend on the data values. A primitive operation does not perform some specific computation on the data, although abstractors, which are higher-order functions, may apply given operations in ways that are special for the data structure. Any parameters to primitive operations must either correspond directly with the data structure or be the parameters of higher-order functions. Primitive operations are

> useful and powerful primitive functions rather than weak ones that could then be used to define useful ones (Backus, 1978, p.620).

The primitive operations should complement each other. A general rule of thumb: define a set of "orthogonal" functions, which means redundancy is kept to a minimum and the number of primitives is kept small. An operation should not be included unless it is a common one or builds up the theory algebraically.

The constructors generate new data types from base types or build-up larger data aggregates from smaller ones. The constructors form a major part of the foundation upon which an algebra is constructed. There should be an adequate number of constructors to capture all of the essential ways the objects are used in the theory, especially in the sense of partitioning both operations and objects. In addition, there should be sufficiently many of them so that it is possible to generate all objects.

There should be an adequate number of destructors to handle all of the various parts which comprise the objects. Mainly it should be possible to select fundamental

properties and the components that are the essential "building units," i.e., not arbitrary subcollections but distinct, entire and useful parts. In addition, there should be enough destructors for common selections so that programming is not too difficult and optimization could take into account frequently used operations. The destructors complement the constructors in order to define all other features of the objects in terms of the data structure. Together, the constructors and destructors totally replace any representation of the objects.

The abstractors are essential parts of equations that arise in theory. They are closely interwoven with the data structure. It is because of these operations that the data structure (object) is used. The purpose of abstractors is to take advantage of the known data structure, to increase the usefulness of the data type and to raise expectations of the theory in general.

Definitions pose challenges as there are frequently choices and it is not always clear which choices are best. Enough primitives are needed so that programming is not too difficult. Try to capture all of the essential aspects of the model. In particular, focus on common operations. On the other hand, do not endeavor to define every conceivable operation that may be useful.

Algebraic notions should be used to define operations. For instance, there are often many ways to define a function but not so many ways to define a left inverse of a given function. Just as programs are composed in terms of the primitives, the primitives may be defined in terms of more elementary functions.

It is recommended that all primitive operations be defined independently so that subsequent revisions will require a minimal amount of effort. This means that the primitives should not be defined in terms of each other. Any operation that is not primitive may be defined in terms of *any* of the primitive operations. In particular, it is not required to define any operation in terms of only the constructors, although such definitions are preferred.

A theory should list some rules that prescribe how objects may be specified as arguments and parameters of functions. Although operations are defined in terms of primitives, parameters and arguments in general may be defined using any model or specification guidelines. There should be specific and simple ways to specify objects.

It is not necessary to try to program everything in terms of primitive operations, starting from the specification of a problem. Otherwise, the program constructions become unwieldy. In particular, some pre-processing and post-processing is allowed. Such processing could be in the form of assignment statements. Parallel programming constructs (such as `SEQ` and `PAR`) could be utilized so that only essential details of

computations are prescribed, which is a unifying theme.

It is not essential to focus only on homomorphisms or to try to classify precisely every algebraic structure in every circumstance. The idea is to espouse algebra but not to attempt to subsume programming as an application of any particular algebraic model. It is enough to be able to say precisely how an operation may be performed in a particular instance.

When constructing a set of primitive operations, focus on the programming paradigm. Using available programs, known laws and identities, attempt to compose a solution to a given task starting from the specification of a problem. If a solution cannot be obtained using existing tools, then begin to write only those subprograms which are needed and which combined with existing software will yield a satisfactory solution. If possible, employ only the constructors. If necessary, other primitives may be employed. Any programming paradigm such as recursion may be used to define needed basic operations. It may be required to add some additional control structures but such structures should be limited in number. In principle, by building up the theory using as much as possible algebraic models, mainly constructors, an algebra of programming can be achieved. Next, useful notation and functions on sequences and tuples are presented in order to formally define primitive operations on arrays.

2.2 ELEMENTARY FUNCTIONS

Often it is useful to refer to sets of the form

$$\{1, 2, 3, 4, \ldots, n\}$$

where $n \in \mathbb{N}$. In order to abbreviate the notation for such sets, the **index generator** ι is defined next.

DEFINITION 2.1 (ι)

For $n \in \mathbb{N}$, let $\iota n = \{1, 2, \ldots, n\}$.

DEFINITION 2.2 A **sequence** is any function with domain $i\,n$ or $\mathbb{N}$ where $n \in \mathbb{N}$. If t is a sequence then the kth component is denoted

$$t_k = t(k).$$

Two sequences s and t are equal on domain D, i.e., $s = t$, if and only if

$$s_k = t_k \text{ for all } k \in D.$$

Recall, for any set S and $n \in \mathbb{N}$,

$$S^n = \underbrace{S \times S \times \cdots \times S}_{n \text{ factors}}.$$

DEFINITION 2.3 ($n-$**tuple**)

$T \in S^n$ is a n-dimensional tuple for S or simply a n-tuple for S.

DEFINITION 2.4 For any set S, the family of all n-tuples with $n \geq k$ is

$$\mathcal{F}_S^k = \bigcup_{n=k}^{\infty} S^n.$$

If T is a n-tuple for $\mathbb{N}$ then T is a n-tuple for $\mathbb{Z}$. For convenience, adopt the symbols

$$\mathcal{N}^k = \mathcal{F}_{\mathbb{N}}^k \quad \text{and} \quad \mathcal{Z}^k = \mathcal{F}_{\mathbb{Z}}^k.$$

In particular, $\mathcal{N} = \mathcal{N}^1$ and $\mathcal{Z} = \mathcal{Z}^1$ are the families of all tuples for $\mathbb{N}$ and $\mathbb{Z}$, respectively. If $T \in S^n$ and $n > 1$ then T is a n-tuple for S and for $k \in \iota n$ there exists

$$t_k \in S$$

such that

$$T = (t_1, \ldots, t_n).$$

To abbreviate a n-tuple, an additional representation for a n-tuple is given next.

DEFINITION 2.5 If an object T admits the form

$$T = (t : n)$$

then necessarily $n \in \mathbb{N}$, t is a sequence with domain ιn and $(t_1, \ldots, t_n) \in S^n$ for some set S.

It is useful to find techniques to avoid handling indices excessively. Vectors and matrices utilize positions instead of indices.

EXAMPLE 2.1 Consider multiplying a 3×1 matrix and a 1×3 matrix:

$$\begin{bmatrix} 5 \\ 7 \\ 11 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 5 & 10 & 15 \\ 7 & 14 & 21 \\ 11 & 22 & 33 \end{bmatrix}.$$

Large multidimensional arrays cannot be displayed in such a way. Instead, a new function is employed to produce the **index set**, which is the set of all indices determined by a given n-tuple for $\mathbb{N}$. Given such an n-tuple, the indices are implicitly understood.

DEFINITION 2.6 $(\mathbb{P})$

For $(t : n) \in \mathbb{N}^n$, the function $\mathbb{P}$ with domain $\mathcal{N}$ is given by

$$\mathbb{P}(t : n) = (\iota\, t_1) \times \cdots \times (\iota\, t_n),$$

which is the **sequential product** of $(t : n)$.

EXAMPLE 2.2 The smallest nonempty index set is $\mathbb{P}(1) = \{1\}$.

EXAMPLE 2.3 A "2D" index set is

$$\mathbb{P}(2,3) = \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3)\}.$$

To further simplify the specification of n-tuples, the tau shift function is introduced. It turns out that primitive operations on multidimensional arrays can be defined in terms of this function.

DEFINITION 2.7 (τ)

The **tau shift**

$$\tau : \mathbb{N} \times \mathbb{Z} \times \mathcal{Z} \to \mathcal{Z}$$

obeys

$$\tau_k(m)(t : n) = \begin{cases} (t_1, \ldots, t_{k-1}, t_k + m - 1, t_{k+1}, \ldots, t_n) & \text{if } k \in \iota\, n \\ (t : n) & \text{otherwise} \end{cases}.$$

EXAMPLE 2.4 Suppose

$$T = (3,1,2,3,2,1,1,2,3,3).$$

To change the sixth component to three (3), calculate

$$\tau_6(3)T = (3,1,2,3,2,3,1,2,3,3).$$

Occasionally, it is convenient to drop a subscript which is one, e.g., $\tau = \tau_1$. Note that the tau shift function is also defined on $\mathbb{N}$.

EXAMPLE 2.5 Shift from position one to three: $\tau(3)(1) = 3$.

EXAMPLE 2.6 Shift the second component: $\tau_2(5)(1,1) = (1,5)$.

EXAMPLE 2.7 Notice $\tau_k(t_k)(t_1,\ldots,t_{k-1},1,t_{k+1},\ldots,t_n) = (t:n)$.

A condition may be attached (after a semicolon) which places additional requirements on the sequence. A special case is defined next.

DEFINITION 2.8 A **conditional** n-tuple has either the form

$$(t:n;t_k = m) \qquad \text{or} \qquad (T;t_k = m)$$

and denotes the n-tuple

$$\begin{aligned}(t:n;t_k = m) &= \tau_k(m+1-t_k)(t:n)\\ &= (t_1,\ldots,t_{k-1},m,t_{k+1},\ldots,t_n),\end{aligned}$$

provided $T = (t:n)$ is a n-tuple for $\mathbb{Z}$ and $m \in \mathbb{Z}$.

EXAMPLE 2.8 Observe

$$\begin{aligned}(i:n;i_k = 1) &= \tau_k(1+1-i_k)(i:n)\\ &= (i_1,\ldots,i_{k-1},i_k+1-i_k,i_{k+1},\ldots,i_n)\\ &= (i_1,\ldots,i_{k-1},1,i_{k+1},\ldots,i_n).\end{aligned}$$

Another tuple function, which simplifies notation, inserts an element into a given n-tuple.

DEFINITION 2.9 ($\sqsupset$)

The function **move right**

$$\sqsupset: \mathbb{N} \times S \times \mathcal{F}_S \to \mathcal{F}_S^2$$

inserts a given element of S into a n-tuple for S:

$$\sqsupset_k\, s\ (t:n) = \begin{cases}(t_1,\ldots,t_{k-1},s,t_k,t_{k+1},\ldots,t_n) & \text{if } k \in \iota\, n\\ (t_1,\ldots,t_n,s) & \text{otherwise}\end{cases}.$$

EXAMPLE 2.9 Insert five: $\sqsupset\, 5\ 3 = (5,3)$.

Another useful tuple function deletes a component of a given n-tuple.

DEFINITION 2.10 (δ)

The function **delete**

$$\delta\colon \mathbb{N} \times \mathcal{F}_S^2 \to \mathcal{F}_S$$

deletes the specified component of a given n-tuple as follows

$$\delta_k(t:n) = \begin{cases} (t_1, \ldots, t_{k-1}, t_{k+1}, \ldots, t_n) & \text{if } k \in \iota\, n \\ (t:n) & \text{otherwise} \end{cases}.$$

EXAMPLE 2.10 Delete the first component: $\delta(5,3) = 3$.

EXAMPLE 2.11 Delete the second component: $\delta_2(5,3) = 5$.

To abbreviate a tuple whose components are all the same constant, a **constant tuple** may be used, which is defined next.

DEFINITION 2.11 (C)

For $k \in \mathbb{Z}$ and $n \in \mathbb{N}$ let $C_k^n = (\underbrace{k, k, \ldots, k}_{n \text{ occurrences}}) \in \mathbb{Z}^n$.

This notation is natural since $\mathbb{Z}$ is a totally ordered set, the subscript k gives the kth index in this set, and the exponent gives the number of factors. An essential property of integers is that they are totally ordered. Next, total orders of n-tuples for $\mathbb{Z}$ are defined in the usual way, which should help to familiarize the notation. In addition, it is illustrative as it is possible to define operations that depend on all of the indices without having to explicitly write all of them. In fact, it is never necessary to identify more than a couple indices.

For $n = 1$ use the familiar orders

$$<_1 = <, \quad \leq_1 = \leq, \quad >_1 = >, \quad \geq_1 = \geq.$$

Recall for $i, j \in \mathbb{Z}$ $i < j$, $i = j$ or $i > j$. Consider two n-tuples for $\mathbb{Z}$, say $I = (i:n)$ and $J = (j:n)$ with $n > 1$. Define the order $<_1$ by

$$I <_1 J$$

if and only if there exists $p \in \iota\, n$ such that $i_q = j_q$ for all (if any) $q < p$ and $i_p < j_p$. Also,

$$I \leq_1 J \quad \text{if} \quad I <_1 J \text{ or } I = J$$

(where as usual $I = J$ if and only if $i_k = j_k$ for all k with $1 \leq k \leq n$). Symmetrically,

$$I >_1 J \quad \text{if} \quad J <_1 I \text{and} I \geq_1 J \quad \text{if} \quad J \leq_1 I.$$

All of the orders $<_1$, $\leq_1$, $>_1$ and $\geq_1$ are now completely defined.

Next, the remaining orders $<_k$, $>_k$, $\leq_k$, and $\geq_k$ for $k = 2, 3, 4, \ldots, n$ are defined. The order $<_k$ obeys

$$I <_k J \quad \text{if} \quad i_k < j_k \quad \text{or} \quad i_k = j_k \text{ and } \langle_k I <_1 \langle_k J.$$

Analogously as for $<_1$,

$$I \leq_k J \quad \text{if} \quad I <_k J \quad \text{or} \quad I = J, I >_k J \quad \text{if} \quad J <_k I \text{and} I \geq_k J \quad \text{if} J \leq_k I.$$

The order $<_1$ is called the lexicographical order or **left order** and $<_n$ the **right order** with respect to n-tuples for $\mathbb{Z}$.

2.3 REPRESENTATION

Consider the notion of array type and possible representations of an array. The simplest form is preferred. First, look at the system of axes. A potential system of axes for arrays with two or three dimensions corresponds in a natural way to the familiar Cartesian coordinate system (see Figure 3.3). Different representations for the"coordinates" or locations may be used.

EXAMPLE 2.12 Consider a matrix, say $A = [e]$, which has a single element. Suppose for some particular index, say i, $A[i] = e$. Similarly, another matrix B has the same element e but uses a different index, say j with $j \neq i$, and suppose $B[j] = e$. It is possible to define arrays so that these matrices are equal ($A = B$).

If different representations were allowed, then it would be necessary to check that all operations are well-defined for different representations. This necessity can be circumvented by choosing not to allow different labeling schemes along axes. In the preceding example, if it is required that $i = j = 1$, then it seems that at least one aspect of the form of an array is unambiguous by construction.

In general, an array has forms that cannot simply be determined from inspection of the system of axes and the elements. In particular, an array may have field directions. Such field directions are not captured by the notions of shape and content alone.

EXAMPLE 2.13 Consider a database in which each row represents homogeneous data so that it is possible to perform some operation on all of the elements in the rows, such as addition of all the elements in each row.

It is possible to specify the elements using different type specifications. Nevertheless, in order to treat elements collectively as a "row" for which operations may be defined, they must be characterized accordingly. Consider a row of elements. Suppose the types are specified using more than one type specifier, e.g., an element may be specified as an integer and another as a rational number. In other words, some sort of "union type" is employed for elements in a row. Mainly, it is difficult to discuss operations and the resulting type without specifying the type over which the operations are performed. Yet, even when the operation is well-defined, the result may be meaningless if the values are not related in any way besides having a common type.

An interpretation of a two-dimensional array is a series of rows with the same shape and within any row all elements have a *prescribed type*. Higher-dimensional arrays are a series of lower-dimensional arrays preserving shapes and *types*. In other words, multidimensional arrays may be created from lower-dimensional arrays by stacking along a new dimension. For instance, suppose there are two-dimensional tables of data, one for each business day of the week. These tables may be stacked to form a three-dimensional array. Similarly, weekly data in three-dimensional arrays for all weeks in a month may be stacked into a four-dimensional array.

A tacit assumption is that an array *may* have a field direction. This means that there may be a direction such that along any axis with this direction elements have some specified type so that it is possible to apply some operation axially. Still, it is possible an array model does not have a field direction. For instance, an array that has more than one element, and for which any pair of elements have distinct and incompatible types. It seems the simplest form of an array model is encapsulated in the description content plus shape. The definition of an array model should enclose no more, no less, than this simplest form.

2.4 IMPLEMENTATION

The intention is to focus on essential details and properties of computations and to ignore nonessential implementation details. An advantage of taking this approach is that programs are highly portable. A theory should not depend on any specific hardware or software implementation. For instance, an array should not be assumed to be stored in a contiguous memory block, even though this assumption might be realistic in

many cases. An array may be regarded as an immutable object that cannot be changed but which can be operated on to produce other objects.

A satisfactory implementation depends on many factors, including the representation of the objects, the common operations that are performed, and the hardware configuration. In any implementation, some choices will have to be made. It is important for optimization to take into account not only the representation of the objects but also the primitive operations that are commonly employed.

It is widely accepted that frequently used operations, like indexing, should be efficiently implemented. In order to implement primitive operations efficiently, it seems that it is necessary to have knowledge of the hardware/software configuration. It is unlikely that all costs can be reduced in some algebraic way. A theory is not intended to replace engineers or hardware designers. On the other hand, *hardware designers continue to create the need for an algebraic theory.*

While an algebraic theory may not provide a panacea, it seems reasonable to have some expectation that if a program is comprised of primitives which are efficiently implemented, then the program will likely inherit some measure of efficiency. Also if the number of primitives is kept small as proposed, work to achieve an efficient implementation will be naturally kept to a minimum. In addition, it is well-known that good theoretical models can yield efficient programs. An algebraic theory advances such models by matching the data type with common operations using algebraic structures.

Ultimately, a high-level program must be translated into machine code. In principle, it is possible to translate into another high-level language, such as a functional, logic, object-oriented or procedural-oriented language. Ideally, a series of transformations would be developed to accomplish the translations. At the topmost level, every primitive operation might be replaced by "skeletons" or models which are suitable for a particular class of computer architectures. A set of skeletons would be developed for each such class. These skeletons could then be refined and mapped to various high-level languages. Finally, compilers could be developed to translate directly from the given programming language to some particular machine code. When a new computer is introduced, it may be possible to use existing skeletons and develop a compiler by building on other work. It is reasonable to expect that many basic operations are available since there exist popular libraries such as `BLAS` or `NAG`.

Although memory has remained inherently sequential, different locations can be accessed in constant time. A primitive indexing operation on arrays is thus feasible. Elements of an array may be stored in some order as has been traditionally done. Tuples could be implemented by storing the number of items followed sequentially by the

addresses of the items in the order they are listed in the tuple.

When the set of locations is small, an array could be implemented via some built-in or user-defined function, such as an efficient case statement, matching locations with elements. A possible implementation would store a set of pairs where the first component is the location and the second component is the value of the data for that location; however, it should not be necessary to search for an item as would be appropriate for sets or general functions. It should be possible to specify an array via some sequence in some specified order.

2.5 SUMMARY OF ELEMENTARY NOTATION

The following notation was introduced:

$\iota\, n$	index generator produces $\{1, 2, \ldots, n\}$
$(t : n)$	a n-tuple $(t_1, \ldots, t_n) \in S^n$ for some S
C_k^n	a constant tuple $(k, \ldots, k) \in \mathbb{Z}^n$
$\tau_k(z)$	adds $z - 1$ to the kth component
$\sqsupset_k(z)$	inserts z at the kth position of a n-tuple
$\langle\!\rangle_k$	deletes the kth component of a given n-tuple
$\mathbb{P}(T)$	produces the index set
$\mathcal{F}_S^k$	family of all n-tuples for S with $n \geq k$
$\rightarrowtail a$	forms a pair with a as the first component
$a \leftarrowtail$	forms a pair with a as the second component
@	applies a given function to an argument: @ $f\, x = f(x)$
(f, g)	$(f, g)(x, y) = (f(x), g(y))$
$<_k, >_k, \leq_k, \geq_k$	total orders placed on n-tuples for $\mathbb{Z}$

CHAPTER 3

A FOUNDATION FOR ARRAYS

This chapter introduces formal notions of arrays. A primitive model, concise notation and descriptive terminology are presented. All arrays possess the same type. Useful concepts for collections of arrays and convenient symbols are introduced. A set of primitive operations as in abstract data types is specified which includes new functions that account for common program structures. All features and properties of arrays are readily available via suitable and well-defined access functions.

3.1 ARRAY MODEL

This section covers the following concepts: n-array, empty array, k-axis, and k-order. The type of an array is stated. A tool is added to translate from lists to arrays. A few ways to specify arrays for parameters and arguments to functions are discussed. Terminology is developed for subarrays and special arrays. Finally, useful notation for sets of arrays is introduced.

3.1.1 ARRAY TYPE

A primitive model of an array is presented next. The term mathematical model describes a functional dependence. An array is a function that depends on a set of locations. Recall a sequence is a function with domain $\mathtt{i}\ \mathtt{n}$ or $\mathbb{N}$. The term **instance** refers to an individual of a class.

DEFINITION 3.1 (**n-array**)

A n-array is an instance of a mathematical model

$$[\mathtt{a} \mid \mathtt{T}]$$

consisting of a n-tuple $T \in \mathbb{N}^n$, and a function $a : \mathbb{P}(T) \to Y$. If $T = (t : n)$ then

$$[a : t_1, \ldots, t_n] = [a \mid T].$$

The notation closely resembles set notation except the delimiters { } are replaced by brackets [ ]. This choice is reasonable as like sets, arrays are collections. The "condition" is that which specifies the domain. For arrays, all that is required is a n-tuple for $\mathbb{N}$ as the domain is completely determined via $\mathbb{P}$. Two forms are employed so that it is possible to distinguish between a tuple and a natural number and to avoid excessively writing delimiters, e.g., although the parentheses in $[a : (t_1, \ldots, t_n)]$ are relevant, they are superfluous delimiters. This notation seems natural, partly because of the similarity to the way arrays are usually indexed.

The fact that it is possible to discuss a n-array for any $n \in \mathbb{N}$ means laws and properties may be stated in full generality. By definition, if $[a \mid T]$ is a n-array then T is a n-tuple for $\mathbb{N}$. Concerning arrays, it is not required to append "for $\mathbb{N}$" after "n-tuple" since it is required that the n-tuple for an array be a n-tuple for $\mathbb{N}$ whenever an object is declared to be a n-array.

An **array model** refers to the model of a n-array. Note in the definition of n-array, ordinary mathematical notions of functions and sets are employed. The definition of a n-array is closely related to Hudak's "functional array" (1986). A 1-array may also be called a list whenever doing so causes no confusion.

Let $A = [a \mid T]$ be a n-array. The function a is called the **array function** of A. For each $(i : n) \in \mathbb{P}(T)$, the value $a(i : n)$ is called the **element** or **entry** of A at **location** $(i : n)$ and each i_k is an **index**. Elements of the set $\mathbb{P}(T)$ of all locations for A are also called **indices**, **nodes** or **points** of A. Two elements, nodes or points are **adjacent** or **adjacent for** k if the corresponding locations $(i : n)$ and $(j : n)$ satisfy $i_p = j_p$ for all p with $p \neq k$ and $|i_k - j_k| = 1$. Suppose $T = (t : n)$ and consider a location $(i : n)$. If for all $k \in \iota\, n$,

$$\text{either } i_k = 1 \text{ or } i_k = t_k$$

then the location $(i : n)$ is **exterior** and $a(i : n)$ is an exterior element. If for some $k \in \iota\, n$,

$$1 < i_k < t_k$$

then the location $(i : n)$ is **interior** and $a(i : n)$ is an interior element.

Two n-arrays $[f \mid P]$ and $[g \mid Q]$ are **equal**, i.e., $[f \mid P] = [g \mid Q]$, if and only if $P = Q$ and $f(t) = g(t)$ for all $t \in \mathbb{P}(P) = \mathbb{P}(Q)$. There are three different equalities here.

Only the first equality $[f|P] = [g|Q]$ was defined. The second equality $P = Q$ is the equality for tuples. The third equality $e(t) = f(t)$ is not necessarily defined.

An array model is versatile with numerous applications. The model conforms well to typical representations of arrays. A model should simplify the universe of objects. Yet, the model should not diminish the objects in any way.

EXAMPLE 3.1 A n-array allows for repetition of values. It is always possible to transform a n-array with duplicates into a n-array with unique elements. As an array function may be chosen arbitrarily, pair elements with locations to obtain distinct elements. For more details, see the Appendix.

EXAMPLE 3.2 Every matrix corresponds in a natural way to a 2-array, i.e.,

$$\begin{bmatrix} a(1,1) & a(1,2) & \cdots & a(1,n) \\ a(2,1) & a(2,2) & \cdots & a(2,n) \\ \vdots & \vdots & \ddots & \vdots \\ a(m,1) & a(m,2) & \cdots & a(m,n) \end{bmatrix} \sim [a : m, n].$$

EXAMPLE 3.3 A $n \times n$ matrix of the form

$$\begin{bmatrix} 1 & 2 & \cdots & n \\ n+1 & n+2 & \cdots & 2n \\ \vdots & \ddots & & \vdots \\ (n-1)n+1 & (n-1)n+2 & \cdots & n^2 \end{bmatrix} \quad \text{such as} \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

is concisely expressed as the 2-array $N_n = [a : n, n]$ with $a(i,j) = n(i-1)+j$, $i, j \in \iota n$. Call any 2-array that is equal to N_n for some n a **squared 2-array**.

EXAMPLE 3.4 Every finite sequence of elements $x_1, x_2, \ldots, x_n$ corresponds to the 1-array $[x : n]$.

EXAMPLE 3.5 **Hypercubes** may be regarded as models of higher-dimensional arrays formed from cubes. Define a simplified version recursively as follows. Omit the specification of the values at all locations. A line segment consisting of two points, a square (with four points), and a cube (with eight points) forms a one-dimensional, two-dimensional, and three-dimensional hypercube, respectively. A $(n+1)$-dimensional hypercube consists of two copies of a n-dimensional hypercube. Using Example 3.4, represent a line segment given two points x_1 and x_2 by the 1-array $[x : 2]$. Represent a

square by the 2-array $[x^2 : 2, 2]$ as shown in Figure 3.1(a). Continuing in this way, a cube is represented via the 3-array $[x^3 : 2, 2, 2]$. If a n-dimensional hypercube is represented as the n-array $[x^n | C_2^n]$ then a $(n+1)$-dimensional hypercube is represented by the $(n+1)$-array $[x^{n+1} | C_2^{n+1}]$ and a location $(i : n+1; i_{n+1} = k)$ corresponds to the kth copy.

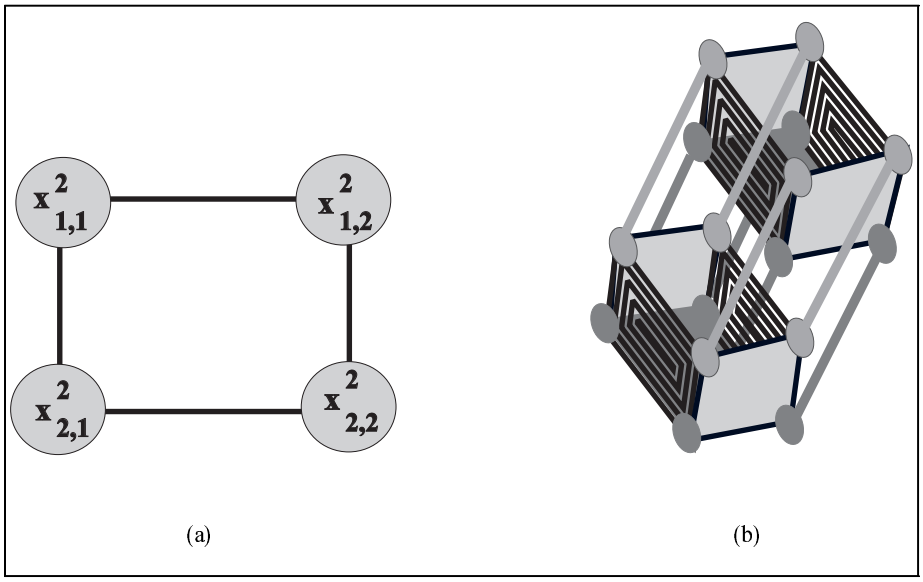


Figure 3.1: Figures 3.1(a) and 3.1(b) show a two-dimensional and a four-dimensional hypercube, respectively.

EXAMPLE 3.6 Generalizing Example 3.5, build up simple multidimensional arrays recursively as follows. Focus only in the shape of an array and disregard the content (assign the value "node" to all elements). A one-dimensional array is finitely many copies of a node, say t_1. Using Example 3.4, represent it via the 1-array $[a : t_1]$. A two-dimensional array is finitely many copies, say t_2, of a one-dimensional array and may be represented by the 2-array $[a^2 : t_1, t_2]$ with a location (i_1, k) corresponding to the kth copy. A three-dimensional array is finitely many copies, say t_3, of a two-dimensional array and may be represented via the 3-array $[a^3 : t_1, t_2, t_3]$ with a location (i_1, i_2, k) corresponding to the kth copy. The set of points

$$\{(i_1, i_2, k) \mid k \text{ is fixed}\}$$

may be described as a "hyperplane." Continuing in this way, an $(n+1)$-dimensional array, which is the $(n+1)$-array $[a^{n+1} | (t : n+1)]$, is t_{n+1} copies of a n-dimensional array $[a^n | (t : n)]$ with a location $(i : n+1; i_{n+1} = k)$ corresponding to the kth copy.

EXAMPLE 3.7 Consider any table of the form

x_1	x_2	x_3	x_4
a(1,1)	a(1,2)	a(1,3)	a(1,4)
a(2,1)	a(2,2)	a(2,3)	a(2,4)
a(3,1)	a(3,2)	a(3,3)	a(3,4)

where x_i is a table header describing the a(i,j)'s for each i, a(i,j) possibly has a different type for each i, and a(i,j) depends on a(i,1) and x_j for $j > 1$. Such a table may be represented using the 1-array [x : 4] and the 2-array [a : 3, 4].

An "empty array" presumptively has no elements, axes or locations. Although empty arrays may not make much sense as arrays per se, they are added to the theory because they are useful to algebraically describe operations on arrays. An empty array may be regarded as extension of an array model in the same way $\infty \notin \mathbb{R}$ is an extension of the real numbers $\mathbb{R}$.

DEFINITION 3.2 (**empty array**)

An empty array is an instance of the prototype

□.

This symbol is chosen because it is "empty" of an array function and a n-tuple. Alternative representations (in case □ is not available) are [ ] and ∅ as these symbols resemble the corresponding symbols for the empty set, namely {} and ∅. Necessarily, any two instances of an empty array, say E and F, are equal: E = F. For definiteness, an empty array does not have any elements, locations or points. An empty array corresponds to a set of nodes, namely the empty set. Whenever needed, operations defined on n-arrays are also defined for the empty array.

DEFINITION 3.3 (**array**)

An object has type array if the object is either a n-array for some $n \in \mathbb{N}$ or an empty array.

A n-array A = [a | T] never equals an empty array. A n-array is said to be a **nonempty array** if it is not equal to the empty array; whence, a nonempty array is a n-array. It is permissible to define an operation on n-arrays only for some fixed $n \in \mathbb{N}$. In particular, such operation need not be defined either for m-arrays with $m \neq n$ or an empty array. Such definitions are common in mathematics, e.g., $\frac{1}{x}$ is defined only for $x \in \mathbb{R} - \{0\}$).

EXAMPLE 3.8 An object $[a:2,3,5,7,11]$ has type array, where $a(i:5)=21$ for all valid locations $(i:5)$.

EXAMPLE 3.9 An object $[a:1,3]$ has type array, where $a(1,1)=\square$, $a(1,2)=[b:1,1]$, $b(1,1)=\square$, $a(1,3)=[c:1,1,1]$ and $c(1,1,1)=\square$.

A n-array may be characterized as an **integer array**, **nonnegative** or **positive** if all the elements are integers, nonnegative or positive, respectively. Similarly, any type descriptor which characterizes all elements may be prefixed to the term array to describe an array with such elements. A 1-array $A=[a:m]$ is said to be **nondecreasing** if a is nondecreasing. A is **strictly increasing** if a is strictly increasing. Similarly, any 1-array may be characterized by properties of its array function.

3.1.2 ORDERING OF ELEMENTS

There is a familiar **lexicographical order** (in alphabetical order), or **row order**, of elements of an array inherited from the total orders defined on n-tuples for $\mathbb{Z}$ in Chapter 1. Traditionally, matrices have been implemented by storing the contents sequentially listing items from a row after the items from the preceding row. This tradition should not be surprising since a popular model for memory has remained a sequential one.

DEFINITION 3.4 For any n-array $A=[a\,|\,(t:n)]$ and for $k\in\iota\,n$, define the **k-order** of elements of A as the list

$$\langle {}_k a\,(\mathbb{C}^n),\ldots,a(i:n),\ldots,a(j:n),\ldots,a(t:n)\rangle,$$

where $(i:n)<_k(j:n)$.

EXAMPLE 3.10 The 1-order of a squared 2-array is the list

$$\langle 1,2,3,\ldots,n^2\rangle$$

where n is the number of rows or columns (see Example 3.3).

EXAMPLE 3.11 Consider any $m\times n$ matrix. The 1-order or row order of elements is the usual listing of elements by rows. The 2-order or **column order** yields a listing of elements by columns.

EXAMPLE 3.12 Consider any $m\times n\times p$ matrix and 3-order. For instance, the 3-array

$[a:2,2,2]$ depicted in Figure 3.2 with

$$a(i,j,k) = 2(i-1)+j+4(k-1)$$

yields the 3-order $\langle_3 1,2,3,4,5,6,7,8\rangle$.

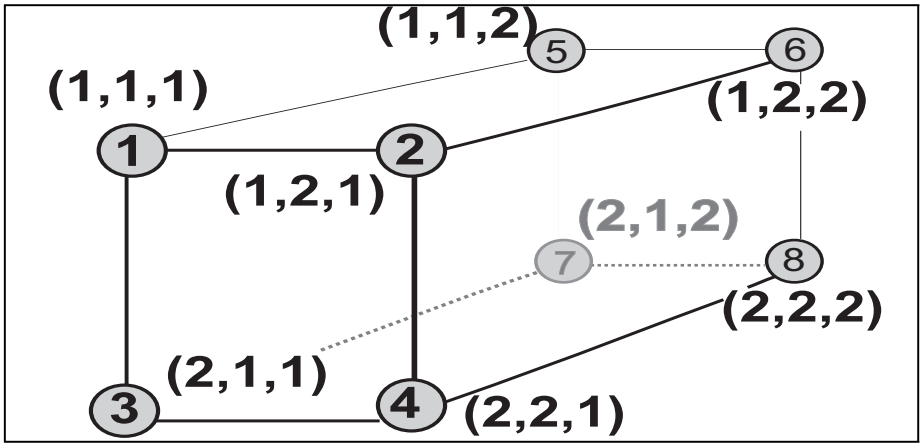


Figure 3.2: The elements of $[a:2,2,2]$ with labelled locations are shown.

3.1.3 SPECIFICATION OF ARRAYS

An array function of a 1-array is a sequence. It is useful to be able to specify short lists without referring to the index set since there is only one axis and the elements are given in order. It is convenient to introduce a method to specify 1-arrays succinctly. Recall a n-tuple can have arbitrary elements for components whereas a n-tuple for $\mathbb{N}$ has components that are all natural numbers.

DEFINITION 3.5 Let A_k be any nonempty set for $k \in \iota n$. The function — with domain

$$A_1 \times A_2 \times \cdots \times A_n$$

is given by

$$\overline{a_1,a_2,\ldots,a_n} \longmapsto [a:n].$$

Consider two 1-arrays $[f:p]$ and $[g:q]$ with $[f:p] = [g:q]$. Evidently $p = q$ and $f_k = g_k$ for all k. This establishes that the **list function** — is well-defined. Incidentally, vinculum in mathematics groups terms by placing a bar over the compound.

EXAMPLE 3.13 Defining $a_k = k$, $\overline{1,\ldots,n} = [a:n]$.

EXAMPLE 3.14 Put $A = \overline{0,3}$. Then $A = [a:2]$ with $a_1 = 0$ and $a_2 = 3$.

Lists are popular. It is convenient to be able to translate quickly from lists to arrays. Since a sequence is listed in order, it is natural to infer the indices. Any small 2-array may be specified by simply presenting a usual matrix, preferably using brackets around it. All $\mathtt{m} \times 1$ matrices should be distinguished as 2-arrays as this information is not available from such matrices, i.e., a 2-array that is presented as a vector appears as a 1-array.

Any small n-array may be presented by specifying the necessary parts using array notation, appending the list of elements in some specified order, preferably enclosed in angled brackets. The following form is proposed

$$[\mathtt{a} : \mathtt{t}_1, \ldots, \mathtt{t}_n \langle_k \mathtt{e}_1, \ldots, \mathtt{e}_s \rangle]$$

where the $\mathtt{e}_k$'s are the elements listed in k-order (and s denotes the total number of elements). It is also possible to use the forms

$$[\mathtt{a} \mid (\mathtt{t} : \mathtt{n}) \langle_k \mathtt{e}_1, \ldots, \mathtt{e}_s \rangle] \text{ or } [\mathtt{a} \mid \mathtt{T} \langle_k \mathtt{e}_1, \ldots, \mathtt{e}_s \rangle]$$

provided that the sequence t or the n-tuple T is defined elsewhere.

EXAMPLE 3.15 $[\mathtt{a} : 2,2,2 \langle_3 1,2,3,4,5,6,7,8 \rangle]$ specifies the 3-array in Example 3.12.

EXAMPLE 3.16 $[\mathtt{a} : 2,2,2 \langle 1,5,2,6,3,7,4,8 \rangle]$ specifies the 3-array in Example 3.12.

The **iota array** is the 1-array

$$\urcorner_n = \overline{1, \ldots, n} = [\mathtt{id}_{\mathbb{N}} : n].$$

The infinite iota array $\urcorner_\infty$ is a 1-array with type array of the form $[\mathtt{id}_{\mathbb{N}} : n]$ where n is variable with type $\mathbb{N}$. This array would be treated as a constant array which is as large as needed. The variable n would be determined from other parameters. For instance if an operation is defined for a given parameter which is a 1-array of length n and an argument which is a n-array then if the parameter is $\urcorner_\infty$ the operation is defined for any given array that is nonempty.

Extend the constant tuple to arrays naturally:

$$\overline{\mathtt{c}^n_k} = [\mathtt{c} : n]$$

where $\mathtt{c}_i = k$ for all $i \in \iota n$. Call $\overline{\mathtt{c}^n_k}$ the "constant-k by n" array. The constant-k array $\overline{\mathtt{c}^\infty_k}$ could also be used as an infinite constant array.

EXAMPLE 3.17 $\ulcorner = [\mathrm{id}_{\mathbb{N}} : 1]$.

EXAMPLE 3.18 $\overline{\square^4} = \overline{1,1,1,1}$.

It is permitted to use arrays of the form

$$[f \mid \infty]$$

provided f is a function with domain $\mathbb{N}$ and $[f \mid \infty]$ is treated as the 1-array $[f : n]$ where n is a variable with type $\mathbb{N}$.

3.1.4 AXES

A n-array has **axes**. The system of axes together with the nodes comprise the **net**, **mesh** or **network** of a n-array. An axis is an imaginary **vector** with a **head** and a **tail**. This vector always has magnitude and direction. Two vectors are equal if and only if they have the same magnitude and direction.

The magnitude or length of an axis corresponds to the number of elements at locations in the mesh along parallel vectors. An axis may be moved from one place to another in the net provided its direction is not changed and its **head** and **tail** coincide with exterior locations in the mesh. In particular, given an arbitrary n-array $[a \mid (t : n)]$ and a natural number $k \in \iota\, n$, there is a k-axis with **magnitude** or **extent** t_k and **direction** k. Given any k-axis, its tail may be placed only at locations of the following form $(i : n; i_k = 1)$. Whenever the tail of the k-axis is placed at such a location, the head is necessarily at location $(i : n; i_k = t_k)$.

All axes with the same magnitude and direction are equal and correspond to a k-axis for some k. To identify a particular k-axis, it is necessary to specify the direction k and to identify a location along the k-axis such as the head of the axis. Axes are naturally ordered so that the k-axis is the kth axis. So the first axis is the 1-axis, the second axis is the 2-axis, and so forth.

Principal operations on arrays act along some direction determined by a k-axis for some fixed k. A **field direction** k is a direction k provided along any particular k-axis there exists a binary operator that may be placed between any two adjacent elements on that axis. Such operations are described by saying they are performed along rows or columns and so forth. The empty array has no axes.

3.1.5 SUBARRAYS

Some new terminology is helpful to properly discuss arrays. A practical notion is subarray, which corresponds to the restriction of an array function to a subset of the

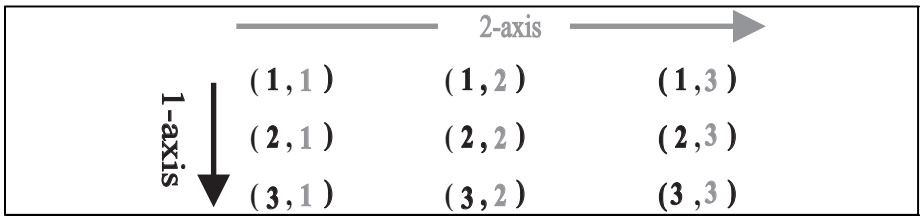


Figure 3.3: Locations along 1-axis and 2-axis, vary vertically and horizontally, respectively, i.e., rotate the usual coordinate system 90 degrees and identify the tuple (i,j) with the coordinate (i,j).

nodes.

DEFINITION 3.6 A n-array $A = [a \mid (s : n)]$ is a **subarray** of another n-array $B = [b \mid (t : n)]$, provided $s_i \leq t_i$ for all $i \in \iota\, n$, and there are strictly increasing functions $\lambda^k : \iota\, s_k \to \iota\, t_k$ such that $a(i : n) = b\left(\gamma^i : n\right)$ where $\gamma^i_k = \lambda^k_{i_k}$. An empty array $\square$ is a subarray of every n-array.

EXAMPLE 3.19 A submatrix is obtained by crossing out rows and columns. Upon removing the second row and column of the squared 2-array N_3 (see Example 3.3), the 2-array $A = [a : 2, 2] \sim \left[\begin{smallmatrix} 1 & 3 \\ 7 & 9 \end{smallmatrix}\right]$ is obtained, which is a subarray of N_3.

EXAMPLE 3.20 The squared 2-array N_1 is a subarray of the squared 2-array N_n for every $n \in \mathbb{N}$. Put $\lambda_1 = \lambda_2 = 1$ (see Example 3.3).

If A is a subarray of B, then say B **contains** or **includes** A. An array A is said to be a **strict subarray**, or a **proper subarray**, of an array B provided A is not an empty array and A is a subarray of B with $A \neq B$. It is also convenient to be able to say when an array is a contiguous subarray of another array.

DEFINITION 3.7 A n-array $A = [a \mid (s : n)]$ is a **contiguous subarray** of another n-array $B = [b \mid (t : n)]$ provided $s_i \leq t_i$ for all $i \in \iota\, n$ and there exist nonnegative shifts δ_i such that

$$a(i : n) = b\left(\gamma^{i,\delta} : n\right)$$

where $\gamma^{i,\delta} = +\circ(i, \delta)$, i.e., $\gamma^{i,\delta}(k) = i_k + \delta_k$.

EXAMPLE 3.21 A submatrix which is any row or column of a matrix is a contiguous subarray.

EXAMPLE 3.22 The squared 2-array N_1 is a contiguous subarray of the squared 2-array N_n for every $n \in \mathbb{N}$. (see Example 3.3)

Recall, arrays may have duplicates of values. Hence, two subarrays formed from different parts of an array may have common elements and may even be equal. It is useful to be able to say when subarrays overlap in an array that contains them.

DEFINITION 3.8 A collection

$$\left\{\left[\mathtt{a}^{\mathtt{k}} \mid \left(\mathtt{s}^{\mathtt{k}} : \mathtt{n}\right)\right]\right\}_{\mathtt{k} \in \iota\, \mathtt{m}}$$

of nonempty subarrays of a $\mathtt{n}$-array $\mathtt{B} = [\mathtt{b} \mid (\mathtt{t} : \mathtt{n})]$ is pairwise **disjoint** or **nonoverlapping** in $\mathtt{B}$ if there exist strictly increasing functions $\lambda^{\mathtt{k},\mathtt{p}} : \iota\, \mathtt{s}_{\mathtt{p}}^{\mathtt{k}} \rightarrow \iota\, \mathtt{t}_{\mathtt{p}}$ (which gives the $\mathtt{p}$th component) such that

$$\mathtt{a}^{\mathtt{k}}(\mathtt{i} : \mathtt{n}) = \mathtt{b}\left(\gamma^{\mathtt{k},\mathtt{i}} : \mathtt{n}\right)$$

and

$$\varnothing = \bigcap_{\mathtt{k}=1}^{\mathtt{m}} \left\{\left(\gamma^{\mathtt{k},\mathtt{i}} : \mathtt{n}\right) \mid (\mathtt{i} : \mathtt{n}) \in \mathbb{P}\left(\mathtt{s}^{\mathtt{k}} : \mathtt{n}\right)\right\}$$

where $\gamma_{\mathtt{p}}^{\mathtt{k},\mathtt{i}} = \lambda_{\mathtt{i}_{\mathtt{p}}}^{\mathtt{k},\mathtt{p}}$; otherwise, the collection is said to be **overlapping**.

EXAMPLE 3.23 The collection $\{\mathtt{B}\}$ is nonoverlapping for all $\mathtt{n}$-arrays $\mathtt{B}$.

EXAMPLE 3.24 The 2-arrays $\left\{\mathtt{B}^{\mathtt{k}}\right\}_{\mathtt{k} \in \mathtt{n}}$ are nonoverlapping in the squared 2-array $\mathtt{N}_{\mathtt{n}}$ (see Example 3.3) where $\mathtt{B}^{\mathtt{k}} = \left[\mathtt{b}^{\mathtt{k}} : 1, 1\right]$ with $\mathtt{b}^{\mathtt{k}}(1, 1) = \mathtt{k}$. Put

$$\lambda_1^{\mathtt{k},1} = \frac{\mathtt{k} - (\mathtt{k} \bmod \mathtt{n})}{\mathtt{n}} + 1$$

and

$$\lambda_1^{\mathtt{k},2} = \mathtt{k} \bmod \mathtt{n}\,.$$

where $\mathtt{k} \bmod \mathtt{n}$ is the remainder after dividing $\mathtt{k}$ by $\mathtt{n}$.

Informally, two subarrays overlap if and only if they both have an element which was taken from the same location in the larger array that contains them. Two disjoint arrays may or may not have some common elements and do have at least one common location. In fact, two disjoint arrays may be equal. A nonoverlapping collection has the *potential* to cover an array.

DEFINITION 3.9 Let $\mathtt{B} = [\mathtt{b} \mid (\mathtt{t} : \mathtt{n})]$ be any $\mathtt{n}$-array. Suppose $\mathtt{C}$ is any collection of $\mathtt{n}$-arrays. Then $\mathtt{C}$ forms a **covering** of $\mathtt{B}$ or **covers** $\mathtt{B}$, or is a **cover** for $\mathtt{B}$, if it is possible to form $\left\{\mathtt{A}^{\mathtt{k}}\right\}_{\mathtt{k} \in \iota\, \mathtt{m}}$ which is a nonoverlapping collection of contiguous subarrays in $\mathtt{B}$

such that each $A^k = [a^k \mid (s^k : n)]$ is a subarray of a unique $A \in C$ and

$$\sum_{k=1}^{m} \prod_{i=1}^{n} s_i^k = \prod_{i=1}^{n} t_i,$$

where $\prod$ denotes usual multiplication of operands. If B is an empty array, then any collection of n-arrays covers B.

Informally, a collection of arrays covers B if it is possible to overlap B completely by overlaying arrays from the collection onto the mesh of B. The main requirement is there must be enough arrays that contain contiguous subarrays to cover B.

EXAMPLE 3.25 For every n-array B, $\{B\}$ covers B.

EXAMPLE 3.26 Consider the 2-array $A = [a : 2, 2] \sim \left[\begin{smallmatrix} 4 & 3 \\ 2 & 1 \end{smallmatrix}\right]$ with $a(i,j) = 7 - 2i - j$. The collection $\{A\}$ does *not* cover the squared 2-array $N_2 \sim \left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$ (see Example 3.3). Basically, it is possible to cover only one of the elements in N_2, since every subarray from A which is also a contiguous subarray of N_2 contains only one element. It is permitted to form only one subarray from A as there are no other arrays in the collection. Setting $B^k = A$ for $k \in \iota\, 4$, $\{B^k\}_{k \in \iota\, 4}$ forms a covering of N_2.

EXAMPLE 3.27 The 2-arrays $\{B^k\}_{k \in \iota\, n^2}$ in Example 3.24 covers the squared 2-array N_n (see Example 3.3).

An array is comprised of a finite series of hyperplanes.

DEFINITION 3.10 Any n-array of the form $[a \mid (t : n)]$ with $n > 1$ and $t_k = 1$ is a **hyperplane**. To identify the index k with $t_k = 1$, write a hyperplane "for the k-axis."

Hyperplanes are ordered in the natural order.

EXAMPLE 3.28 Let $A = [a \mid (t : n)]$ be any n-array with $n > 1$ and $t_k \geq p \geq 1$. Set $B = [b \mid T]$ where

$$b(i_1, \ldots, i_{k-1}, 1, i_{k+1}, \ldots, i_n) = a(i_1, \ldots, i_{k-1}, p, i_{k+1}, \ldots, i_n),$$

and

$$T = (t : n; t_k = 1) = (t_1, \ldots, t_{k-1}, 1, t_{k+1}, \ldots, t_n).$$

Then B is the pth hyperplane of A for the k-axis. B is a contiguous subarray of A.

3.1.6 SETS OF ARRAYS

Some useful notation and terminology for sets of arrays is given next. For $\mathtt{T} \in \mathbb{N}^n$, define **array space for** $\mathtt{T}$ as the set

$$\mathbb{A}(\mathtt{T}) = \{[\mathtt{a} \mid \mathtt{T}] \text{ has type array}\}$$

which is the set of all arrays with shape $\mathtt{T}$. Hence $\mathtt{n}$-**array space** is the set

$$\mathbb{A}^n = \bigcup_{\mathtt{T} \in \mathbb{N}^n} \mathbb{A}(\mathtt{T})$$

which is the set of all $\mathtt{n}$-arrays.$\mathbb{A}^{n*} = \{\square\} \cup \mathbb{A}^n$ is the set that contains the empty array and all $\mathtt{n}$-arrays. Define the family

$$\mathcal{A}^k = \bigcup_{\mathtt{n}=\mathtt{k}}^{\infty} \mathbb{A}^n,$$

which is the family of all $\mathtt{n}$-arrays with $\mathtt{n} \geq \mathtt{k}$. Note $\mathcal{A}^1 = \mathcal{A}$ is the set of all nonempty arrays. Although it is not possible to define the set of all sets, it is possible to define the set of all arrays. The family of all arrays is $\mathcal{A}^* = \{\square\} \cup \mathcal{A}$.

For $\mathtt{T} \in \mathbb{N}^n$, define the family $\mathcal{C}_\mathtt{Y}(\mathtt{T})$ by announcing $\mathtt{a} \in \mathcal{C}_\mathtt{Y}(\mathtt{T})$ if $\mathtt{a} : \mathbb{P}(\mathtt{T}) \to \mathtt{Y}$ is any function. $\mathbb{A}_\mathtt{Y}(\mathtt{T})$ is **array space** for $\mathtt{T}$ in $\mathtt{Y}$. So $\mathtt{n}$-array space in $\mathtt{Y}$ is

$$\mathbb{A}_\mathtt{Y}^n = \bigcup_{\mathtt{T} \in \mathbb{N}^n} \mathbb{A}_\mathtt{Y}(\mathtt{T}),$$

which is the set of all $\mathtt{n}$-arrays in $\mathtt{Y}$. The family

$$\mathcal{A}_\mathtt{Y}^k = \bigcup_{\mathtt{n}=\mathtt{k}}^{\infty} \mathbb{A}_\mathtt{Y}^n,$$

is the family of all $\mathtt{n}$-arrays in $\mathtt{Y}$ with $\mathtt{n} \geq \mathtt{k}$. In addition, $\mathcal{A}_\mathtt{Y}$ is the family of all nonempty arrays in $\mathtt{Y}$. In addition,

$$\mathcal{A}_\mathtt{Y}^* = \{\square\} \cup \mathcal{A}_\mathtt{Y}$$

is the family of all arrays in $\mathtt{Y}$. It is convenient to admit a symbol for the set of all hyperplanes

$$\mathbb{H} = \left\{[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] \in \mathcal{A}^2 \mid \mathtt{t}_\mathtt{k} = 1 \text{ for some } \mathtt{k}\right\}.$$

The set of all hyperplanes in $\mathtt{Y}$ is

$$\mathbb{H}_\mathtt{Y} = \mathbb{H} \cap \mathcal{A}_\mathtt{Y}.$$

3.2 CONSTRUCTORS

A new data type may be introduced by listing **generators** that are actual values of the data type and also specifying **generating relations** or simply relations, which are functions that generate all other values. An empty array $\square$ is admitted as a generator, which is the only generator employed. In the remainder of this section, all of the generating relations are defined. Together the generators and relations comprise the constructors.

Consider an array that contains a single element. A function to construct such arrays is defined next.

DEFINITION 3.11 (CONSTRUCTOR)

For any object $\mathtt{x}$ of arbitrary type α and for arbitrary $\mathtt{n} \in \mathbb{N}$, the function

$$(\cdot)_{\mathtt{n}} : \alpha \to \mathbb{A}^{\mathtt{n}}$$

obeys

$$(\cdot)_{\mathtt{n}}(\mathtt{x}) = \left[\mathtt{a} \,|\, \complement^{\mathtt{n}}\right],$$

where $\mathtt{a}$ is the array function

$$\mathtt{a} : \complement^{\mathtt{n}} \to \{\mathtt{x}\}$$

given by $\mathtt{a}\left(\complement^{\mathtt{n}}\right) = \mathtt{x}$.

An array of the form $\left[\mathtt{a} \,|\, \complement^{\mathtt{n}}\right]$ is called **scalar**, which is suggestive as unlike a vector there is no special direction. The constructor $(\cdot)_{\mathtt{n}}$ is called a **scalar function**. Notice an infinite number of scalar functions have been defined. A scalar function $(\cdot)_{\mathtt{n}}$ always produces a $\mathtt{n}$-array. An array is **nonscalar** if it is not scalar. If $\mathtt{n} = 1$, a 1-array that is scalar is also called a **singleton**. For $\mathtt{n} > 1$, a $\mathtt{n}$-array that is scalar is also a hyperplane, even though it has just one point. The term hyperplane is used only to suggest more than one dimension with one axis of minimal extent. A 1-array is never a hyperplane.

EXAMPLE 3.29 $[\mathtt{a} : 1, 1, 1, 1]$ is scalar for all possible functions $\mathtt{a}$ defined on $\complement^4 = (1, 1, 1, 1)$.

EXAMPLE 3.30 $[+1 : 1] = \overline{\complement_2}$ denotes the singleton that has the element 2.

EXAMPLE 3.31 $[\mathtt{a} : 1] = (\cdot)(\mathtt{a}_1) = \overline{\mathtt{a}_1}$.

Appending a new axis converts a $\mathtt{n}$-array into a $(\mathtt{n}+1)$-array. The content of the array is unchanged. A "1" is simply appended to every location. This transformation

is a structure preserving operation, i.e., the elements are not rearranged in any way and the orientation of the mesh is unchanged.

EXAMPLE 3.32 A squared 2-array $\mathtt{N_n} = [\mathtt{a} : \mathtt{n}, \mathtt{n}]$ (see Example 3.3) may be transformed into the 3-array $[\mathtt{b} : \mathtt{n}, \mathtt{n}, 1]$ with $\mathtt{b}(\mathtt{i}, \mathtt{j}, 1) = \mathtt{a}(\mathtt{i}, \mathtt{j})$.

EXAMPLE 3.33 A 1-array may be mapped to a 2-array via a transformation of the "list" into a column matrix as shown in Figure 3.4(a).

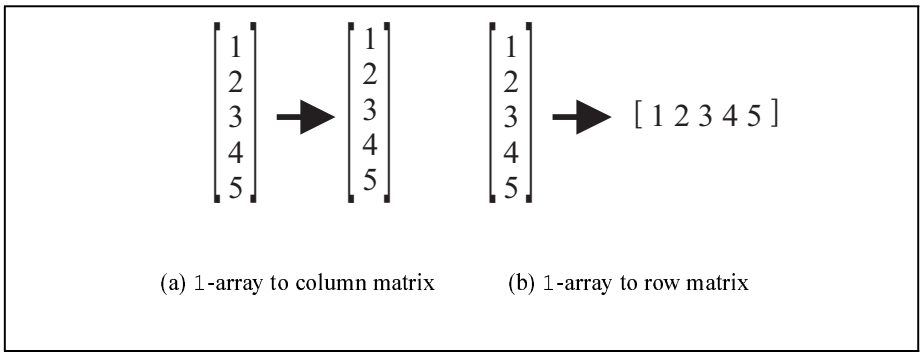


Figure 3.4: (a) A 1-array $\mathtt{A} = [\mathtt{a} : 5]$ where $\mathtt{a_i} = \mathtt{i}$ (which appears as a column yet has only one axis) is transformed into the 2-array $\mathtt{B} = [\mathtt{b} : 5, 1]$ where $\mathtt{b}(\mathtt{i}, 1) = \mathtt{i}$. (b) A is transformed into the 2-array $\mathtt{B} = [\mathtt{b} : 1, 5]$ with $\mathtt{b}(1, \mathtt{i}) = \mathtt{i}$.

It is always possible to insert an axis for any n-array. If a k-axis is inserted, then whatever was the length of the k-axis, its length is now 1, and what was the k-axis is now the $(\mathtt{k}+1)$-axis, what was the $(\mathtt{k}+1)$-axis is now the $(\mathtt{k}+2)$-axis, and so forth.

EXAMPLE 3.34 Consider transforming a 1-array to a 2-array by transposing the corresponding vector to a row matrix as depicted in Figure 3.4(b).

EXAMPLE 3.35 Consider transforming a squared 2-array $\mathtt{N_2}$ into a 3-array B by rotating the array so that the 1-axis becomes the 2-axis and then pushing the former 2-axis down so that it becomes the 3-axis and the newly added axis is the 1-axis. See Figure 3.5.

Adding an axis at the end results in a structure preserving transformation, while inserting an axis yields a rotation. It is not necessary to single out one hyperplane as being the last one and it should be possible to view *any* plane as being formed via a structure preserving operation that adds an axis to an array.

It should be possible to build up arrays in any direction without rotations. The

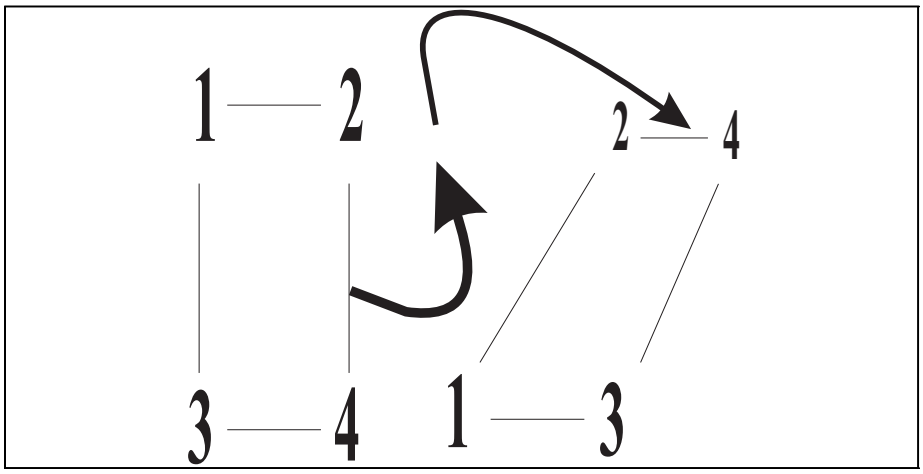


Figure 3.5: A squared 2-array $\mathtt{A} = [\mathtt{a} : 2, 2]$ where $\mathtt{a}(\mathtt{i}, \mathtt{j}) = 2(\mathtt{i}-1)+\mathtt{j}$ is transformed into a 3-array $\mathtt{B} = [\mathtt{b} : 1, 2, 2]$ where $\mathtt{b}(1, \mathtt{i}, \mathtt{j}) = 2(\mathtt{i}-1)+\mathtt{j}$.

problem arises due to the ordering of the axes. Yet, the ordering of the axes serves a useful purpose.

EXAMPLE 3.36 Look at the 1×3 matrix

$$\big[(1,1) \quad (1,2) \quad (1,3)\big]$$

in which the elements are the same as the locations. Now remove the 1-axis :

$$\big[(1) \quad (2) \quad (3)\big]$$

Apparently, no rotation is required.

An array may be viewed with many different orientations (labeling schemes). It is possible to change the orientation by relabeling axes. The work of relabeling, which maps one system of axes to another seemingly identical system, is delegated to the generating relation reax, defined next.

DEFINITION 3.12 (CONSTRUCTOR)

Let $\mathtt{k} \in \mathbb{N}$. The function

$$\mathrm{reax}_{\mathtt{k}} : \mathcal{A} \to \mathbb{H}$$

is given by

$$\mathrm{reax}_{\mathtt{k}}\ [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] = [\mathtt{a} \circ \langle\!\rangle_{\mathtt{k}} \mid \sqsupset_{\mathtt{k}} (1)(\mathtt{t} : \mathtt{n})].$$

Redundant operations should be avoided. Only one reax function is required to generate all arrays. Yet, infinitely many have been added. Nevertheless, it is precisely the right number. Since arrays possess several dimensions, it is possible to build up

arrays in several ways. Fairly or not, it is the operations, not the objects, that are held in high esteem. It is plausible to interpret reax as performing a rotation. However, reax is useful mainly as a relabeling relation that allows operations to be defined along different axes.

To specify the k-axis along which an operation takes place, it is preferable to write f_k instead of $f\ k$. With this convention, the role of k is clear. If a subscript, superscript or parameter is missing, it is taken to be 1, e.g., reax A reads as $\text{reax}_1\ A$.

Using multidimensional arrays, it is possible to perform many operations axially. Such operations can be partitioned as the data structure itself is partitioned. After partitioning an array into hyperplanes, a method is needed to put them back together again. In the theory of lists, an essential operator for handling lists is the concatenation operator. Concatenation is a natural operation for arrays. Such operations are available in commercial mathematics programs such as Maple V. A 1-array is a list. Since homomorphisms for lists are well-known, when concatenation is taken as a constructor, list homomorphisms are obtained for free.

For any n-array, there are exactly n **concatenation operators**:

$$\{\#_1, \ldots, \#_n\}.$$

This number of operators is needed to be able to build up arrays in all directions. An expression of the form $A_1 \#_k A_2 \#_k \cdots \#_k A_m$ for any finite set of arrays $\{A_i\}_{i \in \iota m}$ is a **concatenation** of the A_i's. The verb **concatenate** may be used to refer to combining two arrays via a concatenation operator. An operator $\#_k$ is defined for any n-array with $n \geq k$.

For any array A, define

$$A \#_k \square = \square \#_k A = A.$$

The empty array is the identity element for $\#_k$. Now all concatenations are defined whenever one of the arrays is an empty array. It is possible to concatenate any two nonempty arrays, provided that they have sufficiently similar shapes for stacking. The stacking requirement is that both given arrays have the "same shape ignoring the k-axis," which means the number n of axes is the same for both arrays and the lengths of all corresponding axes are equal, except possibly for the k-axis. In particular, it is possible to form a new array with the same shape as the given arrays ignoring the k-axis, and the length of the k-axis in the new array equals the sum of the lengths from the given arrays. Below, a concatenation of two 2-arrays is depicted, which corresponds

to adjoining a 2×1 matrix and a 2×2 matrix into a 2×3 matrix:

$$\left[\begin{bmatrix}1\\4\end{bmatrix} \rightarrowtail\!\!\!\!\leftarrowtail \begin{bmatrix}2 & 3\\5 & 6\end{bmatrix}\right] \longmapsto \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}$$

DEFINITION 3.13 (CONSTRUCTOR)

For $k \in \mathbb{N}$ the binary operator $+\!\!+_k$ is a partial function

$$\mathcal{A}^* \times \mathcal{A}^* \longrightarrow \mathcal{A}^*$$

that obeys

$$A +\!\!+_k \square = \square +\!\!+_k A = A,$$

and

$$[a\,|\,(t:n)] +\!\!+_k [b\,|\,(t:n; t_k = m)] = [c\,|\,(t:n; t_k = t_k + m)],$$

where

$$c(i:n) = \begin{cases} a(i:n) & \text{if } 1 \le i_k \le t_k \\ b(i:n; i_k = i_k - t_k) & \text{if } t_k < i_k \le t_k + m \end{cases}. \tag{3.1}$$

Informally, arrays are put together along the k-axis by placing the corresponding axes in the same line. Hence, it is possible to join two points (scalar arrays), a point (singleton) and a line (1-array), two lines, or two "boxes" that have common dimensions except for at most one. By convention, in any equation a concatenation operator has less precedence than any other constructor.

Although it may be possible to concatenate, the resulting array may be useless. In particular, if a n-array has a field direction k and it is desired to concatenate along the k-axis, there exist additional requirements, which concatenate does not enforce, on the types of elements along every k-axis. It may be desired to join two particular k-axes only if all elements along those axes have the same type. Regardless, concatenation is defined independently of the notion of field direction.

EXAMPLE 3.37 Join two singletons into a 1-array thusly

$$(\cdot)(5) +\!\!+ (\cdot)(7) = \overline{5,7}.$$

A collection of n-arrays is **conformable** if all arrays in the collection can be concatenated together in some way to yield a single n-array. All elements in the collection are used at least once to produce the result. An array is a **nested** array if every element in the array has type array.

EXAMPLE 3.38 Any nonoverlapping covering is a conformable collection.

EXAMPLE 3.39 The set $\{(\cdot)(\mathtt{A}), (\cdot)(\mathtt{B})\}$ is a conformable collection for any objects $\mathtt{A}$ and $\mathtt{B}$.

EXAMPLE 3.40 The set $\{\mathtt{A}, \text{reax } \mathtt{A}\}$ is *not* a conformable collection for any $\mathtt{n}$-array $\mathtt{A}$.

EXAMPLE 3.41 The array $[\mathtt{a} : 1, 5]$ with $\mathtt{a}(1, \mathtt{k}) = \mathtt{N}_\mathtt{k}$ is a nested array where $\mathtt{N}_\mathtt{k}$ is a squared 2-array (see Example 3.3).

DEFINITION 3.14 A $\mathtt{n}$-array $\mathtt{A} = [\mathtt{a} \mid \mathtt{T}]$ is a **tiled** array or a **tiling** if $\mathtt{A}$ is a nested array such that

$$\mathtt{a}(\mathtt{s}) \dashv\!\vdash_\mathtt{k} \mathtt{a}(\mathtt{t})$$

is defined for all $\mathtt{k} \in \iota\, \mathtt{n}$ and for all locations $\mathtt{s}, \mathtt{t} \in \mathtt{T}$ that are adjacent for $\mathtt{k} \in \mathtt{A}$.

Informally, a tiling is a nested array all of whose elements have the same shape along any axis ignoring that axis. Form a tiling by slicing up an array, inserting parallel planes all the way through.

EXAMPLE 3.42 The 1-array $(\cdot)(\mathtt{A})$ is a tiling for all arrays $\mathtt{A}$.

EXAMPLE 3.43 Choose $\mathtt{n} \in \mathbb{N}$ and put $\mathtt{B} = [\mathtt{b} : \mathtt{n}, \mathtt{n}]$ with

$$\mathtt{b}(\mathtt{i}, \mathtt{j}) = (\cdot)((\mathtt{n}-1)\,\mathtt{i} + \mathtt{j})\,.$$

Then $\mathtt{B}$ is a tiling for $\mathtt{N}_\mathtt{n}$, a squared 2-array (see Example 3.3).

3.3 DESTRUCTORS

Destructors select entire and distinct parts of objects or extract fundamental properties. For any given object, destructors yield either structural data, i.e., information about organization, or another object defined solely in terms of structural information. Begin by looking at familiar fundamental properties of an array.

3.3.1 FUNDAMENTAL SELECTIONS

In the theory of lists, a useful operation yields the length of a list, which is the number of elements in the list (Bird, 1987, p. 7). An analogous operation for arrays is

size, which gives the number of elements in the array. The function size is defined for all arrays.

DEFINITION 3.15 (DESTRUCTOR)

The function

$$\text{size} : \mathcal{A}^* \to \mathbb{N}_0$$

obeys

$$\text{size } \square = 0,$$

and

$$\text{size } [a \mid (t : n)] = \prod_{i=1}^{n} t_i.$$

EXAMPLE 3.44 The size of a 2×3 matrix is six.

EXAMPLE 3.45 Always $\text{size}\,[a : 7, 7] = \text{size}\left[a \mid \mathbb{C}_7^2\right] = 49$.

EXAMPLE 3.46 The size of an array with m rows and n columns is

$$\text{size}\,[a : m, n] = mn.$$

APL has an unary function ρ that returns the shape of an array. The function shape applied to a n-array produces a 1-array of lengths of each axis and is defined next.

DEFINITION 3.16 (DESTRUCTOR)

The function

$$\text{shape} : \mathcal{A}^* \to \mathbb{A}_{\mathbb{N}}^*$$

is given by the equations

$$\text{shape } \square = \square,$$

and

$$\text{shape } [a \mid (t : n)] = [t : n].$$

The result of a shape operation may also be referred as the **shape** of an array.

EXAMPLE 3.47 The shape of a squared 2-array is $[K_n : 2]$, where n is the number of rows or columns (see Example 3.3) and K_c is the constant function $K_c(x) = c$.

EXAMPLE 3.48 Applying shape to a vector produces a singleton that contains the length of the vector:

$$\text{shape } [a : n] = [K_n : 1].$$

EXAMPLE 3.49 For a 3-array $[\mathtt{a} : 2, 4, 6]$,

$$\text{shape } [\mathtt{a} : 2, 4, 6] = [\times 2 : 3].$$

To find the length of the $\mathtt{k}$-axis for a nonempty array $[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$, select the $\mathtt{k}$th element of the shape tuple $(\mathtt{t} : \mathtt{n})$.

DEFINITION 3.17 (DESTRUCTOR)

The function

$$\text{length} : \mathbb{N} \times \mathcal{A}^* \to \mathbb{N}_0$$

is given by

$$\text{length}_{\mathtt{k}}\ \square = 0$$

and for any $\mathtt{n}$-array $\mathtt{A}$ if $\mathtt{k} > \mathtt{n}$ then

$$\text{length}_{\mathtt{k}}\ \mathtt{A} = 0$$

and otherwise

$$\text{length}_{\mathtt{k}}\ [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] = \mathtt{t}_{\mathtt{k}}.$$

EXAMPLE 3.50 Let $\mathtt{f} = + \circ ((\times \circ \looparrowright 10) \circ \mathtt{div}\ 8, \mathtt{mod}\ 8) \circ \times$. Then the number of columns in a 4×7 octal multiplication table is

$$\text{length}_2\ [\mathtt{f} : 4, 7] = 7.$$

EXAMPLE 3.51 The number of rows in a 5×7 multiplication table is

$$\text{length } [\times : 5, 7] = 5.$$

Next, consider the valency of an array. In APL, it is possible to simply apply shape twice ($\rho\rho$). This operation is denoted by δ in (Mullin, 1988). Next, the function axes is defined, which yields the number of axes of a given array.

DEFINITION 3.18 (DESTRUCTOR)

The function

$$\text{axes} : \mathcal{A}^* \to \mathbb{N}_0$$

is given by

$$\text{axes } \square = 0,$$

and

$$\text{axes } [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] = \mathtt{n}.$$

EXAMPLE 3.52 An empty array has zero axes, any vector or 1-array has one axis and a 5×7 matrix or any 2-array has two axes.

EXAMPLE 3.53 The number of axes of squared 2-array is 2 (see Example 3.3).

EXAMPLE 3.54 A 1-array has only one axis: $\text{axes}[\mathtt{a} : 9] = 1$.

For completeness, consider the selector that retrieves the array function itself, namely the **function selector** fn which is defined next.

DEFINITION 3.19 (DESTRUCTOR)

For any n-array $[\mathtt{a} \mid \mathtt{T}]$,

$$\text{fn}[\mathtt{a} \mid \mathtt{T}] = \mathtt{a}|_{\mathbb{P}(\mathtt{T})},$$

which is the array function a restricted to $\mathbb{P}(\mathtt{T})$.

EXAMPLE 3.55 $\text{fn}[\times : 5, 4] = \times$ with domain $\mathbb{P}(5, 4)$ where $\times(\mathtt{a}, \mathtt{b}) = \mathtt{a} \times \mathtt{b}$.

3.3.2 RELABELING

The generating relation reax relabels locations. Inversely, it is possible to collapse any hyperplane, removing an unnecessary axis.

DEFINITION 3.20 (DESTRUCTOR)

The partial function

$$\text{collapse} : \mathbb{N} \times \mathbb{H} \to \mathcal{A}$$

is given by

$$\text{collapse}_{\mathtt{k}}[\mathtt{a} \mid \sqsupset_{\mathtt{k}}(1)(\mathtt{t} : \mathtt{n})] = [\mathtt{a} \circ \sqsupset_{\mathtt{k}}(1) \mid (\mathtt{t} : \mathtt{n})].$$

EXAMPLE 3.56 Consider the 2-array $\mathtt{A} = [\mathtt{a} : 1, 3] \sim [1\ 2\ 3]$, where $\mathtt{a}(1, \mathtt{k}) = \mathtt{k}$. Then $\text{collapse}[\mathtt{a} : 1, 3] = [\mathtt{b} : 3]$, where $\mathtt{b}(\mathtt{k}) = \mathtt{k}$, is a 1-array $\mathtt{B} \sim \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$.

3.3.3 BLOCK SELECTIONS

Arrays are comprised of blocks. An important observation for arrays is that the axes and the hyperplanes are ordered in a natural way. In this section, primitives are added to extract common subarrays.

3.3.4 INDEXING

The smallest blocks in an array are singletons. Next, define the selector index which retrieves the elements of an array. This primitive operation is a familiar one in functional programming (cf. for example Harrison, 1992, p. 557).

DEFINITION 3.21 (DESTRUCTOR)

The partial function

$$\text{index} : \mathcal{A}_{\mathtt{Y}} \times \mathbb{A}_{\mathbb{N}} \to \mathtt{Y}$$

is given by

$$\text{index}\ [\mathtt{a}\,|\,(\mathtt{t} : \mathtt{n})][\mathtt{i} : \mathtt{n}] = \mathtt{a}(\mathtt{i} : \mathtt{n}),$$

whenever $(\mathtt{i} : \mathtt{n}) \in \mathbb{P}(\mathtt{T})$.

The function index is not defined for invalid locations.

EXAMPLE 3.57 Let $\mathtt{t} = \overline{1, 999}$. Then index $\mathtt{N}_{1000}(\mathtt{t}) = 999$ where $\mathtt{N}_{1000}$ is a squared 2-array (see Example 3.3).

3.3.5 EXTRACTING AN AXIS

An array is comprised of a system of *parallel* axes (plus content). It is useful to be able to extract any axis. It is not a subarray as such objects possess more structure than needed. An axis may be regarded as a 1-array, that could be transformed into a subarray provided suitable reax operations were applied to the axis. An array may be regarded as an array of axes. In particular, any $\mathtt{n}$-array with $\mathtt{n} > 1$ is an $(\mathtt{n}-1)$-array of axes that are 1-arrays.

For this operation, a more expressive than concise form is chosen. To specify an axis, the field direction along with the head of the axis is required. For any given axis, the location for the head along the given axis is required to be one, which is not omitted in the specification.

DEFINITION 3.22 (DESTRUCTOR)

The partial function

$$\text{ax} : \mathbb{N} \times \mathcal{A} \times \mathbb{A}_{\mathbb{N}} \to \mathbb{A}$$

is given by

$$\text{ax}_{\mathtt{k}}\ [\mathtt{a}\,|\,(\mathtt{t} : \mathtt{n})]\ [\mathtt{h} : \mathtt{n}] = [\mathtt{a} \circ \boxminus : \mathtt{t}_{\mathtt{k}}],$$

where

$$\boxminus(\mathtt{j}) = \tau_{\mathtt{k}}(\mathtt{j})(\mathtt{h} : \mathtt{n}),$$

provided $k \in \iota n$ and necessarily

$$h_q \leq t_q \quad \text{for} \quad 1 \leq q < k,$$
$$h_k = 1,$$

and

$$h_q \leq t_{q+1} \quad \text{for} \quad k < q < n.$$

EXAMPLE 3.58 Observe that

$$ax_2\ N_4\ \overline{3,1} = [+8 : 4],$$

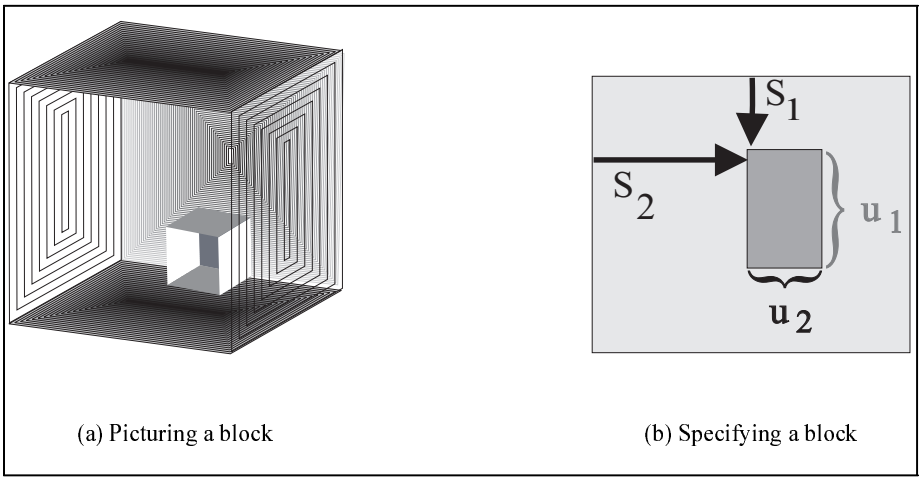
where N_4 is a squared 2-array (see Example 3.3).

3.3.6 EXTRACTING A BLOCK

A block is a contiguous subarray obtained by selecting a range of indices for each axis. A way to specify the desired block is to state

- the shifts, and
- the size of the block as depicted in Figure 3.6(b).

Both the translation, say s_k, and the length, say u_k, of the k-axis are required for each k. Unfortunately for large arrays there are many s_k's and u_k's. In Figure 3.6(a), a small block is pictured inside a larger array. In Figure 3.6(b), information needed to specify a block is shown.



(a) Picturing a block (b) Specifying a block

Figure 3.6: Picturing and specifying a block.

In view of the operations that are needed and the blocks that are used, it is often possible to arrange the data conveniently. For example, it may be possible to arrange that the blocks are simply rows of the array. In this case, it suffices to specify only the rows.

Some flexibility will be allowed when specifying the values of the $\mathtt{s}_\mathtt{k}$'s and $\mathtt{u}_\mathtt{k}$'s as explained next. Any missing shift $\mathtt{s}_\mathtt{k}$ will be taken to be zero, which means no shift, a shift of

$$\mathtt{s}_\mathtt{k} = \mathtt{p}$$

means shift over to the $(\mathtt{p}+1)\,\mathtt{st}$ hyperplane, and if $\mathtt{s}_\mathtt{k}$ is too large then the result will be an empty array. To avoid list orientations, negative shifts will also be permitted. For a $\mathtt{n}$-array $[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})]$, a shift

$$\mathtt{s}_\mathtt{k} = -\mathtt{p} < 0$$

will mean shift over to the $\mathtt{t}_\mathtt{k} - \mathtt{p} + 1$ hyperplane. If any length $\mathtt{u}_\mathtt{k}$ is missing or if $\mathtt{u}_\mathtt{k}$ is too large then the result will be as large as possible after translation.

DEFINITION 3.23 (DESTRUCTOR)

The function

$$\text{block} : \mathbb{A}^*_{\mathbb{Z}} \times \mathbb{A}^*_{\mathbb{N}} \times \mathcal{A}^* \to \mathcal{A}^*$$

is given by

$$\text{block}\ \mathtt{S}\ \mathtt{U}\ \square = \square$$

and for an arbitrary $\mathtt{n}$-array $\mathtt{A} = [\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})]$,

$$\text{block}\ \mathtt{S}\ \mathtt{U}\ \mathtt{A} = \mathtt{C}$$

where $\mathtt{C}$ is prescribed next. If $\mathtt{S} \neq \square$, then write $\mathtt{S} = [\mathtt{s} : \mathtt{m}_\mathtt{s}]$ and set

$$\sigma_\mathtt{i} = \mathtt{s}_\mathtt{i} \text{ for } 1 \leq \mathtt{i} \leq \mathtt{m}_\mathtt{s};$$

otherwise, set $\mathtt{m}_\mathtt{s} = 0$. For $\mathtt{m}_\mathtt{s} < \mathtt{k} \leq \mathtt{n}$, put $\sigma_\mathtt{k} = 0$. Then calculate

$$\mathtt{s}'_\mathtt{k} = \begin{cases} \sigma_\mathtt{k} & \text{if } \sigma_\mathtt{k} \geq 0 \\ \mathtt{t}_\mathtt{k} + \sigma_\mathtt{k} + 1 & \text{if } -(\mathtt{t}_\mathtt{k}+1) \leq \sigma_\mathtt{k} < 0 \\ \mathtt{t}_\mathtt{k} & \text{if } \sigma_\mathtt{k} < -(\mathtt{t}_\mathtt{k}+1) \end{cases}.$$

Let

$$\widehat{\mathtt{s}}_\mathtt{k} = \min\left\{\mathtt{s}'_\mathtt{k}, \mathtt{t}_\mathtt{k}\right\},\ \mathtt{k} \in \iota\,\mathtt{n}.$$

If $\widehat{\mathtt{s}}_\mathtt{k} = \mathtt{t}_\mathtt{k}$ for any $\mathtt{k}$ then

$$\mathtt{C} = \square.$$

If $\mathtt{U} \neq \square$, then write $\mathtt{U} = [\mathtt{u} : \mathtt{m}_\mathtt{u}]$ and set

$$\mu_\mathtt{i} = \mathtt{u}_\mathtt{i} \text{ for } 1 \leq \mathtt{i} \leq \mathtt{m}_\mathtt{u};$$

otherwise, set $m_u = 0$. For $m_u < k \leq n$, set $\mu_k = t_k$. Then compute

$$\hat{u}_k = \min\{\mu_k, t_k - \hat{s}_k\}, \quad k \in \iota n.$$

If $\hat{u}_k \leq 0$ for any k then take $C = \square$; otherwise, take

$$\begin{aligned} C &= [c|(\hat{u}:n)], \\ c(i:n) &= a\left(\gamma^{i,\hat{s}}:n\right) \end{aligned}$$

where

$$\gamma^{i,\hat{s}} = +\circ(i,\hat{s}).$$

EXAMPLE 3.59 Remove the first and last rows of a squared 2-array N_n via

$$\text{block } \neg\, \overline{n-2}\, N_n,$$

where n denotes the number of rows (see Example 3.3).

EXAMPLE 3.60 Given the 2-array

$$A \sim \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix},$$

extract the subarray $\left[\begin{smallmatrix} 3 & 4 \\ 8 & 9 \end{smallmatrix}\right]$ by calculating block $S\ U\ A$, where $S = \overline{0,2}$ and $U = \overline{2,2}$.

EXAMPLE 3.61 Compute block $\square\ \overline{2}\ A$ using the array A in Example 3.60 to obtain the first two rows:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \end{bmatrix}.$$

Whenever convenient, the result of a block operation is referred to as a **block array** or simply a **block**. Many algorithms rely on blocks. Extracting a block should be implemented as efficiently as possible. A block operation is available, for instance, in **occam 2**, a programming language with PAR, SEQ and ! (send) and ? (receive) constructs.

3.3.7 SELECTING HYPERPLANES

Consider any array that is a hyperplane for the k-axis. Upon collapsing by removing the k-axis, the result is an array that is *not* a subarray, even though it resembles the original array. On the other hand, partitioning a nonscalar array into a nonoverlapping collection of hyperplanes that forms a covering produces only subarrays.

EXAMPLE 3.62 Looking at $\mathtt{B}$ in Example 3.33, five hyperplanes for the $\mathtt{1}$-axis are given by

$$\mathtt{B}^{\mathtt{k}} = \left[\mathtt{b}^{\mathtt{k}} : \mathtt{1}, \mathtt{1}\right] = \left[\mathtt{b}^{\mathtt{k}} \middle| \mathtt{C}^{\mathtt{2}}\right]$$

where $\mathtt{b}^{\mathtt{k}}\left(\mathtt{C}^{\mathtt{2}}\right) = \mathtt{b}(\mathtt{k}, \mathtt{1}) = \mathtt{k}$ for $\mathtt{k} \in \iota\ \mathtt{5}$.

EXAMPLE 3.63 For $\mathtt{B}$ in Example 3.34, five hyperplanes for the $\mathtt{2}$-axis are given by $\mathtt{B}^{\mathtt{k}} = \left[\mathtt{b}^{\mathtt{k}} \middle| \mathtt{C}^{\mathtt{2}}\right]$ where $\mathtt{b}^{\mathtt{k}}\left(\mathtt{C}^{\mathtt{2}}\right) = \mathtt{b}(\mathtt{1}, \mathtt{k})$ for $\mathtt{k} \in \iota\ \mathtt{5}$.

EXAMPLE 3.64 Consider any squared $\mathtt{2}$-array $\mathtt{N}_{\mathtt{n}} = [\mathtt{a} : \mathtt{n}, \mathtt{n}] = \left[\mathtt{a} \middle| \mathtt{C}_{\mathtt{n}}^{\mathtt{2}}\right]$ (see Example 3.3). Then $\mathtt{n}$ hyperplanes for the $\mathtt{1}$-axis are formed via $\mathtt{B}^{\mathtt{k}} = \left[\mathtt{b}^{\mathtt{k}} : \mathtt{1}, \mathtt{n}\right]$ where $\mathtt{b}^{\mathtt{k}}(\mathtt{1}, \mathtt{j}) = \mathtt{a}(\mathtt{k}, \mathtt{j})$. In addition, $\mathtt{n}$ hyperplanes for the $\mathtt{2}$-axis are given by $\mathtt{C}^{\mathtt{k}} = \left[\mathtt{c}^{\mathtt{k}} : \mathtt{n}, \mathtt{1}\right]$ where $\mathtt{c}^{\mathtt{k}}(\mathtt{j}, \mathtt{1}) = \mathtt{a}(\mathtt{j}, \mathtt{k})$.

EXAMPLE 3.65 From $\mathtt{B}$ in Example 3.35, two hyperplanes for the $\mathtt{2}$-axis are given by $\mathtt{B}^{\mathtt{k}} = \left[\mathtt{b}^{\mathtt{k}} : \mathtt{1}, \mathtt{1}, \mathtt{2}\right]$ where $\mathtt{b}^{\mathtt{k}}(\mathtt{1}, \mathtt{1}, \mathtt{j}) = \mathtt{b}(\mathtt{1}, \mathtt{k}, \mathtt{j})$. Two hyperplanes for the $\mathtt{3}$-axis are $\mathtt{C}^{\mathtt{k}} = \left[\mathtt{c}^{\mathtt{k}} : \mathtt{1}, \mathtt{2}, \mathtt{1}\right]$ where $\mathtt{c}^{\mathtt{k}}(\mathtt{1}, \mathtt{j}, \mathtt{1}) = \mathtt{b}(\mathtt{1}, \mathtt{j}, \mathtt{k})$.

Arrays are concatenations of hyperplanes. It should be possible to select any hyperplane. A $\mathtt{k}$-axis must be specified. To select the $\mathtt{p}$th hyperplane, employ the function

$$\mathrm{slice}_{\mathtt{k}}\ \mathtt{p}.$$

Negative parameters will also be allowed. For a $\mathtt{n}$-array $[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$, if $\mathtt{p} < \mathtt{0}$ then for sufficiently large $\mathtt{p}$,

$$\mathrm{slice}_{\mathtt{k}}\ \mathtt{p}\ [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] = \mathrm{slice}_{\mathtt{k}}\ (\mathtt{t}_{\mathtt{k}} + \mathtt{p} + \mathtt{1})\ [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})].$$

DEFINITION 3.24 (DESTRUCTOR)

The function

$$\mathrm{slice} : \mathbb{N} \times \mathbb{Z} \times \mathcal{A}^* \to \mathcal{A}^*$$

is given by

$$\mathrm{slice}_{\mathtt{k}}\ \mathtt{p}\ \square = \square$$

and for any $\mathtt{n}$-array $\mathtt{A} = [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$, if $\mathtt{k} \notin \iota\ \mathtt{n}$ or $|\mathtt{p}| \notin \iota\ \mathtt{t}_{\mathtt{k}}$ (where $|\ |$ denotes the usual absolute value function, i.e., $|\mathtt{x}|$ equals $\mathtt{x}$ or $-\mathtt{x}$ if $\mathtt{x} \geq \mathtt{0}$ or $\mathtt{x} < \mathtt{0}$, respectively), then

$$\mathrm{slice}_{\mathtt{k}}\ \mathtt{p}\ \mathtt{A} = \square;$$

otherwise,

$$\mathrm{slice}_{\mathtt{k}}\ \mathtt{p}\ \mathtt{A} = [\widehat{\mathtt{a}} \mid (\mathtt{t} : \mathtt{n}; \mathtt{t}_{\mathtt{k}} = \mathtt{1})]$$

where

$$q = \begin{cases} p & \text{if } 1 \le p \le t_k \\ t_k + p + 1 & \text{if } -t_k \le p \le -1 \end{cases}$$

and

$$\hat{a}(i:n; i_k = 1) = a(i:n; i_k = q).$$

An infinite number of slice functions have been defined. The function slice_k is defined for all arrays.

EXAMPLE 3.66 Compute

$$\text{slice}_2\ N_5 = [a:5,1]$$

where $a(k,1) = 5k - 1$ (see Example 3.3).

Suppose it is desired to work with a subarray that is not a block. Imagine, for instance, crossing off every other row of a matrix. Take is a familiar list operation that has been defined for arrays (see for example Mullin, 1988). If only consecutive hyperplanes are used, apply block instead. Use take to make a sparse selection of hyperplanes.

It is possible to make selections along several axes. However, allowing more axes makes the operation more difficult to specify. Moreover, selecting too sparsely does not seem to be an efficient usage of arrays. For instance, selecting only one hyperplane on every axis corresponds to an indexing operation. So take shall be employed to select hyperplanes along only one axis. Notwithstanding, any subarray may be formed by composing a series of take operations. If several take operations are needed, investigate alternative solutions.

To specify a take operation, it is necessary to identify an axis, say k-axis, and the hyperplanes to take along that axis. As sparse selections are made, it is efficient to simply list the indices corresponding to the desired hyperplanes. It is natural to specify the indices in order. In the definition of take, a tacit assumption is that the indices are stored in a strictly increasing integer array B. If they are not in order, only the initial ones in order will be used (and the rest ignored).

Indices out of range will be ignored. For instance, taking the first and third rows of a 2-array that has only two rows, will yield only the first row. Given a positive integer 1-array $B = [b:p]$, the operation

$$\text{take}_k\ B$$

selects the

$$b_1\text{st}, b_2\text{nd}, \ldots, b_p\text{th}$$

hyperplanes in that order. Negative b_i's will be allowed provided

$$b_1 < b_2 < \cdots < b_p,$$

which determines the "**list partition**" of the hyperplanes. If any b_i is negative then it corresponds to selecting the $(t_k+b_i+1)\,st$ hyperplane.

DEFINITION 3.25 (DESTRUCTOR)

The function

$$\text{take} : \mathbb{N} \times \mathbb{A}_{\mathbb{Z}}^{*} \times \mathcal{A}^{*} \to \mathcal{A}^{*}$$

is given by

$$\text{take}_k \ \square \ A = \square$$

$$\text{take}_k \ B \ \square = \square$$

, for $B \neq \square$ and $A \neq \square$

$$\text{take}_k \ B \ A = C,$$

which is specified next. Write $B = [b : m]$ and $A = [a \,|\, (t : n)]$. If $k > n$ then assign $C = \square$; otherwise, set

$$S = \{i \in \mathbb{N} | b_j < b_{j+1} \quad \text{for} \quad 1 \leq j \leq i < m\},$$

and

$$q = 1 + \max(S \cup \{0\}).$$

For $1 \leq i \leq q$ set

$$\beta_i = \begin{cases} t_k + b_i + 1 & \text{if } -t_k \leq b_1 \leq -1 \\ b_i & \text{otherwise} \end{cases}$$

and

$$S' = \{i \,|\, 1 \leq \beta_i \leq t_k\}.$$

If $S' = \varnothing$ then take $C = \square$; otherwise, put

$$s = \min(S'),$$

$$u = 1 - s + \max(S'),$$

and take

$$C = [\widehat{a} \,|\, (t : n; t_k = u)]$$

where

$$\widehat{a}(i : n) = a(i : n; i_k = \beta_{i_k+s-1}).$$

EXAMPLE 3.67 Consider the squared 2-array $N_3 = [a : 3, 3] = \left[a \,|\, \mathbb{C}_3^2\right]$ (see Example

3.3). To select the first and third rows, compute

$$\text{take } \mathtt{B}\,\mathtt{N}_3 \sim \begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{bmatrix},$$

where $\mathtt{B} = \overline{\mathtt{1,3}}$. To select the first and third columns, compute

$$\text{take}_2\ \mathtt{B}\,\mathtt{N}_3 \sim \begin{bmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{bmatrix}.$$

The result is a $\mathtt{2}$-array in both cases.

The function take is a fairly powerful function. To accomplish a similar task using lists, it is necessary to apply a composition of functions (see for example Axford and Joy, 1993). The operations block and take are important enough to warrant some overlap. For example, if only take could be used to form blocks then arrays would be processed like lists as it would be necessary to compose a sequence of take operations to form a single block.

3.3.8 DELETING HYPERPLANES

Suppose it is desired to work with a large subarray that is not a block. Imagine, for instance, crossing off a single row in the middle of a large matrix. Although it is possible to use take, it is tedious to list so many indices. Instead, use drop to delete a sparse number of hyperplanes along an axis. Any subarray may then be formed via a composition of drop operations.

To specify a drop operation, it is necessary to identify an axis, say $\mathtt{k}$-axis, and the hyperplanes to drop along that axis. Any extra indices will be ignored. For instance, dropping the first and third rows of a $\mathtt{2}$-array that has only two rows, will yield merely the second row. Given a positive integer $\mathtt{1}$-array $\mathtt{B} = [\mathtt{b} : \mathtt{p}]$, the operation

$$\text{drop}_{\mathtt{k}}\ \mathtt{B}$$

drops the

$$\mathtt{b}_1\mathtt{st}, \mathtt{b}_2\mathtt{nd}, \ldots, \mathtt{b}_\mathtt{p}\mathtt{th}$$

hyperplanes in that order. Negative $\mathtt{b_i}$'s are permitted provided

$$\mathtt{b}_1 < \mathtt{b}_2 < \cdots < \mathtt{b}_\mathtt{p}.$$

If a $\mathtt{b_i}$ is negative then it corresponds to removing the $(\mathtt{t}_\mathtt{k} + \mathtt{b}_\mathtt{i} + \mathtt{1})\,\mathtt{st}$ hyperplane.

DEFINITION 3.26 (DESTRUCTOR)

The function

$$\text{drop} : \mathbb{N} \times \mathbb{A}_{\mathbb{Z}}^{*} \times \mathcal{A}^{*} \to \mathcal{A}^{*}$$

is given by

$$\text{drop}_{\mathtt{k}}\ \square\ \mathtt{A} = \mathtt{A},$$

$$\text{drop}_{\mathtt{k}}\ \mathtt{B}\ \square = \square,$$

for $\mathtt{B} \neq \square$ and $\mathtt{A} \neq \square$

$$\text{drop}_{\mathtt{k}}\ \mathtt{B}\ \mathtt{A} = \mathtt{C}$$

which is specified next. Write $\mathtt{B} = [\mathtt{b} : \mathtt{m}]$ and $\mathtt{A} = [\mathtt{a} \,|\, (\mathtt{t} : \mathtt{n})]$. If $\mathtt{k} > \mathtt{n}$ then assign $\mathtt{C} = \mathtt{A}$. If $\mathtt{k} \leq \mathtt{n}$ then set

$$\mathtt{S} = \{\mathtt{i} \in \mathbb{N} \,|\, \mathtt{b}_{\mathtt{j}} < \mathtt{b}_{\mathtt{j}+1} \quad \text{for} \quad 1 \leq \mathtt{j} \leq \mathtt{i} < \mathtt{m}\},$$

$$\mathtt{q} = 1 + \max(\mathtt{S} \cup \{0\}),$$

and for $1 \leq \mathtt{i} \leq \mathtt{q}$ put

$$\beta_{\mathtt{i}} = \begin{cases} \mathtt{t}_{\mathtt{k}} + \mathtt{b}_{\mathtt{i}} + 1 & \text{if } -\mathtt{t}_{\mathtt{k}} \leq \mathtt{b}_1 \leq -1 \\ \mathtt{b}_{\mathtt{i}} & \text{otherwise} \end{cases}.$$

Let

$$\mathtt{S}' = \{\mathtt{i} \,|\, 1 \leq \beta_{\mathtt{i}} \leq \mathtt{t}_{\mathtt{k}}\}.$$

If $\mathtt{S}' = \varnothing$ then take $\mathtt{C} = \mathtt{A}$; otherwise, let

$$\mathtt{u} = \max(\mathtt{S}') - \min(\mathtt{S}') + 1.$$

If $\mathtt{u} = \mathtt{t}_{\mathtt{k}}$ then take $\mathtt{C} = \square$; otherwise, take

$$\mathtt{C} = [\widehat{\mathtt{a}} \,|\, (\mathtt{t} : \mathtt{n}; \mathtt{t}_{\mathtt{k}} = \mathtt{t}_{\mathtt{k}} - \mathtt{u})]$$

where

$$\widehat{\mathtt{a}}(\mathtt{i} : \mathtt{n}) = \mathtt{a}(\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{k}} = \mathtt{s}_{\mathtt{i}_{\mathtt{k}}}),$$

$$\mathtt{s}_0 = 0,$$

and

$$\mathtt{s}_{\mathtt{q}} = \min\left\{\mathtt{j} \in \mathbb{N} \,|\, \mathtt{j} > \mathtt{s}_{\mathtt{q}-1}, \mathtt{j} \notin \left\{\beta_{\mathtt{i}} \,|\, \min(\mathtt{S}') \leq \mathtt{i} \leq \max(\mathtt{S}')\right\}\right\}.$$

EXAMPLE 3.68 Consider the squared 2-array $\mathtt{N}_3 = [\mathtt{a} : 3, 3] = \left[\mathtt{a} \,|\, \mathtt{C}_3^2\right]$ (see Example 3.3). To delete the second row, compute

$$\text{drop}\ \overline{2}\ \mathtt{N}_3 \sim \begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{bmatrix}.$$

To delete the second column, calculate

$$\mathrm{drop}_2\ \overline{2}\ \mathtt{N}_3 \sim \begin{bmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{bmatrix}.$$

The result is a 2-array in both cases.

3.4 ABSTRACTORS

Up to this point, arrays have not yet been fully explained. Nothing will be added to model the objects in terms of the data structure, which has been completely incorporated into the constructors and destructors. There are purposes for the data type which cannot be understood by studying the data structure alone. These purposes are determined by the patterns of computations which are performed on the data type. It is paramount that common program structures be accounted for in the theory. The intention is that such common computational patterns are recognized via the abstractors. In this section all abstractors are introduced.

Not all functions need to be defined as homomorphisms. It is not intended to subsume the theory under some restrictive algebraic model. All components of the sets of equations in the theory should be taken into account.

3.4.1 HIGHER-ORDER FUNCTIONS

In general, binary operators take operands that need not be collections of objects. When a binary operator is placed between arrays, however, there is a special case that stands out from the general case of combining two operands. In particular, there is a pointwise case in which the binary operator is "lowered" to the level of the elements. The higher-order partial function $\underset{\longleftrightarrow}{}$ is used to suggest this lowering of the binary operator to the elements.

DEFINITION 3.27 (ABSTRACTOR)

The higher-order partial function

$$\underset{\longleftrightarrow}{} : \{\mathtt{X} \times \mathtt{X} \to \mathtt{Y}\} \times \mathcal{A}^*_{\mathtt{X}} \times \mathcal{A}^*_{\mathtt{X}} \to \mathcal{A}^*_{\mathtt{Y}}$$

lowers any binary operator $\oplus : \mathtt{X} \times \mathtt{X} \to \mathtt{Y}$ to the partial function

$$\underset{\longleftrightarrow}{\oplus} : \mathcal{A}^*_{\mathtt{X}} \times \mathcal{A}^*_{\mathtt{X}} \to \mathcal{A}^*_{\mathtt{Y}}$$

defined pointwise by

$$\square \underset{\longleftrightarrow}{\oplus} \square = \square,$$

and

$$[\mathtt{a}\,|\,\mathtt{T}] \underset{\longleftrightarrow}{\oplus} [\mathtt{b}\,|\,\mathtt{T}] = [\oplus \circ (\mathtt{a},\mathtt{b})\,|\,\mathtt{T}].$$

Suppose, for instance, that

$$\mathtt{A} \underset{\longleftrightarrow}{\oplus} \square = \square \underset{\longleftrightarrow}{\oplus} \mathtt{A} = \mathtt{A}.$$

Consider usual addition of matrices. Let $\mathtt{Z}$ denote the array with the same shape as $\mathtt{A}$ and only zero entries. Plainly,

$$\mathtt{A} \underset{\longleftrightarrow}{+} \mathtt{Z} = \mathtt{Z} \underset{\longleftrightarrow}{+} \mathtt{A} = \mathtt{A}.$$

There would be two distinct identity elements under usual addition unless $\mathtt{A} = \square$.

Call $\underset{\longleftrightarrow}{\ }\mathtt{zip}$, which is a familiar function in functional programming. The function $\mathtt{zip}$ preserves the data structure of the operands. No special information such as a parameter is required to define this operation. This function expresses a common pattern of computation. Homomorphisms distribute across concatenation, which is an adjoining of "similar pairs" and pairing itself is a special case of $\mathtt{zip}$.

EXAMPLE 3.69 Compute the sum of two squared 2-arrays:

$$\mathtt{N}_\mathtt{n} \underset{\longleftrightarrow}{+} \mathtt{N}_\mathtt{n} = [\mathtt{a} : \mathtt{n}, \mathtt{n}]$$

where $\mathtt{a}(\mathtt{i},\mathtt{j}) = 2(\mathtt{n}(\mathtt{i}-1)+\mathtt{j})$ (see Example 3.3).

A familiar pointwise operation is $\mathtt{map}$ denoted by $*$.

DEFINITION 3.28 (ABSTRACTOR)

The higher-order function $* : \{\mathtt{X} \to \mathtt{Y}\} \times \mathcal{A}^*_\mathtt{X} \to \mathcal{A}^*_\mathtt{Y}$ lifts any function $\mathtt{f} : \mathtt{X} \to \mathtt{Y}$ on the elements to an unary function

$$*\mathtt{f} : \mathcal{A}^*_\mathtt{X} \to \mathcal{A}^*_\mathtt{Y}$$

which is defined pointwise via

$$*\mathtt{f}\,\square = \square,$$

and

$$*\mathtt{f}[\mathtt{a}\,|\,\mathtt{T}] = [\mathtt{f} \circ \mathtt{a}\,|\,\mathtt{T}].$$

A $\mathtt{map}$ preserves the data structure of the argument. This abstractor is primitive since it is defined for any array given any suitable function.

EXAMPLE 3.70 Calculate

$$\begin{aligned} *(+2)\overline{3,6,9,\cdots,(3\mathtt{n})} &= *(+2)[\times 3 : \mathtt{n}] = [+2 \circ \times 3 : \mathtt{n}] \\ &= \overline{5,8,11,\cdots,(3\mathtt{n}+2)}. \end{aligned}$$

EXAMPLE 3.71 Let

$$\mathtt{A} = [\mathtt{a} : 2, 2] \sim \begin{bmatrix} 4 & 9 \\ 16 & 25 \end{bmatrix}$$

where $\mathtt{a}$ is given by $\mathtt{a}(\mathtt{i}, \mathtt{j}) = (2(\mathtt{i}-1)+\mathtt{j}+1)^2$. If $\sqrt{\ }$ denotes the usual square root function, then

$$*\sqrt{\ }\,\mathtt{A} = [\sqrt{\ } \circ \mathtt{a} : 2, 2] \sim \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}.$$

It may be possible to combine all the elements by placing a furnished binary operator $\oplus$ between all the elements listed in arbitrary order. In this case, an array is viewed mainly as a collection of homogeneous elements as in a set. Mostly, the data structure is ignored. The symbol $\bigsqcup$ is used as it closely resembles set union and is called **combine**.

DEFINITION 3.29 (ABSTRACTOR)

The higher-order partial function

$$\bigsqcup : \{\mathtt{X} \times \mathtt{X} \to \mathtt{Y}\} \times \mathtt{X} \times \mathcal{A}^*_{\mathtt{X}} \to \mathcal{A}^*_{\mathtt{Y}}$$

takes an associative binary operator $\oplus : \mathtt{X} \times \mathtt{X} \to \mathtt{Y}$ with identity element $\mathtt{e} \in \mathtt{X}$ and an argument in $\mathcal{A}^*_{\mathtt{X}}$ and obeys

$$\bigsqcup \oplus \square = \mathtt{e},$$

and

$$\bigsqcup \oplus [\mathtt{a} \mid \mathtt{T}] = \bigoplus_{\mathtt{t} \in \mathbb{P}(\mathtt{T})} \mathtt{a}(\mathtt{t}).$$

EXAMPLE 3.72 Let $\mathtt{B} = [\mathtt{b} : 1, 2, 2]$ where $\mathtt{b}(1, \mathtt{i}, \mathtt{j}) = 2(\mathtt{i}-1)+\mathtt{j}$ which is depicted in Figure 3.5. Then

$$\bigsqcup \times \mathtt{B} = \prod_{\mathtt{k}=1}^{4} \mathtt{k} = 1 \times 2 \times 3 \times 4 = 24,$$

where $\times$ denotes usual multiplication.

For lists, there is only one axis and so it is acceptable to regard the data type as one of lists of elements of *homogeneous type*. For nonscalar arrays with more than one axis it is a narrow view to think of the data type as arrays of elements of homogeneous type. Many useful operations may be defined for nonhomogeneous arrays. The data structure for arrays especially allows different types on different axes.

It is possible to apply not only a single (unary) operation to an array but also an array of operations to an array. There is no reason why the elements of an array cannot

be functions. In order to operate on arrays in a manner befitting the data type, some operations should possess a structure resembling the data structure. *By adding arrays of functions, a high degree of uniformity in the program structure and the data structure will be achieved.*

The element functions may be applied to an array as shown in Figure 3.7. This type of operation is a higher-order operation which is implicitly highly parallel. In order for a formalism to incorporate a successful programming paradigm, higher-order operations should capture common patterns of computations in suitable ways.

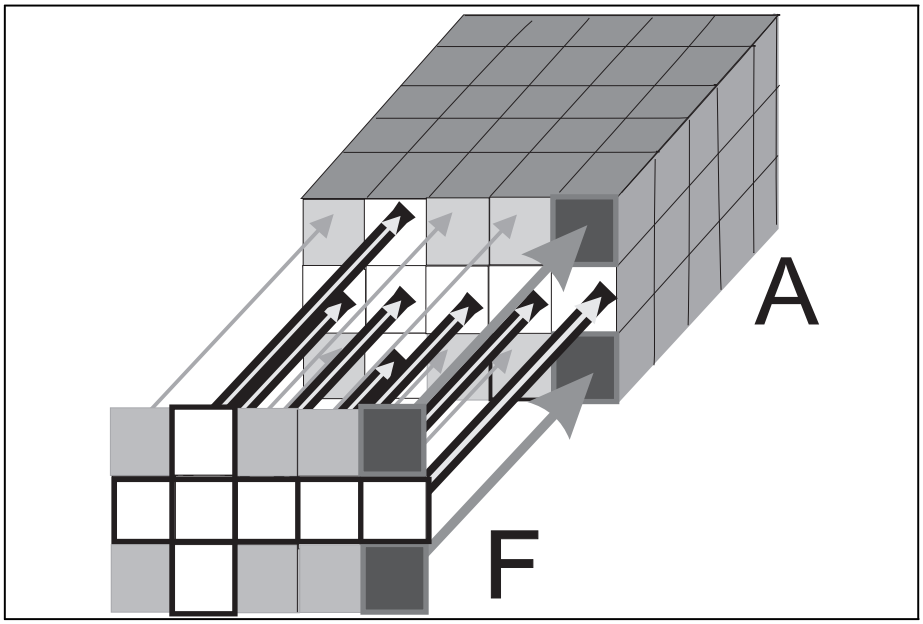


Figure 3.7: A function of F is mapped to elements along an axis of A.

Consider first the simpler case when the argument has the same shape as the array of functions.

DEFINITION 3.30 (ABSTRACTOR)

The binary partial function

$$\multimap : \mathcal{A}^* \times \mathcal{A}^* \to \mathcal{A}^*$$

is defined pointwise by

$$\multimap \square\ \square = \square,$$

and

$$\multimap [\mathtt{f} \mid \mathtt{T}]\ [\mathtt{a} \mid \mathtt{T}] = [@\circ(\mathtt{f}, \mathtt{a}) \mid \mathtt{T}].$$

Recall

$$@\circ(\mathtt{f}, \mathtt{a})(\mathtt{t}) = @\circ(\mathtt{f}(\mathtt{t}), \mathtt{a}(\mathtt{t})) = \mathtt{f}(\mathtt{t})(\mathtt{a}(\mathtt{t})).$$

Call $\multimap$ bicomp. This operation preserves the data structure of the operands. The operation bicomp is defined only for suitable pairs of arrays.

EXAMPLE 3.73 Let

$$\mathtt{F} = \left[\mathtt{f} \,|\, \mathbb{C}_2^2\right] \sim \begin{bmatrix} +1 & +2 \\ +3 & +4 \end{bmatrix}$$

where f is given by $\mathtt{f}(\mathtt{i},\mathtt{j}) = +(2(\mathtt{i}-1)+\mathtt{j})$. Let A be as in Example 3.71. Calculate

$$\multimap \mathtt{F}\,\mathtt{A} = \left[\mathtt{b} \,|\, \mathbb{C}_2^2\right] \sim \begin{bmatrix} 5 & 11 \\ 19 & 29 \end{bmatrix},$$

where $\mathtt{b}(\mathtt{i},\mathtt{j}) = (2(\mathtt{i}-1)+\mathtt{j}+1)^2 + (2(\mathtt{i}-1)+\mathtt{j})$.

Consider second the case when the argument has some depth as depicted in Figure 3.7.

DEFINITION 3.31 (ABSTRACTOR)

The higher-order partial function

$$\circledast : \mathbb{N} \times \mathcal{A} \times \mathcal{A}^2 \to \mathcal{A}$$

satisfies

$$\circledast_{\mathtt{k}}\left[\mathtt{f} \,|\, (\mathtt{t}:\mathtt{n})\right]\left[\mathtt{x} \,|\, \sqsupset_{\mathtt{k}}(\mathtt{s})(\mathtt{t}:\mathtt{n})\right] = \left[\mathtt{u} \,|\, (\mathtt{t}:\mathtt{n})\right]$$

where

$$\mathtt{u}(\mathtt{i}:\mathtt{n}) = \mathtt{f}(\mathtt{i}:\mathtt{n})\left[\mathtt{b}^{(\mathtt{i}:\mathtt{n})} : \mathtt{s}\right]$$

and the 1-array $\left[\mathtt{b}^{(\mathtt{i}:\mathtt{n})} : \mathtt{s}\right]$ is an axis of X whose elements are given by

$$\mathtt{b}_{\mathtt{j}}^{(\mathtt{i}:\mathtt{n})} = \mathtt{x} \circ \sqsupset_{\mathtt{k}}(\mathtt{j})(\mathtt{i}:\mathtt{n}).$$

Call $\circledast$ biaxis since it pairs functions with axes. This abstractor applies all operations in the same field direction. This partial function is defined for only suitable pairs of arrays.

EXAMPLE 3.74 Consider the 2-array A in Example 3.60. Let F equal the 1-array $\left[\mathtt{K}_{\mathrm{length}} : 3\right]$ where $\mathtt{K}_{\mathtt{f}}$ is the constant function $\mathtt{K}_{\mathtt{f}}(\mathtt{i}) = \mathtt{f}$ for all functions f. Then

$$\circledast_2 \mathtt{F}\,\mathtt{A} = [\mathtt{y}:3] \sim \begin{bmatrix} 5 \\ 5 \\ 5 \end{bmatrix}$$

where $\mathtt{y}_{\mathtt{k}} = \mathrm{length}\left[\mathtt{b}^{\mathtt{k}} : 5\right] = 5$.

Another important operation utilizes an array of binary operators instead of just one as in a zip operation.

DEFINITION 3.32 (ABSTRACTOR)

The higher-order partial function

$$\overleftrightarrow{\ } : \mathcal{A} \times \mathcal{A}^* \times \mathcal{A}^* \to \mathcal{A}^*$$

combines two operands using a furnished array of binary operators:

$$\overleftrightarrow{[\oplus \mid \mathtt{T}]}\ \square\ \square = \square,$$
$$\overleftrightarrow{[\oplus \mid \mathtt{T}]}\ [\mathtt{a} \mid \mathtt{T}]\ [\mathtt{b} \mid \mathtt{T}] = [\mathtt{tribi} \circ (\oplus, \mathtt{a}, \mathtt{b}) \mid \mathtt{T}],$$

where $\mathtt{tribi} \circ (\oplus, \mathtt{a}, \mathtt{b})(\mathtt{t}) = \mathtt{tribi} \circ (\oplus(\mathtt{t}), \mathtt{a}(\mathtt{t}), \mathtt{b}(\mathtt{t})) = \mathtt{a}(\mathtt{t}) \oplus_{\mathtt{t}} \mathtt{b}(\mathtt{t})$.

Call $\overleftrightarrow{\ }$ `triad`. Whenever convenient, write

$$\overleftrightarrow{\ominus}\ [\mathtt{a} \mid \mathtt{T}]\ [\mathtt{b} \mid \mathtt{T}] = [\mathtt{a} \mid \mathtt{T}] \overleftrightarrow{\ominus} [\mathtt{b} \mid \mathtt{T}],$$

where $\ominus = [\oplus \mid \mathtt{T}]$.

EXAMPLE 3.75 Let $\ominus = \left[\oplus \mid \mathtt{C}_2^2\right] \sim \left[\begin{smallmatrix} \div & \times \\ + & - \end{smallmatrix}\right]$. Let $\mathtt{A}$ be as in Example 3.71. Then

$$\overleftrightarrow{\ominus}\ \mathtt{A}\ \mathtt{N}_2 \sim \begin{bmatrix} 4 & 18 \\ 19 & 21 \end{bmatrix}$$

where $\mathtt{N}_2$ is a squared 2-array (see Example 3.3).

Next, three functions are defined to perform a specified operation along a given axis so that only part of array (argument) is used. Such operations are important, for instance, in database queries. In all cases, it is necessary to state the field direction and the head of the axis along which the operation is performed. Although all elements in the array need not have the same type, all elements along an axis must have the same type for the binary operation to be defined.

DEFINITION 3.33 (ABSTRACTOR)

The higher-order partial function

$$\smile : \mathbb{N} \times \{\mathtt{X} \times \mathtt{X} \to \mathtt{Y}\} \times \mathbb{A}^n \times \mathbb{N}^n \to \mathtt{Y}$$

is given by

$$\smile_{\mathtt{k}} \oplus\ [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]\ \mathtt{h} =$$
$$((\cdots(\mathtt{a} \circ \tau_{\mathtt{k}}(1)\mathtt{h} \oplus \mathtt{a} \circ \tau_{\mathtt{k}}(2)\mathtt{h}) \cdots) \oplus \mathtt{a} \circ \tau_{\mathtt{k}}(\mathtt{t}_{\mathtt{k}} - 1)\mathtt{h}) \oplus \mathtt{a} \circ \tau_{\mathtt{k}}(\mathtt{t}_{\mathtt{k}})\mathtt{h},$$

whenever $\mathtt{k}$ is a valid field direction and $\mathtt{h}$ is the head of a $\mathtt{k}$-axis for any $[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] \in \mathbb{A}^{\mathtt{n}}$, provided all operations are defined.

DEFINITION 3.34 (ABSTRACTOR)

The higher-order partial function

$$\frown : \mathbb{N} \times \{\mathtt{X} \times \mathtt{X} \to \mathtt{Y}\} \times \mathbb{A}^{\mathtt{n}} \times \mathbb{N}^{\mathtt{n}} \to \mathtt{Y}$$

is given by

$$\frown_{\mathtt{k}} \oplus [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]\ \mathtt{h} =$$
$$\mathtt{a} \circ \tau_{\mathtt{k}}(1)\,\mathtt{h} \oplus (\mathtt{a} \circ \tau_{\mathtt{k}}(2)\,\mathtt{h} \oplus (\cdots (\mathtt{a} \circ \tau_{\mathtt{k}}(\mathtt{t}_{\mathtt{k}} - 1)\,\mathtt{h} \oplus \mathtt{a} \circ \tau_{\mathtt{k}}(\mathtt{t}_{\mathtt{k}})\,\mathtt{h}) \cdots)),$$

whenever $\mathtt{k}$ is a valid field direction and $\mathtt{h}$ is the head of a $\mathtt{k}$-axis for any $[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] \in \mathbb{A}^{\mathtt{n}}$, provided all operations are defined.

DEFINITION 3.35 (ABSTRACTOR)

The higher-order partial function

$$\asymp : \mathbb{N} \times \{\mathtt{X} \times \mathtt{X} \to \mathtt{Y}\} \times \mathbb{A}^{\mathtt{n}} \times \mathbb{N}^{\mathtt{n}} \to \mathtt{Y}$$

is given by

$$\asymp_{\mathtt{k}} \oplus [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]\ \mathtt{h} = \mathtt{a} \circ \tau_{\mathtt{k}}(1)\,\mathtt{h} \oplus \mathtt{a} \circ \tau_{\mathtt{k}}(2)\,\mathtt{h} \oplus \cdots \oplus \mathtt{a} \circ \tau_{\mathtt{k}}(\mathtt{t}_{\mathtt{k}})\,\mathtt{h},$$

whenever $\mathtt{k}$ is a valid field direction and $\mathtt{h}$ is the head of a $\mathtt{k}$-axis for any $[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] \in \mathbb{A}^{\mathtt{n}}$, provided $\oplus$ is an associative operator.

The operations $\smile_{\mathtt{k}}\oplus$, $\frown_{\mathtt{k}}\oplus$ and $\asymp_{\mathtt{k}}\oplus$ may be referred to as **upper**, **lower** and **mixed folds**, respectively.

EXAMPLE 3.76 Consider the squared 2-array $\mathtt{N}_3 = [\mathtt{a} : 3, 3] = \left[\mathtt{a} \mid \mathbb{C}_3^2\right]$ (see Example 3.3). Let

$$\mathtt{a} \oplus \mathtt{b} = \mathtt{a} - \frac{\mathtt{b}}{2}.$$

Then

$$\begin{aligned}
\smile \oplus \mathtt{N}_3\ (1,2) &= -\frac{9}{2},\\
\frown \oplus \mathtt{N}_3\ (1,2) &= \frac{3}{2},\\
\smile_2 \oplus \mathtt{N}_3\ (2,1) &= -\frac{3}{2},\\
\frown_2 \oplus \mathtt{N}_3\ (2,1) &= 3,\\
\asymp - \mathtt{N}_3\ (1,2) &= -11,
\end{aligned}$$

and

$$\asymp_2 - \mathtt{N}_3\,(2,1) = -7.$$

3.4.2 PERMUTATIONS

The notions of permutations and arrays are closely interwoven. Recall the concept of a permutation matrix.

EXAMPLE 3.77 Usual multiplication of matrices yields

$$\begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix}\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} = \begin{bmatrix}4&5&6\\7&8&9\\1&2&3\end{bmatrix}$$

and

$$\begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} = \begin{bmatrix}7&8&9\\1&2&3\\4&5&6\end{bmatrix}.$$

Multiplying by zeros and ones is inefficient. Besides, such computations can only be performed on arrays whose elements are of appropriate type. Basically a permutation is a rearrangement which can be performed on any array.

DEFINITION 3.36 A **n-permutation** is a sequence $\iota\,\mathtt{n} \to \iota\,\mathtt{n}$ that is one-to-one and onto.

The problem is not to compute a $\mathtt{n}$-permutation. Instead, the problem is to reassemble according to the given $\mathtt{n}$-permutation.

DEFINITION 3.37 (ABSTRACTOR)

The higher-order partial function

$$\circlearrowleft\colon \mathbb{N} \times (\iota\,\mathtt{m} \to \iota\,\mathtt{m}) \times \mathcal{A} \to \mathcal{A}$$

is given by

$$\circlearrowleft_{\mathtt{k}}\,\mathtt{p}\,[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n};\mathtt{t}_{\mathtt{k}} = \mathtt{m})] = [\mathtt{b}\,|\,(\mathtt{t}:\mathtt{n};\mathtt{t}_{\mathtt{k}} = \mathtt{m})]$$

where $\mathtt{b}(\mathtt{i}:\mathtt{n}) = \mathtt{a}\left(\mathtt{i}:\mathtt{n};\mathtt{i}_{\mathtt{k}} = \mathtt{p}^{-1}(\mathtt{i}_{\mathtt{k}})\right)$.

Call $\circlearrowleft$ **permute**, which rearranges hyperplanes in any given field direction $\mathtt{k}$.

EXAMPLE 3.78 Let $\mathtt{p}_{\mathtt{k}} = 1 + (\mathtt{k} \bmod 3)$. Then $\mathtt{p}_1 = 2$, $\mathtt{p}_2 = 3$ and $\mathtt{p}_3 = 1$ is a

3-permutation. Permute the rows of N_3 via

$$\circlearrowleft p\, N_3 = [b|C_3^2] \sim \begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.$$

where N_3 is the squared 2-array with three rows and three columns (see previous example or Example 3.3).

EXAMPLE 3.79 Let $p_k = 1 + (k \bmod 3)$. Permute the columns of N_3 via

$$\circlearrowleft_2 p\, N_3 = [b|C_3^2] \sim \begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}.$$

3.4.3 PARTITIONING OPERATIONS

Any array may be built-up by stacking hyperplanes. Yet, an array is much more than a list of hyperplanes. The capability to partition an array into blocks that have the same data structure sets arrays apart from other models. An array is not merely a collection of elements or even a set of pairs of elements with unique locations. An array has a "mesh" that can be partitioned into subarrays, which can be operated upon and reassembled. Arrays exist partly because it is useful to partition operations as arrays themselves are partitioned. The widespread usage of block algorithms attests to this fact. Since arrays may be nested,

$$*f\, A$$

may be used to apply an operation to every block of a tiled array. It remains to define primitive operations to form the blocks themselves. Such block operations are primitive, i.e., they are defined for all arrays, since they depend only on the data structure.

BORDER

For various applications, it may be desired to enlarge an array by duplicating entries around the border. Imagine that every n-array is a subarray of an **infinite array** by extending the axes in all directions and copying data from the nearest location in the n-array. Consider partitioning the infinite array into a subarray which is the original array plus a border.

Call a 1-array $B = [b : n]$ a border array if it specifies the number b_k of additional entries needed along the k-axis, possibly zero, on both sides, for each $k \in \iota\, n$. The

thickness of the border is assumed to be symmetric, i.e., the border has the same number of elements on both sides along any axis. This requirement does not imply that an operation must use all elements on both sides.

DEFINITION 3.38 (ABSTRACTOR)

The function

$$\text{border} : \mathbb{A}^*_{\mathbb{N}_0} \times \mathcal{A}^* \to \mathcal{A}^*$$

is given by

$$\text{border } \mathtt{B}\ \square = \square,$$

$$\text{border } \square\ \mathtt{A} = \mathtt{A},$$

$$\text{border } \mathtt{B}\ [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] = \left[\mathtt{c} \mid \left(\gamma^{\mathtt{t},\mathtt{b}} : \mathtt{n}\right)\right],$$

where

$$\gamma^{\mathtt{t},\mathtt{b}} = + \circ (\mathtt{t},\ \times 2 \circ \mathtt{b})$$

and letting $\mathtt{B} = \left[\widehat{\mathtt{b}} : \widehat{\mathtt{n}}\right]$,

$$\mathtt{b}_\mathtt{k} = \begin{cases} \widehat{\mathtt{b}}_\mathtt{k} & \text{if } 1 \le \mathtt{k} \le \widehat{\mathtt{n}} \\ 0 & \text{if } \widehat{\mathtt{n}} < \mathtt{k} \le \mathtt{n} \end{cases},$$

$$\mathtt{c}(\mathtt{i} : \mathtt{n}) = \mathtt{a}\left(\widehat{\mathtt{i}} : \mathtt{n}\right),$$

and

$$\widehat{\mathtt{i}_\mathtt{k}} = \min\{\mathtt{t}_\mathtt{k},\ \max\{1,\ \mathtt{i}_\mathtt{k} - \mathtt{b}_\mathtt{k}\}\}.$$

EXAMPLE 3.80 Taking $\mathtt{B} = \overline{\mathtt{C}^2}$ and $\mathtt{N}_2 = \left[\mathtt{a} \mid \mathtt{C}^2_2\right] \sim \left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$,

$$\text{border } \mathtt{B}\ \mathtt{N}_2 \sim \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}.$$

3.4.4 UNBORDER

Next, an operation to remove a border is defined. This operation may be applied to any $\mathtt{n}$-array that is large enough to allow the removal of the specified border. No checking is done to determine if the exterior elements are actually duplicates of the interior elements.

DEFINITION 3.39 (ABSTRACTOR)

The function

$$\text{unborder} : \mathbb{A}^*_{\mathbb{N}_0} \times \mathcal{A}^* \to \mathcal{A}^*$$

is given by

$$\text{unborder } \mathtt{B}\ \square = \square,$$

$$\text{unborder } \square\ \mathtt{A} = \mathtt{A},$$

$$\text{unborder}\,\mathtt{B}\ [\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})] = \begin{cases} \square & \text{if } \mathtt{t}_k - 2\mathtt{b}_k < 1 \text{ for some } \mathtt{k} \\ [\mathtt{c}\,|\,(\gamma^{\mathtt{t},\mathtt{b}}:\mathtt{n})] & \text{otherwise} \end{cases},$$

where

$$\gamma^{\mathtt{t},\mathtt{b}} = -\circ(\mathtt{t},\times 2\circ\mathtt{b})$$

and letting $\mathtt{B} = \left[\widehat{\mathtt{b}} : \widehat{\mathtt{n}}\right]$,

$$\mathtt{b}_k = \begin{cases} \widehat{\mathtt{b}}_k & 1 \le \mathtt{k} \le \widehat{\mathtt{n}} \\ 0 & \widehat{\mathtt{n}} < \mathtt{k} \le \mathtt{n} \end{cases},$$

$$\mathtt{c}(\mathtt{i}:\mathtt{n}) = \mathtt{a}(+\circ(\mathtt{i},\mathtt{b}):\mathtt{n}).$$

EXAMPLE 3.81 Taking $\mathtt{B} = \overline{\mathtt{C}^2}$ and $\mathtt{N}_3 \sim \left[\begin{smallmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{smallmatrix}\right]$,

$$\text{unborder } \mathtt{B}\ \mathtt{N}_3 = \left[\mathtt{c}\,|\,\mathtt{C}^2\right] \sim [5].$$

3.4.5 SPLIT

The function split splits an array along some k-axis at some specified location.

DEFINITION 3.40 (ABSTRACTOR)

The function

$$\text{split} : \mathbb{N} \times \mathbb{Z} \times \mathcal{A}^* \to \mathcal{A}^* \times \mathcal{A}^*$$

is given by

$$\text{split}_k\ \mathtt{p}\ \mathtt{A} = \square \qquad \text{if } \mathtt{A} = \square \text{ or } \mathtt{A} = [\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})] \text{ and } \mathtt{k} \notin \iota\,\mathtt{n},$$

and

$$\text{split}_k\ \mathtt{p}\ \mathtt{A} = \begin{cases} (\square, \mathtt{A}) & \text{if } \mathtt{p} \le 0 \\ (\mathtt{A}, \square) & \text{if } \mathtt{p} \ge \mathtt{t}_k \\ (\mathtt{B}, \mathtt{C}) & \text{otherwise} \end{cases},$$

where

$$\mathtt{A} = [\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})],$$

$$\mathtt{B} = [\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n};\mathtt{t}_k = \mathtt{p})],$$

$$\mathtt{C} = [\mathtt{a}\circ\tau_k(\mathtt{p}+1)\,|\,(\mathtt{t}:\mathtt{n};\mathtt{t}_k = \mathtt{t}_k - \mathtt{p})].$$

EXAMPLE 3.82 For instance, split $2\ \mathtt{N}_3$ (see Example 3.3) yields a pair $(\mathtt{B}, \mathtt{C})$ where $\mathtt{B}$ consists of the first two rows and $\mathtt{C}$ holds the last row as indicated below

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \mapsto \left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 8 & 9 \end{bmatrix} \right)$$

3.4.6 COVER

Next, a covering operation is added to the set of abstractors. First, consider a simpler partitioning of a matrix.

EXAMPLE 3.83 Partition a 2-array $\mathtt{A} = \left[\mathtt{a} \mid \mathtt{C}_4^2\right]$ into two parts to obtain a 1-array $\mathtt{D} = [\mathtt{d} : 2]$ where $\mathtt{d}_1$ contains the top two rows and $\mathtt{d}_2$ contains the bottom two rows as depicted below:

$$\mathtt{A} \sim \begin{bmatrix} 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 \\ 23 & 24 & 25 & 26 \end{bmatrix} \mapsto \mathtt{D} \sim \begin{bmatrix} [11\ 12\ 13\ 14] \\ [15\ 16\ 17\ 18] \\ \\ [19\ 20\ 21\ 22] \\ [23\ 24\ 25\ 26] \end{bmatrix}.$$

Fix $\mathtt{n}$. Consider any $\mathtt{n}$-array $\mathtt{A} = [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$. Let the desired partitions be specified via a **partition pair** $(\mathtt{X}, \mathtt{P})$. The 1-array $\mathtt{X} = [\mathtt{x} : \mathtt{m}]$ is an **axes array** that specifies the axes along which partitioning takes place with $\mathtt{x}_\mathtt{i} < \mathtt{x}_\mathtt{j}$ whenever $\mathtt{i} < \mathtt{j}$, i.e., partitioning takes place along the $\mathtt{x}_\mathtt{i}$-axis for each $\mathtt{i} = 1, \ldots, \mathtt{m}$. It is natural to require that axes are listed in order since locations and hyperplanes are ordered. Associate each partition with an axis. This association is implicit. The **partition array** $\mathtt{P} = [\mathtt{p} : \mathtt{m}]$ is a nested 1-array and each element of this partition array is called a **partition**. A partition

$$\mathtt{p}_\mathtt{k} = \left[\mathtt{h}^\mathtt{k} : \mathtt{l}^\mathtt{k}\right], \mathtt{l}^\mathtt{k} \leq \mathtt{t}_{\mathtt{x}_\mathtt{k}},\ 1 \leq \mathtt{k} \leq \mathtt{m},$$

is a 1-array of partitioning locations $\mathtt{h}_\mathtt{i}^\mathtt{k}$ such that

$$1 \leq \mathtt{h}_1^\mathtt{k} < \mathtt{h}_2^\mathtt{k} < \cdots < \mathtt{h}_{\mathtt{l}^\mathtt{k}}^\mathtt{k} = \mathtt{t}_{\mathtt{x}_\mathtt{k}}.$$

This last chain of inequalities delineates a valid partition. After partitioning in the new array of blocks, the length of the $\mathtt{x}_\mathtt{k}$-axis for the $\mathtt{i}$th block is $\mathtt{h}_\mathtt{i}^\mathtt{k} - \mathtt{h}_{\mathtt{i}-1}^\mathtt{k}$ with $\mathtt{h}_0^\mathtt{k} = 0$. So $\mathtt{h}_\mathtt{i}^\mathtt{k}$ is one the locations where the original array is partitioned and all indices from $1 + \mathtt{h}_{\mathtt{i}-1}^\mathtt{k}$ to $\mathtt{h}_\mathtt{i}^\mathtt{k}$ in the $\mathtt{i}$th block are included in the partition.

EXAMPLE 3.84 In Example 3.83, $\mathtt{X} = [\mathtt{x} : 1]$, $\mathtt{x}_1 = 1$, $\mathtt{P} = [\mathtt{p} : 1]$, $\mathtt{p}_1 = [\mathtt{h} : 2]$, $\mathtt{h}_1 = 2$ and $\mathtt{h}_2 = 4$.

Generalize the partitioning strategy as follows. After partitioning, in order to perform some operation on a block, suppose that near the boundary, additional information is needed from a neighboring block (see for example Kolesnikov and Trishina, 1995, p. 10). For this reason, add a border around each block and call the operation cover since the array is covered with blocks, possibly overlapping. Call these blocks **cells** as they are part of a large "honeycomb." Any $\mathtt{n}$-array that is the result of a cover operation is called a **cover array**.

The width of the border is specified via a $\mathtt{1}$-array $\mathtt{B} = [\mathtt{b} : \mathtt{n}]$, which is a border array. Note the border array prescribes the number of elements taken from an adjacent block and elements are duplicated if there are no such elements. It is not required to add a border on any axis and so any of the $\mathtt{b}_\mathtt{k}$'s may be zero.

If every axis is not partitioned, it is possible to store the resulting tiled array as an array with less valency. However, the valency should be maintained so that reconstruction is possible by putting the blocks back together to form the original array. In general, a covering operation is applied only if some (partial) reconstruction will be subsequently performed later with minimal effort. In summary, preserve the valency of the original array regardless of the number of axes along which partitioning takes place.

In the following definition, it is implicit that $(\mathtt{X}, \mathtt{P})$ is a suitable partition pair, i.e., the $\mathtt{x}_\mathtt{k}$-axis corresponds to the partition $\mathtt{p}_\mathtt{k}$. As usual, some flexibility will be allowed in the specification of the partition pair. If

$$\mathtt{h}^{\mathtt{k}}_{\mathtt{l}^{\mathtt{k}}} > \mathtt{t}_{\mathtt{x}_\mathtt{k}},$$

then the last partition location is set to $\mathtt{t}_{\mathtt{x}_\mathtt{k}}$. Although $\mathtt{X}$ and $\mathtt{P}$ should have the same length, they may have different lengths. Any invalid values at the end of $\mathtt{X}$ and $\mathtt{P}$ will be ignored.

DEFINITION 3.41 (ABSTRACTOR)

The function

$$\text{cover} : \mathbb{A}^*_{\mathbb{N}_0} \times \mathbb{A}^*_{\mathbb{N}} \times \mathbb{A}^*_{\mathbb{A}^*_{\mathbb{N}}} \times \mathcal{A}^* \to \mathcal{A}$$

is given by

$$\text{cover}\ \mathtt{B}\ \mathtt{X}\ \mathtt{P}\ \mathtt{A} = \begin{cases} (\cdot)\,(\text{border}\ \mathtt{B}\ \mathtt{A}) & \text{if}\ \mathtt{X} = \square, \mathtt{P} = \square, \mathtt{A} = \square \text{ or } \delta = \mathtt{0} \\ [\mathtt{c} : (\mathtt{z} : \mathtt{n})] & \text{otherwise} \end{cases},$$

where the sequence $\mathtt{z}$, function $\mathtt{c}$, and δ are prescribed next. Let $\mathtt{A} = [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$, $\mathtt{X} = [\mathtt{x} : \mathtt{m}]$, $\mathtt{x}_\mathtt{0} = \mathtt{0}$ and

$$\delta = \max(\{\mathtt{0}\} \cup \{\mathtt{w} \mid \mathtt{x}_{\mathtt{k}-\mathtt{1}} < \mathtt{x}_\mathtt{k} \leq \mathtt{n} \text{ for all } \mathtt{k} \text{ with } \mathtt{1} \leq \mathtt{k} \leq \mathtt{w} \leq \mathtt{m}\}).$$

If $\delta \neq 0$ write $P = [p : \hat{m}]$ and set $\delta = \min\{\hat{m}, \delta\}$. If

$$p_k = \square \text{ for some } k \text{ with } 1 \leq k \leq \delta$$

put $\delta = 0$. Let

$$p_k = \left[h^k : l^k\right] \quad \text{and} \quad h_0^k = 0 \quad \text{for} \quad 1 \leq k \leq \delta.$$

For $k \in \iota\, \delta$, set

$$S^k = \left\{i \in \mathbb{N} \mid h_{j-1}^k < h_j^k \leq t_{x_k} \text{ for all } j \text{ with } 1 \leq j \leq i \leq l^k\right\},$$

and

$$r_k = \max\left(S^k \cup \{0\}\right).$$

For $k \in \iota\, \delta$ with $h_{r_k}^k < t_{x_k}$, set

$$r_k = r_k + 1, h_{r_k}^k = t_{x_k}.$$

If

$$0 = \min_{1 \leq k \leq \delta} \{r_k\},$$

then assign $\delta = 0$. Although the actual implementation is irrelevant, in order to specify the q_i's precisely, initialize

$$q_k = 0 \text{ for all } k \in \iota\, n.$$

Then set

$$q_{x_k} = k \text{ for all } k \in \iota\, \delta.$$

Put

$$z_k = \begin{cases} 1 & q_k = 0 \\ r_{q_k} & q_k \neq 0 \end{cases}.$$

If $B = \square$, then set $\hat{n} = 0$; otherwise, write $B = \left[\hat{b} : \hat{n}\right]$. Lastly, set

$$b_k = \begin{cases} \hat{b}_k & \text{for} \quad 1 \leq k \leq \hat{n} \\ 0 & \text{for} \quad \hat{n} < k \leq n \end{cases},$$

$$c(i : n) = \left[\alpha^{(i:n)} \mid \left(u^{(i:n)} : n\right)\right],$$

$$u_k^{(i:n)} = \begin{cases} t_k + 2b_k & \text{if } q_k = 0 \\ h_{i_k}^{q_k} - h_{i_k - 1}^{q_k} + 2b_k & \text{if } q_k \neq 0 \end{cases},$$

$$\alpha^{(i:n)}(j : n) = a\left(\gamma^{i,j} : n\right),$$

$$\gamma_k^{i,j} = \phi(i_k, j_k),$$

$$\phi(i_k, j_k) = \begin{cases} \min\{\max\{1, j_k - b_k\}, t_k\} & \text{if } q_k = 0 \\ \min\left\{\max\left\{1, j_k + h^{q_k}_{i_k - 1} - b_k\right\}, t_k\right\} & \text{if } q_k \neq 0 \end{cases}.$$

Only a small portion of the indices actually make an appearance in any equation, regardless of the valency of the array or the number of axes along which partitioning takes place. The valency of a block in a cover array must be constant and equals the valency of the original array.

EXAMPLE 3.85 Continuing from Example 3.83 and Example 3.84,

$$\text{cover } \mathtt{B\ X\ P\ A} = \mathtt{E},$$

where $\mathtt{B} = \square$ and $\mathtt{E} = [\mathtt{e} : 2, 1]$ with $\mathtt{e}(1,1) = d_1$ and $\mathtt{e}(2,1) = d_2$.

EXAMPLE 3.86 Replacing $\mathtt{B}$ in Example 3.85 by $\mathtt{B} = \overline{1,2}$,

$$\text{cover } \mathtt{B\ X\ P\ A} = \mathtt{E} \sim \left[\begin{bmatrix} 11 & 11 & 11 & 12 & 13 & 14 & 14 & 14 \\ 11 & 11 & 11 & 12 & 13 & 14 & 14 & 14 \\ 15 & 15 & 15 & 16 & 17 & 18 & 18 & 18 \\ 19 & 19 & 19 & 20 & 21 & 22 & 22 & 22 \end{bmatrix}\right.$$
$$\left.\begin{bmatrix} 15 & 15 & 15 & 16 & 17 & 18 & 18 & 18 \\ 19 & 19 & 19 & 20 & 21 & 22 & 22 & 22 \\ 23 & 23 & 23 & 24 & 25 & 26 & 26 & 26 \\ 23 & 23 & 23 & 24 & 25 & 26 & 26 & 26 \end{bmatrix}\right].$$

EXAMPLE 3.87 Taking $\mathtt{B} = \overline{\mathtt{C}^2}$ (add one element on the border), $\mathtt{X} = \overline{1,2}$ (partition on both axes), $\mathtt{P} = [\mathtt{p} : 2]$, $\mathtt{p}_1 = \left[\mathtt{h}^1 : 2\right]$, $\mathtt{p}_2 = \left[\mathtt{h}^2 : 2\right]$, $\mathtt{h}^1 = \overline{\mathtt{C}^2}$, $\mathtt{h}^2 = \overline{\mathtt{C}^2}$, ("cut" at every location), and $\mathtt{N}_2 \sim \left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$,

$$\text{cover } \mathtt{B\ X\ P\ N}_2 = \mathtt{F} \sim \left[\begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 2 \\ 3 & 3 & 4 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 2 \\ 1 & 2 & 2 \\ 3 & 4 & 4 \end{bmatrix}\right.$$
$$\left.\begin{bmatrix} 1 & 1 & 2 \\ 3 & 3 & 4 \\ 3 & 3 & 4 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 2 \\ 3 & 4 & 4 \\ 3 & 4 & 4 \end{bmatrix}\right].$$

3.4.7 UNTILE

Presumably, a cover is performed in order to work with smaller blocks and with the intention to reassemble as in divide-and-conquer algorithms. Although additional elements from adjacent blocks might be needed for some computation, it is inefficient to compute elements that are going to be discarded. If it is needed to remove some border from all blocks, then apply

$$*\text{unborder}.$$

Hence, borders are not removed during reassembly.

EXAMPLE 3.88 Letting $\mathtt{E}$ be the 2-array in Example 3.86,

$$*\left(\text{unborder } \overline{\square^2}\right)\mathtt{E} \quad \sim \quad \begin{bmatrix} [11\ 11\ 12\ 13\ 14\ 14] \\ [15\ 15\ 16\ 17\ 18\ 18] \\ \\ [19\ 19\ 20\ 21\ 22\ 22] \\ [23\ 23\ 24\ 25\ 26\ 26] \end{bmatrix}.$$

There are potential problems when reassembling. One problem is that the given array might not be a tiling. Another potential problem is that an element could be an empty array. This is not a problem if all elements in a hyperplane are empty arrays. Suppose for example there is only one element which is an empty array in a large array. One solution is to remove a hyperplane which contains the empty array. But then there would be more than one solution as there is more than one hyperplane that contains the empty array. An empty array is a tool that in this instance is not useful.

For an untile operation, assume that the given array is a tiled array and that all elements have the same valency. Given a tiled array, it is possible that blocks or cells could be put together in more than one way. However, untile is essentially a left inverse of a cover operation. The structure has been preserved. No additional information is needed to untile.

DEFINITION 3.42 (ABSTRACTOR)

The partial function

$$\text{untile} : \mathcal{A} \to \mathcal{A}^*$$

is given by

$$\text{untile}(\cdot)\,\square = \square,$$

and

$$\text{untile}\,[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] = [\mathtt{u} \mid (\sigma : \mathtt{n})],$$

where the array function $\mathtt{u}$ and sequence σ are defined next. Write

$$\mathtt{a}(\mathtt{i} : \mathtt{n}) = \left[\alpha^{(\mathtt{i}:\mathtt{n})} \mid \left(\mathtt{s}^{(\mathtt{i}:\mathtt{n})} : \mathtt{n}\right)\right].$$

Let $\mathtt{SEQ}$ denote sequential computation and $\mathtt{PAR}$ denote parallel computation (if possible). Compute as follows:
($\mathtt{SEQ}$)

1. For $\mathtt{k} = 1, \ldots, \mathtt{n}$, $\mathtt{p} = 1, \ldots, \mathtt{t}_\mathtt{k}$, ($\mathtt{PARDO}$)

$$\mathtt{h}_0^\mathtt{k} = 0,$$

$$\mathtt{h}_\mathtt{p}^\mathtt{k} = \sum_{\mathtt{q}=1}^{\mathtt{p}} \mathtt{s}_\mathtt{k}^{\tau_\mathtt{k}(\mathtt{q})\square^\mathtt{n}};$$

2. For $k = 1, \ldots, n$, $p = 1, \ldots, t_k$, (PARDO)

$$\psi^k\left(j + h^k_{p-1}\right) = p \text{ for all } j \text{ with } 1 \le j \le s_k^{\tau_k(p)\mathcal{C}_n};$$

3. (PAR)

   (a) For $k = 1, \ldots, n$, (PARDO) $\sigma_k = h^k_{t_k}$,

   (b) Letting $\gamma^i_k = \psi^k_{i_k}$ and $\omega^{i,h,\gamma}_k = i_k - h^k_{\gamma^i_k}$ put

$$u(i:n) = \alpha^{(\gamma^i:n)}\left(\omega^{i,h,\gamma}:n\right).$$

Some programming constructs indicate the potential parallelism (PARDO). Each implementation should exploit the potential parallelism for the particular architecture.

EXAMPLE 3.89 Let F be the 1-array in Example 3.88. Then

$$\text{untile}\, F = G \sim \begin{bmatrix} 11 & 11 & 12 & 13 & 14 & 14 \\ 15 & 15 & 16 & 17 & 18 & 18 \\ 19 & 19 & 20 & 21 & 22 & 22 \\ 23 & 23 & 24 & 25 & 26 & 26 \end{bmatrix}.$$

3.5 MACROS

An operation is a **macro** if it is an alias or shorthand for some function that is defined. For various applications or formalisms, suitable macros may be provided for handling common operations. Briefly, a few examples are given next.

Every n-array has a first item. It is convenient to use the macro first given by

$$\texttt{first} = \text{index} \circ (\texttt{id}_{\mathcal{A}^*}, \boxdot \circ (K_{K_1}, \text{axes})),$$

where $K_c(x) = c$ is a constant function,

$$(K_{K_1}, \text{axes})[a \mid (t:n)] = (K_{K_1}[a \mid (t:n)], \text{axes}[a \mid (t:n)]) = (K_1, n)$$

and

$$\boxdot(K_1, n) = [K_1 : n].$$

Alternatively, it is easier to write

$$\texttt{first}\, A = \text{index}\, A\ \overline{\mathcal{C}^\infty}.$$

It is useful to define macros for handling common blocks. To take all initial (except the last) rows of a n-array, use the macro init given by

$$\texttt{init} = (\text{block}\ \square) \circ ((\cdot) \circ (-1) \circ \text{length}, \texttt{id}_{\mathcal{A}^*}).$$

For convenience, define the macro

$$\texttt{initial k} = \text{block } \square\ \overline{\texttt{k}},$$

which takes the first k rows (or k elements for a 1-array). To take all rows of a n-array except the first one, employ the macro tail given by

$$\texttt{tail} = \text{block } \urcorner\ \square.$$

Note init and tail are applicable to any n-array. A macro to partition an array into blocks is

$$\texttt{dice k} = \text{cover } \square\ \urcorner\ (\cdot)[\times\circ\!\!\searrow \texttt{k} : \infty],$$

which applied to a 1-array produces a nested 1-array of blocks of size k. For instance, if covering a 1-array that has 128 elements with blocks of size 16 then the partition is

$$\overline{\texttt{16,32,48,64,80,96,112,128}}.$$

Next macros are given for common take operations. To select the kth column of a n-array with $\texttt{n} > 1$, use col:

$$\texttt{col k} = \text{take}_2\ \overline{\texttt{k}}.$$

To take the leftmost and rightmost columns, employ

$$\begin{aligned} \texttt{col} &= \text{take}_2\ \urcorner, \\ \texttt{col}_\texttt{r} &= \text{take}_2\ \overline{-1}. \end{aligned}$$

The macros col and $\texttt{col}_\texttt{r}$ are defined for any n-array with $\texttt{n} > 1$. Notice $\texttt{col} = \texttt{col 1}$.

When it is desired to partition without overlap, apply a tile macro:

$$\texttt{tile (X, P)} = \text{cover } \square\ \texttt{X P}.$$

Note in tile P, P is a partition pair $(\texttt{X}, \texttt{P}_\texttt{X})$ and necessarily

$$(\texttt{tile P})\,\texttt{A} = \text{cover } \square\ \texttt{X}\,\texttt{P}_\texttt{X}\,\texttt{A}.$$

3.6 EXAMPLES

A few more examples are given next to show how to specify arrays.

EXAMPLE 3.90 Let $f : \mathbb{N} \to X$ be any function. Then

$$[f : n]$$

is a 1-array. For example,

$$[\times \circ \rho \mapsto 2 : 9] \sim \begin{bmatrix} 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & 18 \end{bmatrix}.$$

EXAMPLE 3.91 The Kronecker delta function

$$K^{\delta} : \mathcal{N} \to \{0, 1\}$$

is given by

$$K^{\delta}(T) = \begin{cases} 1 & \text{if } T = C_k^n \text{ for some } n, k \in \mathbb{N} \\ 0 & \text{otherwise} \end{cases}.$$

Call $\left[K^{\delta} \mid C_k^n\right]$ an **n-dimensional identity array of order** k. Recall the notion of an identity matrix for matrix multiplication. For instance, the 2-dimensional identity array of order 8 is

$$\left[K^{\delta} : 8, 8\right] \sim \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix}.$$

EXAMPLE 3.92 Let the constant function on nonempty tuples

$$K_c : \mathcal{N} \to \{c\}$$

be given by

$$(i : n) \mapsto c.$$

Call any array of the form $[K_0 \mid (t : n)]$ a **zero array**. Note it is necessary to state the number of axes n as well as all of the t_i's. For instance,

$$[K_0 : 5, 9] \sim \begin{bmatrix} 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \end{bmatrix}.$$

3.7 REVIEW

There are valid arguments for modifying the prescribed set of primitive operations. Perhaps for some application area, an important pattern of computation should be incorporated into the set of abstractors. Nevertheless, it is unnecessary to add or remove operations from this set to program with multidimensional arrays. In this sense, the construction of the set of primitives is adequate. The notation is consistent and concise. It became possible to simplify these definitions by introducing elementary func-

tions such as τ_k, $\sqsupset_k$, and $\lozenge$. Nevertheless, it may be possible to improve the notation by changing some of the names or symbols used. Most importantly, all the primitives were well-defined. It would be beneficial to find more elegant ways to formulate some of the definitions such as take, block and collapse.

A primitive model for multidimensional arrays was presented. A n-array, empty array, hyperplane, scalar array, singleton, k-axis and k-order were defined. All arrays possess the same type. The following symbols for sets of arrays were adopted: $\mathbb{A}(\mathtt{T})$, $\mathcal{A}^k$, $\mathcal{A}^*$ and $\mathbb{H}$. Descriptive terminology was introduced to facilitate reasoning about arrays, including the following concepts: field direction, subarrays, contiguous subarrays, overlapping collections, coverings and tilings. The list function was added to succinctly translate from lists to 1-arrays and other ways were provided to specify arrays for parameters and arguments to functions. The iota array $\urcorner_n = \overline{1,\ldots,n}$ and the constant-k by n array $\overline{\mathtt{C}^n_k}$ were defined. Macros and many examples of arrays were given. A complete set of primitive operations was specified which includes new functions that account for common program structures and are listed next:

Constructors:

$\square$	an empty array
$(\cdot)$	produces a scalar array
reax	relabels axes
$+\!\!+$	concatenates or stacks

Destructors:

index	selects an element
fn	retrieves the array function
axes	yields the number of axes (valency)
size	yields the number of elements
length	yields the length of an axis
shape	yields the shape
ax	extracts an axis as a 1-array
block	extracts a block
collapse	collapses a hyperplane by relabeling
slice	slices out a hyperplane
take	selects a stack of hyperplanes
drop	drops a list of hyperplanes

Abstractors:

$\overleftrightarrow{\ominus}$ (triad)	applies operator to corresponding pairs
$\underleftrightarrow{\oplus}$ (zip)	zips with a furnished binary operator
$\bigsqcup\oplus$	combines all elements
$*$ (map)	maps a given function to the elements
$\multimap$ (bicomp)	maps functions pointwise
$\circledast$ (biaxis)	maps functions to axes
$\smile_k \oplus$, $\frown_k \oplus$, $\asymp_k \oplus$	applies operation to a given axis
$\circlearrowleft$	permutes hyperplanes
split	splits an array into a pair
border / unborder	adds/removes a border
cover / untile	covers/untiles an array

CHAPTER 4

A CONSTRUCTIVE THEORY

In this chapter the consequences of the definitions are formally investigated in order to develop a constructive theory for multidimensional arrays. A primitive array model was introduced to define the objects and the primitive operations. All other operations are defined in terms of the primitive operations, not in terms of this model. It is intended to show that there is a sound theoretical basis for programming using the primitive operations. It is also shown that the primitive operations are expressive. Array homomorphisms are explicitly classified and important properties are verified. Useful formulas that lack many of the indices typically required are stated. To avoid distractions, proofs involving tedious details have been placed in the Appendix.

4.1 FUNDAMENTAL IDENTITIES

The consequences of the definitions are explored next.

CLAIM 4.1 If $U, V \in S^n$ with $U = V$ then there exists a sequence t with domain ιn such that $U = (t : n) = V$.

PROOF As $U \in S^n$ there exists a sequence a with domain ιn such that $a_k \in S$ for $k \in \iota n$ and $U = (a_1, \ldots, a_n)$. Write $U = (a : n)$. Similarly, as $V \in S^n$ there exists a sequence b with domain ιn such that $V = (b : n)$. Since $U = V$, by definition of equality for n-tuples, $a_k = b_k$ for $k \in \iota n$. Define the sequence t on ιn by

$$t_k = a_k.$$

By definition of equality for sequences $a = t = b$ on ιn. ■

Next it is established that the representation for a n-array is well-defined.

CLAIM 4.2 If $\mathtt{A}$ and $\mathtt{B}$ are $\mathtt{n}$-arrays with $\mathtt{A} = \mathtt{B}$ then there exists $\mathtt{T} \in \mathbb{N}^{\mathtt{n}}$ and an array function $\mathtt{a}$ defined on $\mathbb{P}(\mathtt{T})$ such that

$$\mathtt{A} = [\mathtt{a} \mid \mathtt{T}] = \mathtt{B}.$$

PROOF Suppose $\mathtt{A}$ and $\mathtt{B}$ are $\mathtt{n}$-arrays with $\mathtt{A} = \mathtt{B}$. By definition of $\mathtt{n}$-array, there exists an array function $\mathtt{c}$ and a $\mathtt{n}$-tuple $\mathtt{U}$ such that $\mathtt{A} = [\mathtt{c} \mid \mathtt{U}]$ and $\mathtt{c}(\mathtt{q})$ is defined for all $\mathtt{q} \in \mathbb{P}(\mathtt{U})$. Similarly, there exists an array function $\mathtt{d}$ and a $\mathtt{n}$-tuple $\mathtt{V}$ such that $\mathtt{B} = [\mathtt{d} \mid \mathtt{V}]$ and $\mathtt{d}(\mathtt{q})$ is defined for all $\mathtt{q} \in \mathbb{P}(\mathtt{V})$. By definition of equality for $\mathtt{n}$-arrays, $\mathtt{U} = \mathtt{V}$. By Claim 4.1 there exists a $\mathtt{n}$-tuple $\mathtt{T} = (\mathtt{t} : \mathtt{n})$ such that

$$\mathtt{U} = (\mathtt{t} : \mathtt{n}) = \mathtt{V}.$$

For all $\mathtt{t} \in \mathbb{P}(\mathtt{T})$ define the function $\mathtt{a}$ by $\mathtt{a}(\mathtt{t}) = \mathtt{c}(\mathtt{t})$. Since $\mathtt{B} = \mathtt{A}$, $\mathtt{d}(\mathtt{t}) = \mathtt{c}(\mathtt{t}) = \mathtt{a}(\mathtt{t})$ for all $\mathtt{t} \in \mathbb{P}(\mathtt{T})$. By the definition of equality for $\mathtt{n}$-arrays, $\mathtt{A} = [\mathtt{a} \mid \mathtt{T}]$ and $\mathtt{B} = [\mathtt{a} \mid \mathtt{T}]$. By transitivity, the stated result holds. ■

To show that a tuple function $\mathcal{F}_{\mathtt{S}}^{\mathtt{p}} \to \mathcal{F}_{\mathtt{T}}^{\mathtt{q}}$ is well-defined, it is enough to check that an arbitrary component in the result is well-defined, provided the length of the result is a constant function of the length of the argument.

LEMMA 4.1 Let $\mathtt{f}$ be any partial function

$$\mathtt{f} : \mathcal{F}_{\mathtt{S}}^{\mathtt{p}} \to \mathcal{F}_{\mathtt{T}}^{\mathtt{q}}$$

such that $\mathtt{f}(\mathtt{u} : \mathtt{n}) = (\mathtt{v} : \varsigma(\mathtt{n}))$. Then $\mathtt{f}$ is well-defined provided an arbitrary component $\mathtt{v}_{\mathtt{k}}$ is well-defined.

PROOF Let $\mathtt{U}, \mathtt{V} \in \mathcal{F}_{\mathtt{S}}^{\mathtt{p}}$ with $\mathtt{U} = \mathtt{V}$. Suppose

$$\mathtt{f}(\mathtt{U}) = \left(\mathtt{v}^{\mathtt{U}} : \varsigma^{\mathtt{U}}(\mathtt{n})\right)$$

and

$$\mathtt{f}(\mathtt{V}) = \left(\mathtt{v}^{\mathtt{V}} : \varsigma^{\mathtt{V}}(\mathtt{n})\right).$$

Assume that

$$\varsigma^{\mathtt{U}}(\mathtt{n}) = \varsigma^{\mathtt{V}}(\mathtt{n}) = \varsigma(\mathtt{n})$$

and for arbitrary $\mathtt{k} \in \iota\, \varsigma(\mathtt{n})$

$$\mathtt{v}_{\mathtt{k}}^{\mathtt{U}} = \mathtt{v}_{\mathtt{k}}^{\mathtt{V}}.$$

Then by the definition of equality for tuples,

$$\left(\mathtt{v}^{\mathtt{U}} : \varsigma^{\mathtt{U}}(\mathtt{n})\right) = \left(\mathtt{v}^{\mathtt{V}} : \varsigma^{\mathtt{V}}(\mathtt{n})\right).$$

By transitivity,

$$\mathtt{f}(\mathtt{U}) = \mathtt{f}(\mathtt{V}),$$

which confirms the statement of the lemma. ■

Any tuple function $\mathbb{Z} \to \mathbb{Z}$ is well-defined provided the length of the resulting tuple is a constant function of the length of the argument and an arbitrary component in the result is a linear combination of the components of the argument.

CLAIM 4.3 If $\mathtt{f}$ is a tuple function

$$\mathtt{f} : \mathbb{Z} \to \mathbb{Z}$$

such that

$$\mathtt{f}(\mathtt{u} : \mathtt{n}) = (\ell : \varsigma(\mathtt{n}))$$

and ℓ is a linear function satisfying

$$\ell_{\mathtt{k}} = \sum_{\mathtt{i}=1}^{\mathtt{n}} (\mathtt{a}_{\mathtt{i}}^{\mathtt{k}} \mathtt{u}_{\mathtt{i}} + \mathtt{b}_{\mathtt{i}}^{\mathtt{k}})$$

for some constants $\mathtt{a}_{\mathtt{i}}^{\mathtt{k}}$ and $\mathtt{b}_{\mathtt{i}}^{\mathtt{k}}$ then $\mathtt{f}$ is well-defined.

PROOF If ℓ is such a linear function, then it is well-defined; consequently, $\mathtt{f}$ is well-defined by Lemma 4.1. ■

To show that a partial function $\mathcal{A}^* \to \mathcal{A}^*$ on arrays is well-defined, it suffices to check that the empty array is mapped to a constant value and the following holds for the resulting array: the length of the tuple is a natural number, an arbitrary component of the tuple is well-defined, the array function is well-defined for an arbitrary location.

CLAIM 4.4 Let $\mathtt{f}$ be any partial function

$$\mathcal{A}^* \to \mathcal{A}^*.$$

Then $\mathtt{f}$ is well-defined provided the following conditions are met:

- If $\mathtt{f}\square$ is defined then $\mathtt{f}\square$ is a constant value;
- If $\mathtt{f}[\mathtt{a} \mid \mathtt{T}] = [\alpha \mid (\mathtt{s} : \mathtt{m})]$ then $\mathtt{m} \in \mathbb{N}$, $\mathtt{s}_{\mathtt{k}} \in \mathbb{N}$ is well-defined for arbitrary $\mathtt{k}$ and $\alpha(\mathtt{i} : \mathtt{m})$ is well-defined for arbitrary $(\mathtt{i} : \mathtt{m}) \in \mathbb{P}(\mathtt{s} : \mathtt{m})$.

PROOF Assume the conditions of the claim are satisfied. Let $\mathtt{A}, \mathtt{B} \in \mathcal{A}^*$ with $\mathtt{A} = \mathtt{B}$. If $\mathtt{A} = \square$ then $\mathtt{B} = \square$. If $\mathtt{f}\square$ is a constant value then

$$\mathtt{f}(\mathtt{A}) = \mathtt{f}\square = \mathtt{C} = \mathtt{f}\square = \mathtt{f}(\mathtt{B}).$$

If $\mathtt{m} \in \mathbb{N}$ and $\mathtt{s}_\mathtt{k} \in \mathbb{N}$ is well-defined for arbitrary $\mathtt{k}$, then

$$\mathtt{s} : \iota\, \mathtt{m} \to \mathbb{N}$$

is a sequence and $(\mathtt{s} : \mathtt{m}) \in \mathbb{N}^{\mathtt{m}}$; whence, $(\mathtt{s} : \mathtt{m})$ is a valid $\mathtt{m}$-tuple for $\mathbb{N}$. If $\alpha(\mathtt{i} : \mathtt{m})$ is well-defined for arbitrary $(\mathtt{i} : \mathtt{m}) \in \mathbb{P}(\mathtt{s} : \mathtt{m})$ then α is a valid array function. Thus, if the conditions of the claim are met then

$$[\alpha \,|\, (\mathtt{s} : \mathtt{m})]$$

is well-defined. Assume $\mathtt{A} \neq \square$. Invoking Lemma 4.2, write $\mathtt{A} = [\mathtt{a} \,|\, \mathtt{T}] = \mathtt{B}$. Suppose

$$\mathtt{f}[\mathtt{a} \,|\, \mathtt{T}] = [\alpha \,|\, (\mathtt{s} : \mathtt{m})]\,.$$

Then

$$\mathtt{f}(\mathtt{A}) = \mathtt{f}[\mathtt{a} \,|\, \mathtt{T}] = [\alpha \,|\, (\mathtt{s} : \mathtt{m})] = \mathtt{f}[\mathtt{a} \,|\, \mathtt{T}] = \mathtt{f}(\mathtt{B})\,. \blacksquare$$

If a function on arrays depends only on the tuple of the argument then it is only required to check that the function on the shape tuple is well-defined, and the empty array is mapped to a constant value.

CLAIM 4.5 Let $\mathtt{f}$ be any partial function

$$\mathcal{A}^* \to \mathtt{S}$$

such that $\mathtt{f}[\mathtt{a} \,|\, \mathtt{T}] = \mathtt{g}(\mathtt{T})$. Then $\mathtt{f}$ is well-defined provided $\mathtt{g}$ is well-defined and whenever $\mathtt{f}\square$ is defined, it is a constant value.

PROOF In view of the proof of the preceding claim it suffices to consider $\mathtt{A} = [\mathtt{a} \,|\, \mathtt{T}] = \mathtt{B}$. If $\mathtt{g}$ is well-defined and $\mathtt{f}[\mathtt{a} \,|\, \mathtt{T}] = \mathtt{g}(\mathtt{T})$, then $\mathtt{f}(\mathtt{A}) = \mathtt{g}(\mathtt{T}) = \mathtt{f}(\mathtt{B})$. ■

LEMMA 4.2 The tau shift, a conditional $\mathtt{n}$-tuple and delete are well-defined. In addition, move right is injective and for all $\mathtt{p} \in \mathtt{S}$ and $\mathtt{n} \in \mathbb{N}$

(DR) $$\langle\!\rangle_\mathtt{k} \circ \sqsupset_\mathtt{k} \mathtt{p} = \mathtt{id}_{\mathcal{T}^{\mathtt{n}}_{\mathtt{S}}}.$$

PROOF (SKETCH) The fact that these functions are well-defined follows since components are linear functions. The function move right is injective since it leaves intact the argument's components even though some are moved to the right. (Any proof which is sketched in the text is proved in the Appendix.) ■

LEMMA 4.3 For any n-tuple $(t:n)$ for $\mathbb{Z}$ there exists another n-tuple

$$T = (t:n; t_k = 1)$$

such that for all $m \in \mathbb{Z}$

(TM) $$\tau_k(m)\,T = (t:n; t_k = m).$$

PROOF The n-tuple T is defined since $1 \in \mathbb{Z}$ and

$$\tau_k(m)\,T = (t:n; t_k = 1+m-1) = (t:n; t_k = m)$$

is defined since $m \in \mathbb{Z}$. ■

Mostly the tau identities are self-evident. Especially

$$\tau(n)\,1 = n$$

may be used without reference to TIN, as the tau shift was designed precisely so that $\tau_k(n)$ would always shift the kth component which is 1 to n.

LEMMA 4.4 (TAU IDENTITIES)

(TI) $$\tau_k(1) = id_{\mathbb{Z}^n} \quad \text{and} \quad [a\,|\,\tau_k(1)\,T] = [a\,|\,T] = [a \circ \tau_k(1)\,|\,T],$$

(TIP) $$\tau_k(p) \circ \tau_k(q) = \tau_k(p+q-1),$$

(TII) $$(\tau_k(q))^{-1} = \tau_k(2-q) \quad \text{and} \quad \tau_k(p) \circ (\tau_k(q))^{-1} = \tau_k(p+1-q),$$

(TIM) $$\tau_k(p) \circ \beth_k(1) = \beth_k(p) \quad \text{on } \mathbb{Z}^n \text{ with } k < n,$$

(TIN) $$\tau(n)\,1 = n.$$

PROOF (SKETCH) First verify TIP by direct calculation. ■

The definition of a concatenation operator reveals how two n-arrays may be stacked together. On the other hand, the definition implicitly prescribes how any n-array may be split apart.

LEMMA 4.5 (CONCATENATION IDENTITY)

For any n-array $[a\,|\,(t:n)]$ and for any m with $1 \le m < t_k$,

(CI) $$[a\,|\,(t:n)] = [a\,|\,\tau_k(m)\,T] \#_k [a \circ \tau_k(m+1)\,|\,\tau_k(t_k - m)\,T].$$

where $T = (t:n; t_k = 1)$.

PROOF (SKETCH) Show that the tuples (right side) have the proper shape for concatenation. Then verify equality by showing the resulting array (on left side) has the proper parts. ■

Any hyperplane may be specified using the tau function as stated next.

LEMMA 4.6 (HYPERPLANE RELABELING)

Let $A = [a\,|\,(t:n)]$ be any n-array with $n > 1$. Then A has exactly t_k hyperplanes for the k-axis and the jth hyperplane of A in field direction k is

(HR) $$[a \circ \tau_k(j)\,|\,(t:n; t_k = 1)].$$

In addition,

(HMR) $$[b|\ \beth_k(1)(t:n)] = \text{reax}_k[b \circ \beth_k(1)\,|\,(t:n)].$$

PROOF By definition, $[a \circ \tau_k(j)\,|\,(t:n; t_k = 1)]$ is the jth hyperplane for the k-axis since

$$(a \circ \tau_k(j))(i:n; i_k = 1) = a(i:n; i_k = 1 + j - 1) = a(i:n; i_k = j).$$

By definition of reax,

$$\text{reax}_k[b \circ \beth_k(1)\,|\,(t:n)] = [b \circ \beth_k(1) \circ \daleth_k\ |\ \beth_k(1)(t:n)].$$

By DR,

$$b \circ \beth_k(1) \circ \daleth_k(\beth_k(1)(i:n)) = b \circ \beth_k(1)(i:n).$$

By inspection of this last equation and the shape tuples, it is evident that

$$[b \circ \beth_k(1) \circ \daleth_k\ |\ \beth_k(1)(t:n)] = [b|\ \beth_k(1)(t:n)],$$

which completes the proof by transitivity. ■

In the Appendix, it is verified that all primitive operations are well-defined. It is not possible to verify the primitives are well-defined in some trivial way. This fact is not a failure of the methodology. In general, it is tedious to prove an operation is well-defined. Therefore, it is difficult to program using a general model as a proof should be supplied for every operation. A benefit of using algebra is that programming

templates can be furnished. Functions defined using these templates are guaranteed to be well-defined.

LEMMA 4.7 The function collapse is well-defined. Furthermore,

(RC) $$\text{reax}_k \circ \text{collapse}_k\ \mathtt{A} = \mathtt{A} = \text{collapse}_k \circ \text{reax}_k\, \mathtt{A}.$$

PROOF (SKETCH) The function is well-defined since $\sqsupset_k$ is well-defined. The stated equalities hold by direct calculation (on the left side $\mathtt{A}$ must be a hyperplane and on the right side $\mathtt{k} \in \iota\, \mathtt{n}+1$ where $\mathtt{A}$ is a $\mathtt{n}$-array). ■

4.2 ARRAY FORMS

Next, consider all of the possible ways to express arrays and operations on them via constructors. Can every nonempty $\mathtt{n}$-array be expressed using only constructors? It suffices to consider all possible $\mathtt{n}$-tuples since every nonempty array is defined via a $\mathtt{n}$-tuple and an array function. Given any $\mathtt{n}$-array $\mathtt{A}$, there exists a $\mathtt{n}$-tuple $(\mathtt{t} : \mathtt{n})$ and an array function $\mathtt{a}$ such that

$$\mathtt{A} = [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})].$$

By definition, if $\mathtt{t}_k = 1$ for all $\mathtt{k}$ then $\mathtt{A} = \left[\mathtt{a} \mid \mathbb{C}^n\right]$ is scalar. Assume $\mathtt{t}_k > 1$ for some $\mathtt{k}$. If $\mathtt{n} = 1$ then $\mathtt{A}$ is a 1-array. Thus, the following criterion holds.

CRITERION 4.1 Every $\mathtt{n}$-array $\mathtt{A} = [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$ is scalar, a 1-array or a $\mathtt{n}$-array with $\mathtt{n} > 1$ and $\mathtt{t}_k > 1$ for some $\mathtt{k}$.

Consider each possibility in turn. First show that any scalar array can be expressed using exactly one constructor.

LEMMA 4.8 (SCALAR IDENTITY)
For any scalar array $\left[\mathtt{a} \mid \mathbb{C}^n\right]$ and for arbitrary $\mathtt{k} \in \iota\, \mathtt{n}$,

(SI) $$\left[\mathtt{a} \mid \mathbb{C}^n\right] = (\cdot)_n \left(\mathtt{a}\left(\mathbb{C}^n\right)\right) = \text{reax}_k (\cdot)_{n-1} \left(\mathtt{a}\left(\mathbb{C}^n\right)\right).$$

PROOF Let $\mathtt{A} = \left[\mathtt{a} \mid \mathbb{C}^n\right]$ be any scalar array. By definition,

$$\mathtt{B} = (\cdot)_n \left(\mathtt{a}\left(\mathbb{C}^n\right)\right) = \left[\mathtt{b} \mid \mathbb{C}^n\right],$$

where $\mathtt{b}\left(\mathbb{C}^n\right) = \mathtt{a}\left(\mathbb{C}^n\right)$. The equality $\left[\mathtt{a} \mid \mathbb{C}^n\right] = \left[\mathtt{b} \mid \mathbb{C}^n\right]$ holds since both scalar arrays have the same $\mathtt{n}$-tuple and the same element.

By definition,

$$(\cdot)_{n-1}\left(a\left(\mathbb{C}^n\right)\right) = \left[d \,|\, \mathbb{C}^{n-1}\right],$$

where $d\left(\mathbb{C}^{n-1}\right) = a\left(\mathbb{C}^n\right)$. By definition,

$$\text{reax}_k\left[d \,|\, \mathbb{C}^{n-1}\right] = \left[d \circ \text{◊}_k \mid \sqsupset_k(1)\,\mathbb{C}^{n-1}\right].$$

By DR,

$$d \circ \text{◊}_k \circ \sqsupset_k(1)\left(\mathbb{C}^{n-1}\right) = d\left(\mathbb{C}^{n-1}\right) = a\left(\mathbb{C}^n\right).$$

By inspection, the stated identity holds. ■

Second, examine 1-arrays. Every 1-array that is not a scalar is a concatenation of singletons.

LEMMA 4.9 Every 1-array either admits the form $(\cdot)(x)$ or is a concatenation of finitely many such singletons. Furthermore,

$$(\cdot)(a_1) = [a : 1]$$

for all valid array functions a.

PROOF By definition of $(\cdot)$,

$$[a : 1] = (\cdot)(a_1),$$

which establishes the basis for induction. Suppose the lemma is true for any 1-array $A = [a : t]$ with $1 \leq t \leq n$ for some $n \geq 1$. Put $t = n+1$. Invoke CI to write

$$A = [a : 1] +\!\!+ [a \circ \tau(2) : n].$$

By the inductive hypothesis, the lemma holds for $[a \circ \tau(2) : n]$. Since by definition $[a : 1]$ is a singleton, and $[a \circ \tau(2) : n]$ is a concatenation of singletons, it follows that

$$A = [a : 1] +\!\!+ [a \circ \tau(2) : n]$$

is a concatenation of singletons. As t_1 is finite, by the principle of mathematical induction, the lemma is true. ■

Consider next a finite sequence of hyperplanes which are stacked together via a concatenation operator.

LEMMA 4.10 Any n-array $A = [a \,|\, (t : n)]$ with $n > 1$ is a hyperplane for the k-axis or a concatenation of finitely many of them for every $k \in \iota\, n$.

PROOF Let A be as in the statement of the lemma. Put $k \in \iota\, n$. If $t_k = 1$ then, by

definition, $\mathtt{A}$ is a hyperplane for the $\mathtt{k}$-axis. Assume $[\mathtt{a} \mid (\mathtt{t}:\mathtt{n})]$ is a hyperplane for the $\mathtt{k}$-axis or a concatenation of finitely many of them whenever $\mathtt{t_k} \leq \mathtt{p}$ for some $\mathtt{p} \geq 1$ and for all array functions $\mathtt{a}$. Let $[\mathtt{a} \mid (\mathtt{t}:\mathtt{n}; \mathtt{t_k} = \mathtt{p}+1)]$ be any $\mathtt{n}$-array. Choose $\mathtt{m} \in \iota\,\mathtt{p}$, put $\mathtt{T} = (\mathtt{t}:\mathtt{n}; \mathtt{t_k} = 1)$ and invoke CI to write

$$[\mathtt{a} \mid (\mathtt{t}:\mathtt{n})] = [\mathtt{a} \mid \tau_\mathtt{k}(\mathtt{m})\,\mathtt{T}] \mathbin{+\!\!+_\mathtt{k}} [\mathtt{a} \circ \tau_\mathtt{k}(\mathtt{m}+1) \mid \tau_\mathtt{k}(\mathtt{t_k}-\mathtt{m})\,\mathtt{T}],$$

which is a concatenation of finitely many hyperplanes for the $\mathtt{k}$-axis since by the inductive hypothesis,

$$[\mathtt{a} \mid \tau_\mathtt{k}(\mathtt{m})\,\mathtt{T}] \qquad \text{and} \qquad [\mathtt{a} \circ \tau_\mathtt{k}(\mathtt{m}+1) \mid \tau_\mathtt{k}(\mathtt{t_k}-\mathtt{m})\,\mathtt{T}]$$

are such concatenations as

$$[\mathtt{a} \mid \tau_\mathtt{k}(\mathtt{m})\,\mathtt{T}] = [\mathtt{a} \mid (\mathtt{t}:\mathtt{n}; \mathtt{t_k} = \mathtt{m})] \text{ by TM,}$$

$$[\mathtt{a} \circ \tau_\mathtt{k}(\mathtt{m}+1) \mid \tau_\mathtt{k}(\mathtt{t_k}-\mathtt{m})\,\mathtt{T}] = [\mathtt{a} \circ \tau_\mathtt{k}(\mathtt{m}+1) \mid (\mathtt{t}:\mathtt{n}; \mathtt{t_k} = \mathtt{t_k}-\mathtt{m})] \text{ by TM,}$$

$\mathtt{m} \leq \mathtt{p}$ and $\mathtt{t_k} - \mathtt{m} \leq \mathtt{p}+1-\mathtt{m} \leq \mathtt{p}+1-1 = \mathtt{p}$. As $\mathtt{t_k}$ is finite, by the principle of mathematical induction, the proof is complete. ■

Next, show that any nonempty array can be expressed solely in terms of reax , $(\cdot)$ and $+\!\!+$ operators. This statement has already been confirmed for singletons and 1-arrays. It remains to consider a $\mathtt{n}$-array $\mathtt{A} = [\mathtt{a} \mid (\mathtt{t}:\mathtt{n})]$ with $\mathtt{n} > 1$ and $\mathtt{t_k} > 1$ for some $\mathtt{k}$. Use the Array Collapse Algorithm given next.

ARRAY COLLAPSE ALGORITHM: Let $\mathtt{A} \in \mathcal{A}^2$ be any nonscalar $\mathtt{m}$-array.

Perform the following steps in order:

1. If $\mathtt{A}$ is not a hyperplane for the $\mathtt{m}$-axis , rewrite $\mathtt{A}$ as a concatenation of hyperplanes for the $\mathtt{m}$-axis, which is possible by Lemma 4.10.

2. If $\mathtt{A}$ is a hyperplane for the $\mathtt{m}$-axis or a concatenation of them, replace each hyperplane for the $\mathtt{m}$-axis by $\mathrm{reax}_\mathtt{m}\,\mathtt{C}$ where $\mathtt{C}$ is a $(\mathtt{m}-1)$-array, which is possible by RC.

3. Suppose $\mathtt{A}$ is a concatenation involving an expression of the form $\mathrm{reax}_\mathtt{m}\,\mathtt{C}$. For every such expression, if $\mathtt{C}$ is scalar or a 1-array, replace $\mathtt{C}$ by a scalar function or a concatenation of singletons, invoking SI and Lemma 4.9; otherwise, $\mathtt{C}$ is a nonscalar $(\mathtt{m}-1)$-array $\mathtt{A} \in \mathcal{A}^2$ by Criterion 4.1 and apply the Array Collapse Algorithm to $\mathtt{C}$.

LEMMA 4.11 Any $\mathtt{n}$-array $\mathtt{A} = [\mathtt{a} \mid (\mathtt{t}:\mathtt{n})]$ with $\mathtt{n} > 1$ and $\mathtt{t_k} > 1$ for some $\mathtt{k}$ can be expressed in terms of only reax, scalar functions and $+\!\!+$ operators.

PROOF Let $\mathtt{A}$ be as in the statement of the lemma. Apply the Array Collapse Algorithm to $\mathtt{A}$. Each time the algorithm is executed, the number of axes is reduced by one. As there are only a finite number $\mathtt{n}$ of axes, eventually the algorithm terminates with $\mathtt{A}$ expressed as stated in the lemma. ■

What are arrays? They are combinations of constructors. What is needed are some practical rules that prescribe all of the possible forms of an object so that a programmer may specify an operation in terms of these forms.

Every nonscalar 1-array is a concatenation of a singleton and a 1-array.

LEMMA 4.12 Every 1-array admits the form $(\cdot)(\mathtt{x}) \# \mathtt{A}$ for some type variable $\mathtt{x}$ and some $\mathtt{A} \in \mathbb{A}^*$.

PROOF Let $[\mathtt{b} : \mathtt{n}]$ be any 1-array. Suppose $\mathtt{n} = 1$. Then

$$(\cdot)(\mathtt{b}_1) \# \square = (\cdot)(\mathtt{b}_1) = [\mathtt{b} : 1] \text{ by SI.}$$

For $\mathtt{n} > 1$,

$$\begin{aligned}[\mathtt{b} : \mathtt{n}] &= [\mathtt{b} : 1] \# [\mathtt{b} \circ \tau(2) : \mathtt{k}] \text{ by CI} \\ &= (\cdot)(\mathtt{b}_1) \# [\mathtt{b} \circ \tau(2) : \mathtt{k}] \text{ by SI.}\end{aligned}$$ ■

COROLLARY 4.13 Every 1-array admits the form $\mathtt{A} \# (\cdot)(\mathtt{x})$ for some type variable $\mathtt{x}$ and for some $\mathtt{A} \in \mathbb{A}^*$.

PROOF A symmetrical argument given in the preceding proof verifies the corollary. ■

COROLLARY 4.14 (LIST FORMS)

Every 1-array can be expressed in one of the **list form**

$$(\cdot)(\mathtt{x}) \quad \text{or} \quad \mathtt{C} \# \mathtt{A}$$

for some type variable $\mathtt{x}$ and for some $\mathtt{C}, \mathtt{A} \in \mathbb{A}$.

PROOF This statement holds by Lemma 4.12 since any 1-array has the form

$$(\cdot)(\mathtt{x}) \# \mathtt{A}$$

which by inspection corresponds to the list forms $(\cdot)(\mathtt{x})$ or $\mathtt{C} \# \mathtt{A}$ if $\mathtt{A} = \square$ or $\mathtt{A} \neq \square$, respectively. ■

LEMMA 4.15 Any n-array $A = [a \mid (t : n)]$ with $n > 1$ admits the form

$$(\text{reax}_k\, C) +\!\!+_k B$$

for some $C \in \mathcal{A}$, $B \in \mathcal{A}^*$ and for every $k \in \iota\, n$.

PROOF Let A be as in the statement of the lemma. If $t_k = 1$ then A is a hyperplane for the k-axis and

$$\begin{aligned} A &= \text{reax}_k \circ \text{collapse}_k\, A \quad \text{by RC} \\ &= \text{reax}_k\, (\text{collapse}_k\, A) +\!\!+_k \square, \end{aligned}$$

has the stated form. Assume $t_k > 1$. Taking $T = (t : n; t_k = 1)$,

$$\begin{aligned} A &= [a \mid (t : n)] \\ &= [a \mid \tau(t_k)\, T] \quad \text{by TM} \\ &= [a \mid \tau_k(1)\, T] +\!\!+_k [a \circ \tau_k(2) \mid \tau_k(t_k - 1)\, T] \quad \text{by CI} \\ &= [a \mid T] +\!\!+_k [a \circ \tau_k(2) \mid \tau_k(t_k - 1)\, T] \quad \text{by TI} \\ &= \text{reax}_k\, (\text{collapse}_k\, [a \mid T]) +\!\!+_k [a \circ \tau_k(2) \mid \tau_k(t_k - 1)\, T] \quad \text{by RC}, \end{aligned}$$

which matches the form given in the statement of the lemma. ■

COROLLARY 4.16 Any n-array $A = [a \mid (t : n)]$ with $n > 1$ admits the form

$$B +\!\!+_k (\text{reax}_k\, C)$$

for some $C \in \mathcal{A}$, $B \in \mathcal{A}^*$ and for every $k \in \iota\, n$.

PROOF Apply a symmetrical argument given in the preceding proof. ■

COROLLARY 4.17 (PLANAR FORMS)

For $n > 1$, every n-array can be expressed in one of the **planar forms**

$$\text{reax}_k\, C \quad \text{or} \quad A +\!\!+_k B,$$

for some $A, B, C \in \mathcal{A}$ and for arbitrary $k \in \iota\, n$.

PROOF Any n-array with $n > 1$ admits the planar form $(\text{reax}_k\, C) +\!\!+_k B$ by the previous lemma. This form corresponds to the planar forms $\text{reax}_k\, C$ or $A +\!\!+_k B$ with $A = \text{reax}_k\, C$ if $B = \square$ or $B \neq \square$, respectively. ■

THEOREM 4.18 (ARRAY FORMS)

Every array admits one of the forms

(i) $\square$,

(ii) $(\cdot)(x)$,

(iii) $(\text{reax}_k C)$ or

(iv) $C +\!\!+_k A$,

for some type variable x, for some $A, C \in \mathcal{A}$ and for $k \in \mathbb{N}$.

PROOF An array is either an empty array or a n-array for some $n \in \mathbb{N}$. By Criterion 4.1, every array has one of the following forms:

(a) $\square$,

(b) $[a \mid C^n]$,

(c) $[a : m]$,

(d) $[a \mid (t : n)]$ with $n > 1$ and $t_k > 1$ for some k.

By inspection, in case (a), form (i) holds. In case (b) by scalar identity SI (Lemma 4.8), form (ii) or (iii) holds. In case (c), in view of list forms, form (ii) or (iv) must hold. Lastly, in case (d), in view of planar forms, form (iii) or (iv) must hold. ■

COROLLARY 4.19 (ARRAY FORMS)

Every array admits one of the forms

(1) $\square$,

(2) $(\cdot)(x) +\!\!+ A$, or

(3) $(\text{reax}_k C) +\!\!+_k A$,

for some type variable x, for some $A \in \mathcal{A}^*$, $C \in \mathcal{A}$ and for all valid $k \in \mathbb{N}$.

PROOF This statement holds by Criterion 4.1, Lemmas 4.8, 4.12 and 4.15. ■

More explicit forms for arrays are needed.

LEMMA 4.20 (LIST CONCATENATION)

Any 1-array can always be expressed

(LC) $$[a : n] = [a \circ \tau(1) : 1] +\!\!+ \cdots +\!\!+ [a \circ \tau(n) : 1].$$

PROOF (SKETCH) Any 1-array is a singleton or a concatenation of finitely many of them (Lemma 4.9). ■

LEMMA 4.21 (HYPERPLANE CONCATENATION)

Setting

$$\mathtt{T} = (\mathtt{t} : \mathtt{n}; \mathtt{t}_\mathtt{k} = 1)$$

for every $\mathtt{k} \in \iota\, \mathtt{n}$,

(HC) $$[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] = [\mathtt{a} \circ \tau_\mathtt{k}(1) \mid \mathtt{T}] +\!\!+_\mathtt{k} \cdots +\!\!+_\mathtt{k} [\mathtt{a} \circ \tau_\mathtt{k}(\mathtt{t}_\mathtt{k}) \mid \mathtt{T}].$$

PROOF (SKETCH) A $\mathtt{n}$-array with $\mathtt{n} > 1$ is a hyperplane or a concatenation of them (Lemma 4.10). ■

So it has been established that $\mathtt{n}$-arrays possess a small number of forms. Yet a $\mathtt{n}$-array model has essentially one form. Why choose constructors over a general model? An answer is summarized in the following word.

4.3 HOMOMORPHISMS

Given an instance of an object, presumably any operation which can be defined on that object can be specified using a general model. The problem with general forms such as a $\mathtt{n}$-array model is that even though they are quite versatile, it is difficult to describe classes of operations using such forms in a useful algebraic fashion. In principle, it is always possible to write a program for any operation using a general model. It may even be possible to draw nice pictures to describe what happens for each *particular* operation. Working with such a general model, however, the programmer typically must start from square one and develop a program for every operation.

Using constructors it is possible to say much more about programming. Before the programmer even gets started, it is possible to deliver some programming templates which can be used to actually write programs. After the programmer has implemented these templates as procedures, they may be called upon during software development.

As the constructors build up data aggregates, they are natural candidates for homomorphisms. Remember, a homomorphism distributes across a binary constructor. There is only one binary constructor, namely concatenation. First, consider 1-arrays and second, $\mathtt{n}$-arrays with $\mathtt{n} > 1$.

4.3.1 LIST HOMOMORPHISMS

As every 1-array can be expressed using one of the list forms given earlier, it is possible to classify homomorphisms for each such form, which is done next. Start with the list form

$$\mathtt{A} +\!\!+ (\cdot)(\mathtt{x})$$

which in a natural way corresponds to performing an (possibly nonassociative) operation from left to right. It is not necessary to insist that a given binary operator possesses a (fictitious) identity element. For this reason, the form

$$(\cdot)(\mathtt{x})$$

is used essentially twice so that it is not necessary to employ an empty array. Otherwise, a couple definitions should be formulated, instead of one (one in case the given operator is assigned an identity element and another definition in the alternate case).

DEFINITION 4.1 (**left hom**)

An operation h is a left hom for an array A if there exists a binary operator $\oplus$ such that

$$\mathtt{h}(\mathtt{B} +\!\!+ (\cdot)(\mathtt{x})) = \mathtt{h}(\mathtt{B}) \oplus \mathtt{h}(\cdot)(\mathtt{x}) \tag{4.1}$$

whenever $\mathtt{B} +\!\!+ (\cdot)(\mathtt{x})$ is a subarray of A with $\mathtt{B} \neq \square$, and an unary function u such that

$$\mathtt{h}(\cdot)(\mathtt{x}) = \mathtt{u}(\mathtt{x})$$

whenever x is an element of A. If $\mathtt{h}(\square)$ is defined, it equals the identity element for $\oplus$.

Next, consider the list form

$$(\cdot)(\mathtt{x}) +\!\!+ \mathtt{A}$$

which is a suitable form whenever performing an (possibly nonassociative) operation from right to left.

DEFINITION 4.2 (**right hom**)

An operation h is a right hom for an array A if there exists a binary operator $\oplus$ such that

$$\mathtt{h}((\cdot)(\mathtt{x}) +\!\!+ \mathtt{B}) = \mathtt{h}(\cdot)(\mathtt{x}) \oplus \mathtt{h}(\mathtt{B}), \tag{4.2}$$

whenever $(\cdot)(\mathtt{x}) +\!\!+ \mathtt{B}$ is a subarray of A with $\mathtt{B} \neq \square$, and an unary function u such that

$$\mathtt{h}(\cdot)(\mathtt{x}) = \mathtt{u}(\mathtt{x})$$

whenever x is an element of A. If $h(\square)$ is defined, it equals the identity element for $\oplus$.

Third, consider the list form

$$A +\!\!+ B$$

which is the only remaining list form. This form is applicable whenever performing an associative operation. In this case, assume concatenation is associative, which is proved later.

DEFINITION 4.3 (**catamorphism**)

An operation h is a catamorphism for an array A if there exists a binary operator $\oplus$ such that

$$h(B +\!\!+ C) = h(B) \oplus h(C), \tag{4.3}$$

whenever $B +\!\!+ C$ is a subarray of A with $B, C \neq \square$, and an unary function u such that

$$h(\cdot)(x) = u(x)$$

whenever x is an element of A. If $h(\square)$ is defined, it equals the identity element for $\oplus$.

It turns out that these homomorphisms are uniquely determined by the specified binary and unary operators, which is known as the **unique extension property**.

THEOREM 4.22 A left hom, a right hom and a catamorphism are uniquely determined by the binary operator $\oplus$ and the unary function u.

PROOF (SKETCH) Evidently any two homomorphisms with the same binary operator $\oplus$ and unary function u agree on singletons. By induction, they must agree on any concatenation of singletons. ■

A few needed operators are listed next.

List Homomorphism	Function
left hom with binary operator $\oplus$ and unary function u	$lom(\oplus, u)$
right hom with binary operator $\oplus$ and unary function u	$rom(\oplus, u)$
catamorphism with binary operator $\oplus$ and unary function u	$(\oplus, u)$

THEOREM 4.23 The following identities hold:

$$\mathtt{lom}(\oplus,\mathtt{u})[\mathtt{a}:\mathtt{n}] = (\cdots((\mathtt{u}(\mathtt{a}_1)\oplus\mathtt{u}(\mathtt{a}_2))\oplus\mathtt{u}(\mathtt{a}_3))\cdots)\oplus\mathtt{u}(\mathtt{a}_n),$$

$$\mathtt{rom}(\oplus,\mathtt{u})[\mathtt{a}:\mathtt{n}] = \mathtt{u}(\mathtt{a}_1)\oplus(\mathtt{u}(\mathtt{a}_2)\oplus(\cdots(\mathtt{u}(\mathtt{a}_{n-1})\oplus\mathtt{u}(\mathtt{a}_n))\cdots)),$$

and

$$(\oplus,\mathtt{u})[\mathtt{a}:\mathtt{n}] = \mathtt{u}(\mathtt{a}_1)\oplus\mathtt{u}(\mathtt{a}_2)\oplus\cdots\oplus\mathtt{u}(\mathtt{a}_n).$$

PROOF (SKETCH) Replace the constructors by the operators. ■

Recall, the composition of two homomorphisms is a homomorphism. The promotion lemma states this fact more precisely.

LEMMA 4.24 (PROMOTION LEMMA)
Let $\mathtt{h}$ be any morphism such that

$$\mathtt{h}(\mathtt{a}\oplus\mathtt{b}) = \mathtt{h}(\mathtt{a})\otimes\mathtt{h}(\mathtt{b}).$$

Then

$$\mathtt{h}\circ\mathtt{lom}(\oplus,\mathtt{u}) = \mathtt{lom}(\otimes,\mathtt{h}\circ\mathtt{u}),$$

$$\mathtt{h}\circ\mathtt{rom}(\oplus,\mathtt{u}) = \mathtt{rom}(\otimes,\mathtt{h}\circ\mathtt{u}),$$

and

$$\mathtt{h}\circ(\oplus,\mathtt{u}) = (\otimes,\mathtt{h}\circ\mathtt{u}).$$

PROOF (SKETCH) This equations follow by direct calculation and uniqueness. ■

In general, if a binary operation is not associative, then the order of "folding" or grouping must be defined. Consider a homomorphism which is a folding of the elements without modifying the elements in any way.

DEFINITION 4.4 If the unary function $\mathtt{u}$ equals the identity function

$$\mathtt{id}(\mathtt{x}) = \mathtt{x}$$

then a left hom, right hom, and catamorphism is called a **left fold**, **right fold**, and **reduce** operation, respectively.

Another special class of homomorphisms is one for which an operation modifies all of the elements without changing the data organization. The elements are not folded at all. These morphisms are called maps.

DEFINITION 4.5 If the binary operator is concatenation and the unary operator u obeys

$$u(x) = (\cdot)(f(x))$$

for some function f, then a left hom, a right hom, or a catamorphism is called a `map`.

For folding homomorphisms, it suffices to specify only the binary operation as the unary function is known. For maps, it suffices to specify only the special function which is applied to the elements as the binary operator is known. The following conventions are adopted:

List Homomorphism	Function
left fold $= \mathtt{lom}(\oplus, \mathtt{id})$	$\twoheadrightarrow \oplus$
right fold $= \mathtt{rom}(\oplus, \mathtt{id})$	$\twoheadleftarrow \oplus$
reduce $= (\oplus, \mathtt{id})$	$/ \oplus$
map $f = (+\!\!+, f)$	$* f$

THEOREM 4.25 For any binary operator $\oplus$,

$$(\twoheadrightarrow \oplus)[a:n] = ((a_1 \oplus a_2) \oplus a_3) \oplus \cdots \oplus a_n, \tag{4.4}$$

$$(\twoheadleftarrow \oplus)[a:n] = a_1 \oplus (a_2 \oplus \cdots \oplus (a_{n-1} \oplus a_n)), \tag{4.5}$$

and for any associative binary operator $\oplus$,

$$(/\oplus)[a:n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n. \tag{4.6}$$

PROOF Plug $u = \mathtt{id}$ in Theorem 4.23. ■

THEOREM 4.26 For any binary operator $\oplus$,

$$(\twoheadrightarrow \oplus)[a:n] = \smile \oplus [a:n]\ 1,$$

$$(\twoheadleftarrow \oplus)[a:n] = \frown \oplus [a:n]\ 1,$$

and for any associative binary operator $\oplus$,

$$(/\oplus)[a:n] = \asymp \oplus [a:n]\ 1.$$

PROOF These identities follow by inspection since

$$\tau(k)\ 1 = k$$

implies $a \circ \tau(k)\ 1 = a_k$. ■

EXAMPLE 4.1 Define $a \downarrow b = \min\{a, b\}$ and $a \uparrow b = \max\{a, b\}$.

Reduce Operation	Equivalent Expression
$/+[x:n]$	$\sum_{i=1}^{n} x_i$
$/\times[x:n]$	$\prod_{i=1}^{n} x_i$

Reduce Operation	Equivalent Expression
$/\downarrow[x:n]$	$\min_{1\le i\le n}\{x_i\}$
$/\uparrow[x:n]$	$\max_{1\le i\le n}\{x_i\}$

EXAMPLE 4.2 Let $a \oplus b = 2a + b$. Then

$$(\rightarrowtail \oplus [id:3]) = ((1 \oplus 2) \oplus 3) = 4 \oplus 3 = 11,$$

and

$$(\leftarrowtail \oplus)[id:3] = 1 \oplus (2 \oplus 3) = 1 \oplus 7 = 9.$$

EXAMPLE 4.3 Since $\#_k$ is an associative operator,

$$(/\#_k)[a:n] = a_1 \#_k a_2 \#_k \cdots \#_k a_n.$$

THEOREM 4.27 The definitions of $*f$ in Definition 3.28 and in Definition 4.5 are consistent. Furthermore,

$$(*f)[a:n] = [f \circ a \circ \tau(1):1] \# \cdots \# [f \circ a \circ \tau(n):1].$$

PROOF Calculate

$$\begin{aligned}
(*f)[a:n] &= (\#, f)[a:n] && \text{by Definition 4.5}\\
&= (\cdot)(f(a_1)) \# \cdots \# (\cdot)(f(a_n)) && \text{by Theorem 4.23}\\
&= [f \circ a \circ \tau(1):1] \# \cdots \# [f \circ a \circ \tau(n):1] && \text{by Lemma 4.9}\\
&= [(f \circ a) \circ \tau(1):1] \# \cdots \# [(f \circ a) \circ \tau(n):1] && \text{as } \circ \text{ is associative}\\
&= [f \circ a:n] && \text{by LC}\\
&= (*f)[a:n] && \text{by Definition 3.28.}\ \blacksquare
\end{aligned}$$

THEOREM 4.28 Let $[a|(t:n)]$ be any n-array with $t_k > 1$ for which $*f$ may be applied. Set

$$T = (t:n; t_k = 1).$$

Pick $\mathtt{m} \in \iota\ (\mathtt{t}_k - 1)$. Then

$$\begin{aligned}&(*\mathtt{f})([\mathtt{a} \mid \tau_k(\mathtt{m})\,\mathtt{T}] +\!\!+_k [\mathtt{a} \circ \tau_k(\mathtt{m}+1) \mid \tau_k(\mathtt{t}_k - \mathtt{m})\,\mathtt{T}])\\ =&(*\mathtt{f})[\mathtt{a} \mid \tau_k(\mathtt{m})\,\mathtt{T}] +\!\!+_k (*\mathtt{f})[\mathtt{a} \circ \tau_k(\mathtt{m}+1) \mid \tau_k(\mathtt{t}_k - \mathtt{m})\,\mathtt{T}].\end{aligned}$$

PROOF Let $[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$, $\mathtt{T}$ and $\mathtt{m}$ be as in the statement of the theorem. Compute

$$\begin{aligned}&(*\mathtt{f})([\mathtt{a} \mid \tau_k(\mathtt{m})\,\mathtt{T}] +\!\!+_k [\mathtt{a} \circ \tau_k(\mathtt{m}+1) \mid \tau_k(\mathtt{t}_k - \mathtt{m})\,\mathtt{T}])\\ =&(*\mathtt{f})[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] \text{ by CI}\\ =&[\mathtt{f} \circ \mathtt{a} \mid (\mathtt{t} : \mathtt{n})] \text{ by Definition 3.28}\\ =&[(\mathtt{f} \circ \mathtt{a}) \mid \tau_k(\mathtt{m})\,\mathtt{T}] +\!\!+_k [(\mathtt{f} \circ \mathtt{a}) \circ \tau_k(\mathtt{m}+1) \mid \tau_k(\mathtt{t}_k - \mathtt{m})\,\mathtt{T}] \text{ by CI}\\ =&(*\mathtt{f})[\mathtt{a} \mid \tau_k(\mathtt{m})\,\mathtt{T}] +\!\!+_k (*\mathtt{f})[\mathtt{a} \circ \tau_k(\mathtt{m}+1) \mid \tau_k(\mathtt{t}_k - \mathtt{m})\,\mathtt{T}]\\ &\text{by Definition 3.28.} \blacksquare\end{aligned}$$

LEMMA 4.29 (MAP DISTRIBUTIVITY)

A `map` distributes for `1`-arrays:

$$*(\mathtt{f} \circ \mathtt{g}) = (*\mathtt{f}) \circ (*\mathtt{g}).$$

PROOF Calculate

$$\begin{aligned}(*\mathtt{f}) \circ (*\mathtt{g}) =&(+\!\!+, \mathtt{f}) \circ (+\!\!+, \mathtt{g}) \text{ by definition}\\ =&(+\!\!+, \mathtt{f} \circ \mathtt{g}) \text{ by the promotion lemma}\\ =&*(\mathtt{f} \circ \mathtt{g}) \text{ by definition.} \blacksquare\end{aligned}$$

Next the homomorphism lemma from the theory of lists is recast for arrays.

LEMMA 4.30 (HOMOMORPHISM LEMMA)

A morphism is a left hom, right hom or catamorphism if and only if the morphism is a composition of a left fold, right fold or reduce operation, respectively, with a `map`.

PROOF Rewrite

$$\begin{aligned}(\rightsquigarrow \oplus) \circ (*\mathtt{u}) &= (\rightsquigarrow \oplus) \circ \mathtt{lom}(+\!\!+, \mathtt{u}) \text{ by definition}\\ &= \mathtt{lom}(\oplus, \mathtt{u}) \text{ by the promotion lemma,}\\ (\leftsquigarrow \oplus) \circ (*\mathtt{u}) &= (\leftsquigarrow \oplus) \circ \mathtt{rom}(+\!\!+, \mathtt{u}) \text{ by definition}\\ &= \mathtt{rom}(\oplus, \mathtt{u}) \text{ by the promotion lemma,}\end{aligned}$$

and lastly,

$$(\diagup\oplus)\circ(*u) = (\diagup\oplus)\circ(\perp\!\!\!\perp, u) \text{ by definition}$$
$$= (\oplus, u) \text{ by the promotion lemma. } \blacksquare$$

THEOREM 4.31 (PROMOTABILITY THEOREM)
Let h be any morphism such that

$$h(a \oplus b) = h(a) \otimes h(b).$$

Then

$$h \circ lom(\oplus, u) = (\rightarrowtail \otimes) \circ (*(h \circ u)),$$
$$h \circ rom(\oplus, u) = (\leftarrowtail \otimes) \circ (*(h \circ u)),$$

and

$$h \circ (\oplus, u) = (\diagup \otimes) \circ (*(h \circ u)).$$

PROOF By direct calculation,

$$h \circ lom(\oplus, u) = lom(\otimes, h \circ u) \text{ by the promotion lemma}$$
$$= (\rightarrowtail \otimes) \circ (*(h \circ u)) \text{ by the homomorphism lemma,}$$
$$h \circ rom(\oplus, u) = rom(\otimes, h \circ u) \text{ by the promotion lemma}$$
$$= (\leftarrowtail \otimes) \circ (*(h \circ u)) \text{ by the homomorphism lemma,}$$

and lastly,

$$h \circ (\oplus, u) = (\otimes, h \circ u) \text{ by the promotion lemma}$$
$$= (\diagup \otimes) \circ (*(h \circ u)) \text{ by the homomorphism lemma. } \blacksquare$$

These identities are versatile.

EXAMPLE 4.4 Taking $u = id$,

$$h \circ (\rightarrowtail \oplus) = (\rightarrowtail \otimes) \circ (*h),$$
$$h \circ (\leftarrowtail \oplus) = (\leftarrowtail \otimes) \circ (*h),$$
$$h \circ (\diagup \oplus) = (\diagup \otimes) \circ (*h).$$

EXAMPLE 4.5 Taking $h = *f$ and $\oplus = \perp\!\!\!\perp$,

$$*f \circ (\diagup \perp\!\!\!\perp_k) = (\diagup \perp\!\!\!\perp_k) \circ (**f)$$

since $*\mathtt{f}\,(\mathtt{A} +\!\!+_k \mathtt{B}) = (*\mathtt{f}\,\mathtt{A}) +\!\!+_k (*\mathtt{f}\,\mathtt{B})$. This example gives an important promotion law for `map` on arrays.

EXAMPLE 4.6 Letting $\mathtt{h} = /\odot$ and $\oplus = +\!\!+$,

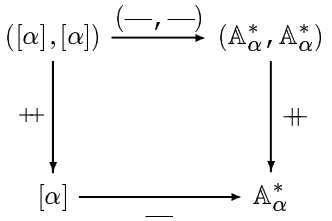
$$(/\odot) \circ (/+\!\!+) = (/\odot) \circ (*(/\odot))$$

since $/\odot\ (\mathtt{A} +\!\!+ \mathtt{B}) = (/\odot \mathtt{A}) \odot (/\odot \mathtt{B})$.

EXAMPLE 4.7 If $\mathtt{h}(\mathtt{A} +\!\!+ \mathtt{B}) = \mathtt{h}(\mathtt{A}) \oplus \mathtt{h}(\mathtt{B})$ then

$$\mathtt{h} \circ (*\mathtt{f}) = (/\oplus) \circ *(\mathtt{h} \circ \mathtt{f}).$$

List homomorphisms have been added to the theory. This fact should not be surprising as concatenation was adopted as the binary constructor. There is a functor taking $[\alpha]$ to $\mathbb{A}^*_\alpha$ and $+\!\!+$ to $+\!\!+$ where the homomorphism $\overline{}$ is given by $\overline{[\,]} = \Box$ and $\overline{[\mathtt{t}_1, \ldots, \mathtt{t}_n]} = [\mathtt{a} : \mathtt{n}]$ with $\mathtt{a}_\mathtt{i} = \mathtt{t}_\mathtt{i}$. It is not an isomorphism as `1`-arrays are not dependent on any (homogeneous) type (or set). A commutative diagram establishes the equivalences.

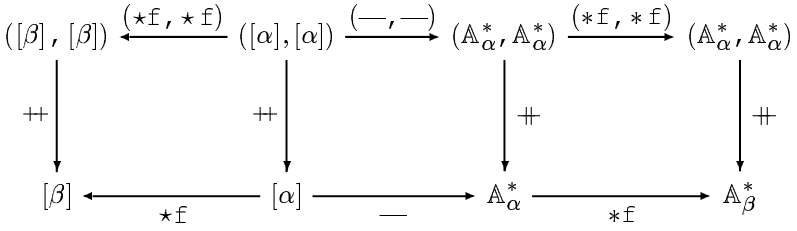


i.e.,

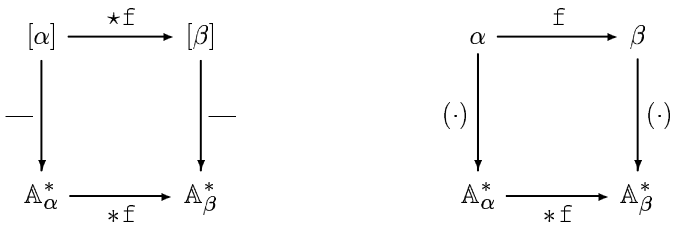
$$\overline{\mathtt{A} +\!\!+ \mathtt{B}} = \overline{\mathtt{A}} +\!\!+ \overline{\mathtt{B}},$$

$$\overline{} \circ +\!\!+ = +\!\!+ \circ (\overline{}, \overline{}).$$

Another commutative diagram is presented to show that maps are homomorphisms for both data types.



Concatenation is a natural transformation since it satisfies a **naturality condition**, which means the diagram commutes for any function $\mathtt{f} : \alpha \to \beta$. By inspection, $\overline{}$ and $(\cdot)$ are natural transformations since the following diagrams commute for all functions $\mathtt{f} : \alpha \to \beta$.



Next, additional array homomorphisms are classified.

4.3.2 EXTENDED LIST HOMOMORPHISMS

Classify three kinds of homomorphisms for n-arrays with $n > 1$, just as for 1-arrays, using the planar forms presented earlier. First, consider the planar form

$$B \#_k (\text{reax}_k C)$$

which in a natural way corresponds to performing a (possibly nonassociative) operation from left to right, axially in some field direction. Use similar names as the operations do not appear to be different, although the objects do. The elements are now hyperplanes, i.e., treat entire hyperplanes as single units. After folding a 1-array, the result need not be a collection of elements and an array data structure may not be needed to store the result. Folding hyperplanes yields a collection which is a hyperplane.

DEFINITION 4.6 (**left array hom**)

An operation h is a left array hom for a n-array A with $n > 1$ if for some field direction $k \in \iota\, n$ and some binary operator $\oplus$,

$$h(B \#_k (\text{reax}_k C)) = h(B) \oplus h(\text{reax}_k C), \tag{4.7}$$

whenever $B \#_k (\text{reax}_k C)$ is a subarray of A with $B \neq \square$, and for some unary function u,

$$h(\text{reax}_k C) = u(\text{reax}_k C),$$

whenever $\text{reax}_k C$ is a subarray of A. If $h(\square)$ is defined, it must be equal to the identity element for $\oplus$.

Second, consider another planar form

$$(\text{reax}_k B) \#_k C$$

which is a suitable form whenever performing an (possibly nonassociative) operation from right to left, axially in some field direction.

DEFINITION 4.7 (**right array hom**)

An operation h is a right array hom for a n-array A with $n > 1$ if for some field direction $k \in \iota\, n$ and some binary operator $\oplus$,

$$h((\text{reax}_k\, B) +\!\!+_k C) = h(\text{reax}_k\, B) \oplus h(C), \tag{4.8}$$

whenever $(\text{reax}_k\, B) +\!\!+_k C$ is a subarray of A with $C \neq \square$, and for some unary function u,

$$h(\text{reax}_k\, C) = u(\text{reax}_k\, C),$$

whenever $\text{reax}_k\, C$ is a subarray of A. If $h(\square)$ is defined, it must be equal to the identity element for $\oplus$.

Third, consider the planar form

$$A +\!\!+_k B$$

which is a suitable form whenever performing an associative operation axially in some field direction. In this case, assume concatenation is associative.

DEFINITION 4.8 (**array catamorphism**)

An operation h is an array catamorphism for a n-array A with $n > 1$ if for some field direction $k \in \iota\, n$ and some binary operator $\oplus$,

$$h(B +\!\!+_k C) = h(B) \oplus h(C), \tag{4.9}$$

whenever $B +\!\!+_k C$ is a subarray of A with $B, C \neq \square$, and for some unary function u,

$$h(\text{reax}_k\, C) = u(\text{reax}_k\, C),$$

whenever $\text{reax}_k\, C$ is a subarray of A. If $h(\square)$ is defined, it must be equal to the identity element for $\oplus$.

An **extended list homomorphism** is a left array hom, a right array hom or an array catamorphism. As list and extended list homomorphisms are alike, it is reasonable to suspect the theorems are similar.

THEOREM 4.32 The unique extension property extends to extended list homomorphisms.

PROOF (SKETCH) Evidently, any two extended list homomorphisms with the same binary operator $\oplus$ and unary function u agree on any hyperplane. By induction, they must agree on any concatenation of hyperplanes. ■

The same notation for list homomorphisms is used, except the field direction is attached for extended list homomorphisms:

Homomorphism	Function
left array hom	$\mathtt{lom}_k(\oplus,u)$
right array hom	$\mathtt{rom}_k(\oplus,u)$
array catamorphism	$(\oplus,u)_k$

THEOREM 4.33 The following identities hold:

$$\mathtt{lom}_k(\oplus,u)[a\,|\,(t:n)] = (\cdots(u[a\circ\tau_k(1)\,|\,T]\oplus u[a\circ\tau_k(2)\,|\,T])\cdots)\oplus u[a\circ\tau_k(t_k)\,|\,T],$$

$$\mathtt{rom}_k(\oplus,u)[a\,|\,(t:n)] = u[a\circ\tau_k(1)\,|\,T]\oplus(\cdots(u[a\circ\tau_k(t_k-1)\,|\,T]\oplus u[a\circ\tau_k(t_k)\,|\,T])\cdots),$$

and

$$(\oplus,u)_k[a\,|\,(t:n)] = u[a\circ\tau_k(1)\,|\,T]\oplus\cdots\oplus u[a\circ\tau_k(t_k)\,|\,T],$$

where using TM

$$T = (t:n;t_k=1).$$

PROOF (SKETCH) Replace the constructors by the operators. ■

The promotion lemma needs to be adjusted slightly to accommodate hyperplanes.

LEMMA 4.34 (HYPERPLANE PROMOTION LEMMA)
If h is any homomorphism such that

$$h(a\oplus b) = h(a)\otimes h(b),$$

then

$$h\circ\mathtt{lom}_k(\oplus,u) = \mathtt{lom}_k(\otimes,h\circ u),$$
$$h\circ\mathtt{rom}_k(\oplus,u) = \mathtt{rom}_k(\otimes,h\circ u),$$

and

$$h\circ(\oplus,u)_k = (\otimes,h\circ u)_k.$$

PROOF (SKETCH) This follows as before by direct calculation and the unique extension property. ■

Extend the definitions of folding operations to n-arrays with $n > 1$.

DEFINITION 4.9 If the unary function u equals the identity function

$$id(x) = x$$

then a left array hom, a right array hom, and an array catamorphism is called a **left fold**, a **right fold**, and a **reduce** operation, respectively.

The definition of map needs to be reformulated for n-arrays with $n > 1$.

DEFINITION 4.10 If the binary operator of a homomorphism is a concatenation operator, then a left array hom, a right array hom, or an array catamorphism is called a **planar map**.

The same higher-order operators are used, furnishing only the field direction in addition to the special operator needed to specify the homomorphism:

Homomorphism	Higher-order Function
left fold $= lom_k(\oplus, id)$	$\twoheadrightarrow_k \oplus$
right fold $= rom_k(\oplus, id)$	$\twoheadleftarrow_k \oplus$
reduce $= (\oplus, id)_k$	$/_k \oplus$
planar map $= (\#_k, f)_k$	$*_k f$

Notice

$$/ \neq /_1$$

and so it is not permissible to drop the 1 in this case. The difference is important. The function $/$ is useful for recursion as the higher-order operation collapses a 1-array via combination of the elements. The function $/_1$ combines hyperplanes without collapsing.

THEOREM 4.35 (REDUCE PROMOTION)
The following identity holds always:

$$(/_k \oplus) \circ (/ \#_k) = (/ \oplus) \circ * (/_k \oplus).$$

PROOF Since by definition

$$(/_k \oplus)(A \#_k B) = (/_k \oplus A) \oplus (/_k \oplus B),$$

plug $h = /_k \oplus$ into the promotability theorem to verify the stated identity. ■

THEOREM 4.36 Setting $\mathtt{T} = (\mathtt{t} : \mathtt{n}; \mathtt{t}_k = 1)$ the following identities hold:

$$(\twoheadrightarrow_k \oplus)[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$$
$$= (\cdots([\mathtt{a} \circ \tau_k(1) \mid \mathtt{T}] \oplus [\mathtt{a} \circ \tau_k(2) \mid \mathtt{T}])\cdots) \oplus [\mathtt{a} \circ \tau_k(\mathtt{t}_k) \mid \mathtt{T}],$$

$$(\twoheadleftarrow_k \oplus)[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$$
$$= [\mathtt{a} \circ \tau_k(1) \mid \mathtt{T}] \oplus (\cdots([\mathtt{a} \circ \tau_k(\mathtt{t}_k - 1) \mid \mathtt{T}] \oplus [\mathtt{a} \circ \tau_k(\mathtt{t}_k) \mid \mathtt{T}])\cdots),$$

and

$$(\diagup_k \oplus)[\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] = [\mathtt{a} \circ \tau_k(1) \mid \mathtt{T}] \oplus \cdots \oplus [\mathtt{a} \circ \tau_k(\mathtt{t}_k) \mid \mathtt{T}].$$

PROOF Plug $\mathtt{u} = \mathtt{id}$ in Theorem 4.33. ■

COROLLARY 4.37 For $\mathtt{n}$-arrays with $\mathtt{n} > 1$ the following identities hold:

$$\twoheadrightarrow_k \#_k = \twoheadleftarrow_k \#_k = \diagup_k \#_k = \left(\#_k, \mathtt{id}_{\mathcal{A}}\right)_k = \mathtt{id}_{\mathcal{A}}.$$

PROOF These identities hold by HC and inspection of Theorem 4.33 since $\#_k$ is associative. ■

EXAMPLE 4.8 Taking $\mathtt{A} = [\mathtt{a} : 3, 4] \sim \begin{bmatrix} 1 & 9 & 7 & 4 \\ 5 & 11 & 3 & 8 \\ 12 & 6 & 10 & 2 \end{bmatrix}$,

$$\diagup_2 \underset{\longleftrightarrow}{+} \mathtt{A} \sim \begin{bmatrix} 21 \\ 27 \\ 30 \end{bmatrix}, \quad \diagup_2 \underset{\longleftrightarrow}{\max} \mathtt{A} \sim \begin{bmatrix} 9 \\ 11 \\ 12 \end{bmatrix} \quad \text{and} \quad \diagup \underset{\longleftrightarrow}{\min} \mathtt{A} \sim \begin{bmatrix} 1 & 6 & 3 & 2 \end{bmatrix}.$$

EXAMPLE 4.9 Let $\mathtt{a} \oplus \mathtt{b} = \mathtt{a} - \frac{\mathtt{b}}{2}$. Let $\mathtt{A} = [\mathtt{a} : 3, 4] \sim \begin{bmatrix} 11 & 10 & 8 & 4 \\ 12 & 12 & 4 & 8 \\ 9 & 6 & 10 & 2 \end{bmatrix}$. Calculate

$$\twoheadrightarrow_2 \underset{\longleftrightarrow}{\oplus} \mathtt{A} \sim \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad \twoheadleftarrow_2 \underset{\longleftrightarrow}{\oplus} \mathtt{A} \sim \begin{bmatrix} \frac{15}{2} \\ 6 \\ \frac{33}{4} \end{bmatrix}.$$

THEOREM 4.38 Let $\mathtt{A} = [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$ with $\mathtt{n} > 1$ and $\mathtt{T} = (\mathtt{t} : \mathtt{n}; \mathtt{t}_k = 1)$. Then

$$\twoheadrightarrow_k \underset{\longleftrightarrow}{\oplus} \mathtt{A} = [\smile_k \oplus \mathtt{A} \mid \mathtt{T}],$$

$$\twoheadleftarrow_k \underset{\longleftrightarrow}{\oplus} \mathtt{A} = [\frown_k \oplus \mathtt{A} \mid \mathtt{T}],$$

and

$$\diagup_k \underset{\longleftrightarrow}{\oplus} \mathtt{A} = [\asymp_k \oplus \mathtt{A} \mid \mathtt{T}].$$

PROOF (SKETCH) The abstractor `zip` connects these folding operations. ■

THEOREM 4.39 A planar map distributes the given function to the hyperplanes as follows:

$$(*_k\ \mathtt{f})\,[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})] = \mathtt{f}\,[\mathtt{a}\circ\tau_k(1)\,|\,\mathtt{T}] \mathbin{+\!\!+_k} \cdots \mathbin{+\!\!+_k} \mathtt{f}\,[\mathtt{a}\circ\tau_k(\mathtt{t}_k)\,|\,\mathtt{T}]$$

where by TM

$$\mathtt{T} = (\mathtt{t}:\mathtt{n};\mathtt{t}_k = 1).$$

PROOF These identities hold by inspection of Theorem 4.33 and the definition of planar map. ■

THEOREM 4.40 A `map` may be demoted to a planar map:

$$*\mathtt{f} = *_k\,(*\mathtt{f})$$

for any `n`-array with $\mathtt{n} > 1$.

PROOF Consider an arbitrary `n`-array $[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})]$ for which $(*_k * \mathtt{f})$ may be applied. Invoking T write

$$\mathtt{T} = (\mathtt{t}:\mathtt{n};\mathtt{t}_k = 1).$$

Compute

$$\begin{aligned}
&(*_k * \mathtt{f})\,[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})] \\
&= (*\mathtt{f})\,[\mathtt{a}\circ\tau_k(1)\,|\,\mathtt{T}] \mathbin{+\!\!+_k} \cdots \mathbin{+\!\!+_k} (*\mathtt{f})\,[\mathtt{a}\circ\tau_k(\mathtt{t}_k)\,|\,\mathtt{T}] \text{ by Theorem 4.39} \\
&= [\mathtt{f}\circ(\mathtt{a}\circ\tau_k(1))\,|\,\mathtt{T}] \mathbin{+\!\!+_k} \cdots \mathbin{+\!\!+_k} [\mathtt{f}\circ(\mathtt{a}\circ\tau_k(\mathtt{t}_k))\,|\,\mathtt{T}] \text{ by Definition 3.28} \\
&= [(\mathtt{f}\circ\mathtt{a})\circ\tau_k(1)\,|\,\mathtt{T}] \mathbin{+\!\!+_k} \cdots \mathbin{+\!\!+_k} [(\mathtt{f}\circ\mathtt{a})\circ\tau_k(\mathtt{t}_k)\,|\,\mathtt{T}] \text{ by associativity} \\
&= [(\mathtt{f}\circ\mathtt{a})\,|\,(\mathtt{t}:\mathtt{n})] \text{ by HC} \\
&= (*\mathtt{f})\,[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})] \text{ by Definition 3.28.}
\end{aligned}$$

In addition,

$$(*_k * \mathtt{f})\,\square = \square$$

since $\square$ is the identity element for $\mathbin{+\!\!+_k}$. By Definition 3.28,

$$(*\mathtt{f})\,\square = \square.$$

This shows that `map` may be demoted to a planar map. ■

A planar map is applied only to the hyperplanes whereas a `map` is applied to all of the elements. Hence the parameter function which is mapped is defined accord-

ingly. Extended list homomorphisms are not defined for 1-arrays since they cannot be decomposed via reax operators.

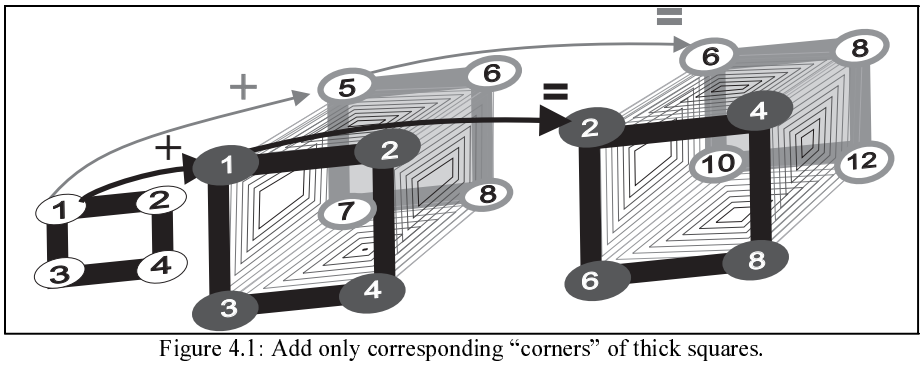


Figure 4.1: Add only corresponding "corners" of thick squares.

EXAMPLE 4.10 Add the number at each corner of $(\mathtt{N}_2)$ illustrated on the far left side of Figure 4.1 (2×2 matrix) to the number at the corresponding corner of both parallel thick squares of the leftmost 3-array B depicted in Figure 4.1 to obtain the rightmost 3-array. For instance, 4 is added only to the elements 4 and 8 of the argument (which is the 3-array on the left) to obtain the new elements 8 and 12, respectively, in the 3-array on the right. In other words, compute

$$*_3 \underset{\longleftrightarrow}{+} \mathtt{N}_2\, \mathtt{B} = *_3 \underset{\longleftrightarrow}{+} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \mathtt{B}.$$

LEMMA 4.41 (HYPERPLANE HOMOMORPHISM LEMMA)

If $*_k\, \mathtt{u}$ is defined then

$$\mathtt{lom}_k(\oplus, \mathtt{u}) = (\rightharpoonup_k \oplus) \circ (*_k\, \mathtt{u}),$$
$$\mathtt{rom}_k(\oplus, \mathtt{u}) = (\leftharpoonup_k \oplus) \circ (*_k\, \mathtt{u}),$$

and

$$(\oplus, \mathtt{u})_k = (\diagup_k \oplus) \circ (*_k\, \mathtt{u}).$$

PROOF This is an immediate consequence of the definitions and the promotion lemma for

$$\begin{aligned}
(\rightharpoonup_k \oplus) \circ (*_k\, \mathtt{u}) &= (\rightharpoonup_k \oplus) \circ \mathtt{lom}_k(+\!\!+_k, \mathtt{u}) \text{ by definition} \\
&= \mathtt{lom}_k(\oplus, \mathtt{u}) \text{ by the promotion lemma,} \\
(\leftharpoonup_k \oplus) \circ (*_k\, \mathtt{u}) &= (\leftharpoonup_k \oplus) \circ \mathtt{rom}_k(+\!\!+_k, \mathtt{u}) \text{ by definition} \\
&= \mathtt{rom}_k(\oplus, \mathtt{u}) \text{ by the promotion lemma,}
\end{aligned}$$

and lastly,

$$(/_k \oplus) \circ (*_k u) = (/_k \oplus) \circ (\!|\, +\!\!+_k, u \,|\!)_k \text{ by definition}$$
$$= (\!|\, \oplus, u \,|\!)_k \text{ by the promotion lemma. } \blacksquare$$

EXAMPLE 4.11 Recall $N_2 = [b : 2, 2]$ where $b(i, j) = 2(i-1)+j$. Put

$$B = [b : 2, 1] \quad \text{and} \quad C = [b \circ \tau_2(2) : 2, 1]$$

so that

$$N_2 = B +\!\!+_2 C \sim \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \text{concatenate}\left(\begin{bmatrix} 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 4 \end{bmatrix}\right).$$

Then

$$\left(/_2 \underset{\longleftrightarrow}{+}\right) \circ (* \times 2) N_2 = \left(/_2 \underset{\longleftrightarrow}{+}\right) \circ (* \times 2) B + \left(/_2 \underset{\longleftrightarrow}{+}\right) \circ (* \times 2) C.$$

In other words, first multiplying N_2 by 2 and then computing the row sums, i.e.,

$$2 \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \longmapsto \begin{bmatrix} 6 \\ 14 \end{bmatrix}$$

yields the same result as first multiplying the columns by 2 and then performing matrix addition:

$$2 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 2 \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \end{bmatrix} + \begin{bmatrix} 4 \\ 8 \end{bmatrix} = \begin{bmatrix} 6 \\ 14 \end{bmatrix}.$$

THEOREM 4.42 (HYPERPLANE PROMOTABILITY THEOREM)

If h is any homomorphism such that

$$h(a \oplus b) = h(a) \otimes h(b),$$

then

$$h \circ \text{lom}_k(\oplus, u) = (\twoheadrightarrow_k \otimes) \circ (*_k (h \circ u)),$$
$$h \circ \text{rom}_k(\oplus, u) = (\twoheadleftarrow_k \otimes) \circ (*_k (h \circ u)),$$

and

$$h \circ (\!|\, \oplus, u \,|\!)_k = (/_k \otimes) \circ (*_k (h \circ u)),$$

provided $*_k(h \circ u)$ is defined.

PROOF Calculate

$$\begin{aligned} h \circ lom_k(\oplus, u) &= lom_k(\otimes, h \circ u) \text{ by the promotion lemma} \\ &= (\rightharpoonup_k \otimes) \circ (*_k(h \circ u)) \text{ by the homomorphism lemma,} \\ h \circ rom_k(\oplus, u) &= rom_k(\otimes, h \circ u) \text{ by the promotion lemma} \\ &= (\leftharpoonup_k \otimes) \circ (*_k(h \circ u)) \text{ by the homomorphism lemma,} \end{aligned}$$

and lastly,

$$\begin{aligned} h \circ (\oplus, u)_k &= (\otimes, h \circ u)_k \text{ by the promotion lemma} \\ &= (\diagup_k \otimes) \circ (*_k(h \circ u)) \text{ by the homomorphism lemma.} \blacksquare \end{aligned}$$

EXAMPLE 4.12 If $u = id$ then

$$h \circ (\rightharpoonup_k \oplus) = (\rightharpoonup_k \otimes) \circ (*_k h),$$
$$h \circ (\leftharpoonup_k \oplus) = (\leftharpoonup_k \otimes) \circ (*_k h),$$

and

$$h \circ (\diagup_k \oplus) = (\diagup_k \otimes) \circ (*_k h).$$

EXAMPLE 4.13 Taking $h = *f$, $\oplus = \#_k$ and $u = id$, and invoking Corollary 4.37, a restatement of Theorem 4.40 is obtained:

$$(*f) \circ (\diagup_k \#_k) = (\diagup_k \#_k) \circ (*_k * f)$$

since

$$*f(A \#_k B) = (*f\, A) \#_k (*f\, B).$$

EXAMPLE 4.14 If $h(A \#_k B) = h(A) \oplus h(B)$ then

$$h \circ (*_k f) = (\diagup_k \oplus) \circ *_k (h \circ f).$$

LEMMA 4.43 The generating relation $reax_k$ is a homomorphism:

$$reax_k(A \#_p B) = (reax_k A) \#_p (reax_k B) \text{ if } p < k,$$
$$reax_k(A \#_p B) = (reax_k A) \#_{p+1} (reax_k B) \text{ if } p \geq k.$$

PROOF (SKETCH) The generating relation $reax_k$ shifts by one all components starting from the kth component. ■

THEOREM 4.44 (REAX PROMOTABILITY)

$$\mathrm{reax}_k \circ (/_p \# _p) = (/_p \# _p) \circ (*_p \mathrm{reax}_k) \quad \text{if } p < k,$$

and

$$\mathrm{reax}_k \circ (/_p \# _p) = (/_p \# _{p+1}) \circ (*_p \mathrm{reax}_k) \quad \text{if } p \geq k.$$

PROOF If $p < k$, then in view of Lemma 4.43 and the hyperplane promotability theorem,

$$\mathrm{reax}_k \circ (\# _p, \mathrm{id})_p = (/_p \# _p) \circ (*_p \mathrm{reax}_k).$$

If $p \geq k$, then in view of Lemma 4.43 and the hyperplane promotability theorem,

$$\mathrm{reax}_k \circ (\# _p, \mathrm{id})_p = (/_p \# _{p+1}) \circ (*_p \mathrm{reax}_k). \blacksquare$$

LEMMA 4.45 The destructor $\mathrm{collapse}_k$ is a homomorphism:

$$\mathrm{collapse}_k (A \#_p B) = (\mathrm{collapse}_k A) \#_p (\mathrm{collapse}_k B) \quad \text{if } p < k,$$

and

$$\mathrm{collapse}_k (A \#_{p+1} B) = (\mathrm{collapse}_k A) \#_p (\mathrm{collapse}_k B) \quad \text{if } p \geq k.$$

PROOF Replace A and B by $\mathrm{collapse}_k A$ and $\mathrm{collapse}_k B$, respectively, in Lemma 4.43 to obtain

$$\mathrm{reax}_k ((\mathrm{collapse}_k A) \#_p (\mathrm{collapse}_k B)) = A \#_p B \quad \text{if } p < k,$$

and

$$\mathrm{reax}_k ((\mathrm{collapse}_k A) \#_p (\mathrm{collapse}_k B)) = A \#_{p+1} B \quad \text{if } p \geq k.$$

Apply $\mathrm{collapse}_k$ to both sides and invoke RC to show that the stated identities hold. ■

THEOREM 4.46 (COLLAPSE PROMOTABILITY)

$$\mathrm{collapse}_k \circ (/_p \# _p) = (/_p \# _p) \circ (*_p \mathrm{collapse}_k) \quad \text{if } p < k,$$

and

$$\mathrm{collapse}_k \circ (/_p \# _{p+1}) = (/_p \# _p) \circ (*_p \mathrm{collapse}_k) \quad \text{if } p \geq k.$$

PROOF If $p < k$, then in view of Lemma 4.45 and the hyperplane promotability theorem,

$$\mathrm{collapse}_k \circ (\# _p, \mathrm{id})_p = (/_p \# _p) \circ (*_p \mathrm{collapse}_k).$$

If $p \geq k$, then in view of Lemma 4.45 and the hyperplane promotability theorem,

$$\text{collapse}_k \circ (\#_{p+1}, \text{id})_p = (/_p \#_p) \circ (*_p \text{collapse}_k) . \blacksquare$$

THEOREM 4.47 For all n-arrays, $k \in \iota n$ and $p \in \iota (n-1)$,

$$\left(\twoheadrightarrow_p \underset{\longleftrightarrow}{\oplus}\right) \circ \text{collapse}_k = \text{collapse}_k \circ \left(\twoheadrightarrow_{p+1} \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if} \quad p \geq k,$$

$$\left(\twoheadrightarrow_p \underset{\longleftrightarrow}{\oplus}\right) \circ \text{collapse}_k = \text{collapse}_k \circ \left(\twoheadrightarrow_p \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if} \quad p < k,$$

$$\left(\twoheadleftarrow_p \underset{\longleftrightarrow}{\oplus}\right) \circ \text{collapse}_k = \text{collapse}_k \circ \left(\twoheadleftarrow_{p+1} \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if} \quad p \geq k,$$

$$\left(\twoheadleftarrow_p \underset{\longleftrightarrow}{\oplus}\right) \circ \text{collapse}_k = \text{collapse}_k \circ \left(\twoheadleftarrow_p \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if} \quad p < k,$$

$$\left(/_p \underset{\longleftrightarrow}{\oplus}\right) \circ \text{collapse}_k = \text{collapse}_k \circ \left(/_{p+1} \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if} \quad p \geq k,$$

and

$$\left(/_p \underset{\longleftrightarrow}{\oplus}\right) \circ \text{collapse}_k = \text{collapse}_k \circ \left(/_p \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if} \quad p < k.$$

PROOF (SKETCH) The field direction may need to be adjusted, depending on when collapsing takes place. ■

List homomorphisms have been extended to n-arrays with $n > 1$. This extension is a significant enhancement. Nevertheless, arrays are more than lists of hyperplanes. Array data structures permit additional array homomorphisms.

4.3.3 ARRAY HOMOMORPHISMS

As with lists, principal operations act in field directions. List homomorphisms lift functions on the elements to functions on lists. It is possible to lift list homomorphisms to operations on axes. This lifting is possible due to the dimensions which arrays possess.

DEFINITION 4.11 A **harray** (pronounced "hu-'rA" as in hurrah or "här · rA" as in **har**mony and **ray**) is a n-array of functions such that every element is a left hom, a right hom or a catamorphism.

Now it is possible to perform different operations in the same field direction. If F is a harray, then

$$\circledast_k F$$

applies each list homomorphism to the corresponding axis.

THEOREM 4.48 (AXIAL MORPHISM THEOREM)

Assume $\mathtt{F} = [\mathtt{f} \mid \mathtt{T}]$ is a harray. For $\mathtt{t} \in \mathbb{P}(\mathtt{T})$, there exist a binary operator $\theta(\mathtt{t})$ and an unary function $\mu(\mathtt{t})$ which determine each homomorphism $\mathtt{f}(\mathtt{t})$. If every element of $\mathtt{F}$ is a left hom, then

$$\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}] = (\multimap [\rightsquigarrow \circ \theta \mid \mathtt{T}]) \circ (\multimap [* \circ \mu \mid \mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}} \mid \mathtt{T}].$$

If every element of $\mathtt{F}$ is a right hom, then

$$\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}] = (\multimap [\leftsquigarrow \circ \theta \mid \mathtt{T}]) \circ (\multimap [* \circ \mu \mid \mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}} \mid \mathtt{T}].$$

If every element of $\mathtt{F}$ is a catamorphism, then

$$\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}] = (\multimap [\diagup \circ \theta \mid \mathtt{T}]) \circ (\multimap [* \circ \mu \mid \mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}} \mid \mathtt{T}].$$

PROOF (SKETCH) Invoke the unique extension property and homomorphism lemma. ■

DEFINITION 4.12 A **reducing array** is a harray in which every element is a left fold, a right fold or a reduce operation.

THEOREM 4.49 Assume $\mathtt{F} = [\mathtt{f} \mid \mathtt{T}]$ is a reducing array. For $\mathtt{t} \in \mathbb{P}(\mathtt{T})$, there exists a binary operator $\theta(\mathtt{t})$ which determines each homomorphism $\mathtt{f}(\mathtt{t})$. If every element of a harray $[\mathtt{f} \mid \mathtt{T}]$ is a left fold then

$$\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}] = (\multimap [\rightsquigarrow \circ \theta \mid \mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}} \mid \mathtt{T}].$$

If every element of a harray $\mathtt{R}$ is a right fold then

$$\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}] = (\multimap [\leftsquigarrow \circ \theta \mid \mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}} \mid \mathtt{T}].$$

If every element of a harray $\mathtt{R}$ is a reduce operation, then

$$\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}] = (\multimap [\diagup \circ \theta \mid \mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}} \mid \mathtt{T}].$$

PROOF (SKETCH) Rewrite $\mathtt{f}$ in the proof of the axial morphism theorem. ■

DEFINITION 4.13 A **map array** is a harray in which every element is a `map`.

THEOREM 4.50 If $\mathtt{M} = [\mathtt{f} \mid \mathtt{T}]$ is a map array then

$$\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}] = (\multimap [* \circ \mu \mid \mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}} \mid \mathtt{T}],$$

where $\mu(t)$ is the unary function which determines the `map f`.

PROOF (SKETCH) Rewrite `f` in the proof of the axial morphism theorem. ■

THEOREM 4.51 (ARRAY MORPHISM THEOREM)

If `F` is a harray then

$$\circledast_k F = (\multimap R) \circ (\circledast_k M)$$

where `R` is a reducing array, `M` is a map array and

$$F = R \underset{\longleftrightarrow}{\circ} M.$$

PROOF (SKETCH) Invoke the homomorphism lemma and perform the computations. ■

The importance of the abstractors is apparent from the theorems. These abstractors provide highly parallel models for performing operations axially. The operations need not be different. It is possible to transform a sequential operation on arrays into an array operation (a harray whose elements are constant) using these implicitly parallel abstractors. In particular, if `h` is any list homomorphism and `k` is any field direction then for any `n`-tuple `T`,

$$\circledast_k [K_h \mid T]$$

is a highly parallel operation in field direction `k`.

Instead of focusing on only one dimension, consider collapsing an array totally. The function `map` modifies the elements without combining them. What is needed are templates for combining elements without modification of them.

It is possible to collapse in different ways. The most natural way to proceed is to collapse in the natural order. Every `n`-array has a `1`-axis. If $n > 1$, then after reducing the elements along the `1`-axis and collapsing, the resulting array has a `1`-axis. It suffices to use the same concatenation operator during each collapsing phase. In order to simplify the following recursive definitions, $h(\square)$ is treated as the identity element for the given binary operator $\oplus$ which is used to combine the elements.

DEFINITION 4.14 (**ordered left fold**)

An operation `h` is an ordered left fold for a `n`-array `A` with $n > 1$ using some binary operator $\oplus$ if

$$h(B +\!\!+ (\text{reax}\, C)) = h(B) \oplus h(C),$$

and h is an ordered left fold for C using the same binary operator $\oplus$ whenever

$$B +\!\!+ (\text{reax}\, C)$$

is a subarray of A. An operation h is an ordered left fold for a 1-array A using some binary operator $\oplus$ if h is a left fold for A employing the binary operator $\oplus$.

DEFINITION 4.15 (ordered right fold)

An operation h is an ordered right fold for a n-array A with $n > 1$ using some binary operator $\oplus$ if

$$h((\text{reax}\, B) +\!\!+ C) = h(B) \oplus h(C),$$

and h is an ordered right fold for B using the same binary operator $\oplus$ whenever

$$(\text{reax}\, B) +\!\!+ C$$

is a subarray of A. An operation h is an ordered right fold for a 1-array A using some binary operator $\oplus$ if h is a right fold for A employing the binary operator $\oplus$.

DEFINITION 4.16 (ordered reduce)

An operation h is an ordered reduce for a n-array A with $n > 1$ using some binary operator $\oplus$ if

$$h(B +\!\!+ C) = h(B) \oplus h(C),$$

whenever $B +\!\!+ C$ is a subarray of A,

$$h(\text{reax}\, C) = h(C),$$

and h is an ordered reduce for C using the same binary operator $\oplus$ whenever reax C is a subarray of A. An operation h is an ordered reduce for a 1-array A using some binary operator $\oplus$ if h is a reduce for A employing the binary operator $\oplus$.

An **ordered reduction** is an ordered left fold, an ordered right fold, or an ordered reduce operation. It is necessary to furnish the binary operator in order to specify an ordered reduction:

Homomorphism	Function
ordered left fold	$\gtrsim \oplus$
ordered right fold	$\lesssim \oplus$
ordered reduce	$\approx \oplus$

Reductions may be performed in any specified order. The entire array may or may not be collapsed to a scalar array. In this case, reductions are performed without collapsing in order to avoid relabeling axes. For instance, if reducing and collapsing

along the p-axis before the q-axis and $p < q$ then after removing the p-axis the q-axis becomes the $(q-1)$-axis in accordance with Theorem 4.47.

DEFINITION 4.17 (**permuted left fold**)

An operation h is a permuted left fold if some fixed binary operator $\oplus$ and for some n-permutation p,

$$h = (\twoheadrightarrow_{p_1} \oplus) \circ \cdots \circ (\twoheadrightarrow_{p_n} \oplus).$$

DEFINITION 4.18 (**permuted right fold**)

An operation h is a permuted right fold if for some fixed binary operator $\oplus$ and for some n-permutation p,

$$h = (\twoheadleftarrow_{p_1} \oplus) \circ \cdots \circ (\twoheadleftarrow_{p_n} \oplus).$$

DEFINITION 4.19 (**permuted reduce**)

An operation h is a permuted reduce if for some fixed binary operator $\oplus$ and for some n-permutation p,

$$h = (\diagup_{p_1} \oplus) \circ \cdots \circ (\diagup_{p_n} \oplus).$$

A **permuted reduction** is a permuted left fold, a permuted right fold or a permuted reduce. Given a n-permutation p and a binary operator $\oplus$, the following operators may be employed to specify a permuted reduction:

Homomorphism	Function
permuted left fold	$\twoheadrightarrow^p \oplus$
permuted right fold	$\twoheadleftarrow^p \oplus$
permuted reduce	$\diagup^p \oplus$

It may be possible to collapse an array doing the reductions in arbitrary order, i.e., the order is not specified.

DEFINITION 4.20 (**total left fold**)

An operation h is a total left fold for a n-array A with $n > 1$ given some binary operator $\oplus$ if for any n-permutation p

$$h(A) = (\cdot)_n^{-1} \circ (\twoheadrightarrow^p \oplus)\ A,$$

An operation h is a total left fold for a 1-array A if h is a left fold for A.

DEFINITION 4.21 (**total right fold**)

An operation $\mathtt{h}$ is an total right fold for a $\mathtt{n}$-array $\mathtt{A}$ with $\mathtt{n} > 1$ given some binary operator $\oplus$ if for any $\mathtt{n}$-permutation $\mathtt{p}$

$$\mathtt{h}(\mathtt{A}) = (\cdot)_{\mathtt{n}}^{-1} \circ (\twoheadleftarrow^{\mathtt{p}} \oplus)\ \mathtt{A},$$

An operation $\mathtt{h}$ is a total right fold for a $\mathtt{1}$-array $\mathtt{A}$ if $\mathtt{h}$ is a right fold for $\mathtt{A}$.

DEFINITION 4.22 (**total reduce**)

An operation $\mathtt{h}$ is a total reduce for a $\mathtt{n}$-array $\mathtt{A}$ with $\mathtt{n} > 1$ given some binary operator $\oplus$ if for any $\mathtt{n}$-permutation $\mathtt{p}$

$$\mathtt{h}(\mathtt{A}) = (\cdot)_{\mathtt{n}}^{-1} \circ (\diagup^{\mathtt{p}} \oplus)\ \mathtt{A},$$

An operation $\mathtt{h}$ is a total reduce for a $\mathtt{1}$-array $\mathtt{A}$ if $\mathtt{h}$ is a reduce for $\mathtt{A}$.

A **total reduction** is a total left fold, a total right fold or a total reduce. To specify a total reduction, it is required to provide the binary operator $\oplus$:

Total Reduction	Higher-order Function
total left fold	$\twoheadrightarrow^* \oplus$
total right fold	$\twoheadleftarrow^* \oplus$
total reduce	$\diagup^* \oplus$

An operation is a **total homomorphism** if it can be expressed as an ordered reduction, a total reduction, a $\mathtt{map}$ or a composition of such functions.

EXAMPLE 4.15 Given a $\mathtt{2}$-array $\mathtt{A}$, an associative and commutative operator such as usual addition $+$, it is possible to reduce by rows and then by columns or by columns and then by rows so that

$$\diagup^* + \mathtt{A}$$

is well-defined.

EXAMPLE 4.16 The binary operator need not be associative in a total left fold. Define $\oplus$ via

$$\mathtt{a} \oplus \mathtt{b} = \mathtt{a} - \frac{\mathtt{b}}{2}.$$

Then

$$(1 \oplus 2) \oplus 3 = -\frac{3}{2} \qquad \text{and} \qquad 1 \oplus (2 \oplus 3) = \frac{3}{4}.$$

This shows that $\oplus$ is nonassociative. Now consider any 2-array $[\mathtt{a}:2,2]$. Compute

$$(\twoheadrightarrow_1 \oplus)\circ(\twoheadrightarrow_2 \oplus)[\mathtt{a}:2,2] = (\twoheadrightarrow_2 \oplus)\circ(\twoheadrightarrow_1 \oplus)[\mathtt{a}:2,2]$$
$$= (\cdot)_2\left(\mathtt{a}_{1,1} - \frac{1}{2}\mathtt{a}_{1,2} - \frac{1}{2}\mathtt{a}_{2,1} + \frac{1}{4}\mathtt{a}_{2,2}\right).$$

The total left fold

$$(\twoheadrightarrow^* \oplus)[\mathtt{a}:2,2] = \mathtt{a}_{1,1} - \frac{1}{2}\mathtt{a}_{1,2} - \frac{1}{2}\mathtt{a}_{2,1} + \frac{1}{4}\mathtt{a}_{2,2}$$

is defined for all suitable array functions $\mathtt{a}$.

EXAMPLE 4.17 Let $\oplus$ be the nonassociative operator in Example 4.16. Let $\mathtt{A} = [\mathtt{a}:2,3]$ be any array. Then

$$(\twoheadrightarrow^* \oplus)[\mathtt{a}:2,3] = \mathtt{a}_{1,1} - \frac{1}{2}\mathtt{a}_{1,2} - \frac{1}{2}\mathtt{a}_{1,3} - \frac{1}{2}\mathtt{a}_{2,1} + \frac{1}{4}\mathtt{a}_{2,2} + \frac{1}{4}\mathtt{a}_{2,3}.$$

For instance, if $\mathtt{A} \sim \left[\begin{smallmatrix}8&4&2\\6&3&1\end{smallmatrix}\right]$ then $(\twoheadrightarrow^* \oplus)[\mathtt{a}:2,3] = 3$. However, splitting the matrix

$$\begin{bmatrix}8&4&2\\6&3&1\end{bmatrix} \quad \text{into two matrices} \quad \begin{bmatrix}8\\6\end{bmatrix} \quad \text{and} \quad \begin{bmatrix}4&2\\3&1\end{bmatrix}$$

(which is an invalid splitting for left homs) and computing

$$\left(8 - \frac{1}{2}6\right) - \frac{1}{2}\left(\left(4 - \frac{1}{2}2\right) - \frac{1}{2}\left(3 - \frac{1}{2}1\right)\right) = \frac{33}{8},$$

it follows that the total reduction

$$(\diagup^* \oplus)[\mathtt{a}:2,3]$$

is not well-defined.

EXAMPLE 4.18 In Examples 4.16 and 4.17, a nonassociative operator is employed in a total left hom. Yet even an associative operator $\oplus$ does not guarantee $\twoheadrightarrow^* \oplus$ is defined. Let

$$\mathtt{A} = [\mathtt{a}:2,2] \sim \begin{bmatrix}\left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right] & \left[\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right]\\ \left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right] & \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right]\end{bmatrix},$$

and $\otimes$ denote usual multiplication of matrices so that

$$\left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right]\otimes\left[\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right] = \left[\begin{smallmatrix}1&2\\1&1\end{smallmatrix}\right], \quad \left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right]\otimes\left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right] = \left[\begin{smallmatrix}1&1\\2&1\end{smallmatrix}\right], \quad \left[\begin{smallmatrix}1&2\\1&1\end{smallmatrix}\right]\otimes\left[\begin{smallmatrix}1&1\\2&1\end{smallmatrix}\right] = \left[\begin{smallmatrix}5&3\\3&2\end{smallmatrix}\right],$$

$$\left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right]\otimes\left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right] = \left[\begin{smallmatrix}1&2\\0&1\end{smallmatrix}\right], \quad \left[\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right]\otimes\left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right] = \left[\begin{smallmatrix}2&1\\1&1\end{smallmatrix}\right], \quad \text{and} \quad \left[\begin{smallmatrix}1&2\\0&1\end{smallmatrix}\right]\otimes\left[\begin{smallmatrix}2&1\\1&1\end{smallmatrix}\right] = \left[\begin{smallmatrix}4&3\\1&1\end{smallmatrix}\right].$$

This shows that

$$(\twoheadrightarrow_1 \otimes) \circ (\twoheadrightarrow_2 \otimes)\,[\mathtt{a}:2,2] \sim \left[\begin{smallmatrix} 5 & 3 \\ 3 & 2 \end{smallmatrix}\right] \neq \left[\begin{smallmatrix} 4 & 3 \\ 1 & 1 \end{smallmatrix}\right] \sim (\twoheadrightarrow_2 \otimes) \circ (\twoheadrightarrow_1 \otimes)\,[\mathtt{a}:2,2].$$

Hence, the total reduction $(\twoheadrightarrow^* \otimes)\,[\mathtt{a}:2,2]$ is not well-defined.

EXAMPLE 4.19 Summing all of the columns and then adding all of the resulting column sums is equivalent to summing all of the rows and then adding all of the resulting row sums. Totally reduce the 2-array

$$\mathtt{A} = [\mathtt{a}:4,3] \sim \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

by computing

$$\diagup^* + \mathtt{A} = 1+2+3+\cdots+12 = \frac{12 \cdot 13}{2} = 78.$$

Without utilizing the total reduction, it would be necessary to compute via either the row sums

$$(\cdot)_2^{-1} \circ (\diagup_1 +) \circ (\diagup_2 +)\ \mathtt{A}$$

or the column sums

$$(\cdot)_2^{-1} \circ (\diagup_2 +) \circ (\diagup_1 +)\ \mathtt{A}.$$

Calculate $\diagup^* \uparrow \mathtt{A}$ to obtain the maximum entry 12 and $\diagup^* \downarrow \mathtt{A}$ to obtain the minimum entry 1.

4.4 LAWS AND PROPERTIES

LEMMA 4.52 For the constant identity function $\mathtt{K}_{\mathtt{id}}$,

$$\multimap [\mathtt{K}_{\mathtt{id}} \,|\, \mathtt{T}]\,[\mathtt{a} \,|\, \mathtt{T}] = [\mathtt{a} \,|\, \mathtt{T}].$$

PROOF By Definition 3.30,

$$\multimap [\mathtt{K}_{\mathtt{id}} \,|\, \mathtt{T}]\ [\mathtt{a} \,|\, \mathtt{T}] = [@ \circ (\mathtt{K}_{\mathtt{id}}, \mathtt{a}) \,|\, \mathtt{T}]$$

and

$$@ \circ (\mathtt{K}_{\mathtt{id}}, \mathtt{a})(\mathtt{t}) = @(\mathtt{id}, \mathtt{a}(\mathtt{t})) = \mathtt{a}(\mathtt{t}),$$

which confirms the stated identity. ■

THEOREM 4.53 (BIAXIS IDENTITY)

The following identity holds always:

$$\circledast_{\mathtt{k}}\,[\mathtt{f}\,|\,\mathtt{T}] = (\multimap\,[\mathtt{f}\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}\,[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}].$$

PROOF (SKETCH) Apply each component function to each axis by first forming the array of axes and then pairing functions with corresponding axes. ■

THEOREM 4.54 (BICOMP DISTRIBUTIVITY)

The following distributive law holds:

$$\multimap\,[(\circ)\circ(\mathtt{f},\mathtt{g})\,|\,\mathtt{T}] = (\multimap\,[\mathtt{f}\,|\,\mathtt{T}]) \circ (\multimap\,[\mathtt{g}\,|\,\mathtt{T}]).$$

PROOF Calculate

$$\begin{aligned}(\multimap\,[\mathtt{f}\,|\,\mathtt{T}]) \circ (\multimap\,[\mathtt{g}\,|\,\mathtt{T}])\ [\mathtt{a}\,|\,\mathtt{T}] &= (\multimap\,[\mathtt{f}\,|\,\mathtt{T}])\,[@\circ(\mathtt{g},\mathtt{a})\,|\,\mathtt{T}] \\ &= [@\circ(\mathtt{f},@\circ(\mathtt{g},\mathtt{a}))\,|\,\mathtt{T}],\end{aligned}$$

$$\begin{aligned}@\circ(\mathtt{f},@\circ(\mathtt{g},\mathtt{a}))(\mathtt{t}) &= @\circ(\mathtt{f}(\mathtt{t}),@\circ(\mathtt{g},\mathtt{a})(\mathtt{t})) \\ &= @\circ(\mathtt{f}(\mathtt{t}),\mathtt{g}(\mathtt{t})(\mathtt{a}(\mathtt{t}))) \\ &= \mathtt{f}(\mathtt{t})(\mathtt{g}(\mathtt{t})(\mathtt{a}(\mathtt{t}))) \\ &= (\mathtt{f}(\mathtt{t})\circ\mathtt{g}(\mathtt{t}))(\mathtt{a}(\mathtt{t})) \\ &= @\circ((\circ)\circ(\mathtt{f},\mathtt{g}),\mathtt{a})(\mathtt{t}),\end{aligned}$$

and

$$\multimap\,[(\circ)\circ(\mathtt{f},\mathtt{g})\,|\,\mathtt{T}]\ [\mathtt{a}\,|\,\mathtt{T}] = [@\circ((\circ)\circ(\mathtt{f},\mathtt{g}),\mathtt{a})\,|\,\mathtt{T}],$$

which verifies the stated identity by inspection. ■

THEOREM 4.55 (TRIAD-ZIP IDENTITY)

If $\oplus$ is any binary operator then

$$\overleftrightarrow{[\mathtt{K}_{\oplus}\,|\,\mathtt{T}]} = \underset{\longleftrightarrow}{\oplus}.$$

PROOF Recall,

$$[\mathtt{a}\,|\,\mathtt{T}]\ \underset{\longleftrightarrow}{\oplus}\ [\mathtt{b}\,|\,\mathtt{T}] = [\oplus\circ(\mathtt{a},\mathtt{b})\,|\,\mathtt{T}]$$

and

$$\overleftrightarrow{[\mathtt{K}_{\oplus}\,|\,\mathtt{T}]}\,[\mathtt{a}\,|\,\mathtt{T}]\,[\mathtt{b}\,|\,\mathtt{T}] = [\mathtt{tribi}\circ(\mathtt{K}_{\oplus},\mathtt{a},\mathtt{b})\,|\,\mathtt{T}]$$

where

$$\mathtt{tribi} \circ (\mathtt{K}_{\oplus}, \mathtt{a}, \mathtt{b})(\mathtt{t}) = \mathtt{a}(\mathtt{t}) \oplus \mathtt{b}(\mathtt{t}),$$

which is the corresponding element of $[\oplus \circ (\mathtt{a}, \mathtt{b}) \mid \mathtt{T}]$. ■

THEOREM 4.56 (MAP DISTRIBUTIVITY)

$$*(\mathtt{f} \circ \mathtt{g}) = (*\mathtt{f}) \circ (*\mathtt{g}).$$

PROOF It suffices to consider $\mathtt{n}$-arrays with $\mathtt{n} > 1$ since map distributivity has been established for 1-arrays. Compare

$$*(\mathtt{f} \circ \mathtt{g})\, \square = \square,$$

$$(*\mathtt{f}) \circ (*\mathtt{g})\, \square = (*\mathtt{f})\, \square = \square,$$

$$*(\mathtt{f} \circ \mathtt{g})\, [\mathtt{a} \mid \mathtt{T}] = [(\mathtt{f} \circ \mathtt{g}) \circ \mathtt{a} \mid \mathtt{T}] = [\mathtt{f} \circ \mathtt{g} \circ \mathtt{a} \mid \mathtt{T}],$$

and

$$(*\mathtt{f}) \circ (*\mathtt{g})\, [\mathtt{a} \mid \mathtt{T}] = (*\mathtt{f})\, [\mathtt{g} \circ \mathtt{a} \mid \mathtt{T}] = [\mathtt{f} \circ (\mathtt{g} \circ \mathtt{a}) \mid \mathtt{T}] = [\mathtt{f} \circ \mathtt{g} \circ \mathtt{a} \mid \mathtt{T}].$$

By inspection, $*(\mathtt{f} \circ \mathtt{g}) = (*\mathtt{f}) \circ (*\mathtt{g})$. ■

LEMMA 4.57 For the constant function $\mathtt{K}_\mathtt{c}$,

$$(*\,\mathtt{K}_\mathtt{c})\, [\mathtt{a} \mid \mathtt{T}] = [\mathtt{K}_\mathtt{c} \mid \mathtt{T}].$$

PROOF By definition,

$$(*\,\mathtt{K}_\mathtt{c})\, [\mathtt{a} \mid \mathtt{T}] = [\mathtt{K}_\mathtt{c} \circ \mathtt{a} \mid \mathtt{T}]$$

and

$$\mathtt{K}_\mathtt{c} \circ \mathtt{a}(\mathtt{t}) = \mathtt{K}_\mathtt{c}(\mathtt{a}(\mathtt{t})) = \mathtt{c} = \mathtt{K}_\mathtt{c}(\mathtt{t}),$$

which verifies the stated identity. ■

LEMMA 4.58 For any binary operator $\oplus$,

$$\underset{\longleftrightarrow}{\oplus} = (*\oplus) \circ \underset{\longleftrightarrow}{\mathtt{id}}.$$

PROOF Since

$$(*\oplus) \circ \underset{\longleftrightarrow}{\mathtt{id}}\ \square\ \square\ =\ (*\oplus)\ \square\ =\ \square = \underset{\longleftrightarrow}{\oplus}\ \square\ \square,$$

and

$$(*\oplus) \circ \underset{\longleftrightarrow}{\mathtt{id}}\ [\mathtt{a} \mid \mathtt{T}]\ [\mathtt{b} \mid \mathtt{T}] = (*\oplus)\, [(\mathtt{a}, \mathtt{b}) \mid \mathtt{T}] = [\oplus \circ (\mathtt{a}, \mathtt{b}) \mid \mathtt{T}] = \underset{\longleftrightarrow}{\oplus}\ [\mathtt{a} \mid \mathtt{T}]\ [\mathtt{b} \mid \mathtt{T}],$$

it follows that the stated identity holds. ■

THEOREM 4.59 (MAP INVERSES)

A `map` has inverses whenever the parameter function possesses an inverse:

$$(*\,\mathtt{f})^{-1} = *\left(\mathtt{f}^{-1}\right).$$

PROOF Compute

$$\left(*\left(\mathtt{f}^{-1}\right)\right)\circ(*\mathtt{f})\,\square = \left(*\left(\mathtt{f}^{-1}\right)\right)\square = \square,$$

$$\begin{aligned}\left(*\left(\mathtt{f}^{-1}\right)\right)\circ(*\mathtt{f})\,[\mathtt{a}\,|\,\mathtt{T}] &= \left(*\left(\mathtt{f}^{-1}\right)\right)[\mathtt{f}\circ\mathtt{a}\,|\,\mathtt{T}]\\ &= \left[\left(\mathtt{f}^{-1}\right)\circ(\mathtt{f}\circ\mathtt{a})\,|\,\mathtt{T}\right]\\ &= \left[\left(\mathtt{f}^{-1}\circ\mathtt{f}\right)\circ\mathtt{a}\,|\,\mathtt{T}\right]\\ &= [\mathtt{id}\circ\mathtt{a}\,|\,\mathtt{T}]\\ &= [\mathtt{a}\,|\,\mathtt{T}],\end{aligned}$$

and similarly,

$$(*\,\mathtt{f})\circ\left(*\left(\mathtt{f}^{-1}\right)\right)(\square) = (*\,\mathtt{f})(\square) = \square,$$

$$\begin{aligned}(*\,\mathtt{f})\circ\left(*\left(\mathtt{f}^{-1}\right)\right)[\mathtt{a}\,|\,\mathtt{T}] &= (*\,(\mathtt{f}))\left[\mathtt{f}^{-1}\circ\mathtt{a}\,|\,\mathtt{T}\right]\\ &= \left[\mathtt{f}\circ\left(\mathtt{f}^{-1}\circ\mathtt{a}\right)\,|\,\mathtt{T}\right]\\ &= \left[\left(\mathtt{f}\circ\mathtt{f}^{-1}\right)\circ\mathtt{a}\,|\,\mathtt{T}\right]\\ &= [\mathtt{id}\circ\mathtt{a}\,|\,\mathtt{T}]\\ &= [\mathtt{a}\,|\,\mathtt{T}].\end{aligned}$$

Evidently, $\left(*\left(\mathtt{f}^{-1}\right)\right)\circ(*\,\mathtt{f}) = \mathtt{id} = (*\,\mathtt{f})\circ\left(*\left(\mathtt{f}^{-1}\right)\right)$ for all arrays whence $(*\,\mathtt{f})^{-1} = *\left(\mathtt{f}^{-1}\right)$. ■

As `n`-arrays are expressed in terms of the constructors, it is essential that they also have useful algebraic properties. A useful property for binary operators is associativity. There is some obvious freedom of computation implied by associativity.

EXAMPLE 4.20 Consider usual addition: $\mathtt{x}_1 + \mathtt{x}_2 + \mathtt{x}_3 + \cdots + \mathtt{x}_n$. In FORTRAN a `DO` loop is typically employed to process elements of an array. In particular, computations proceed from one end to the other. Since addition is an associative operation, it is possible to perform the computation in different ways. For instance, the computation

could be done via a binary tree. Using a low-level language like FORTRAN, programming is controlled and limited by the language. What is desired is a language that allows the specification of a program without imposing unnecessary constraints, and that provides adequate tools in an algebra of programming. Mathematical functions and properties like associativity provide such tools.

It is possible to exhibit the associativity of binary operations via concatenation, which is itself an associative operator.

CLAIM 4.6 The concatenation operator $\#_k$ is an associative operator.

PROOF (SKETCH) Verify that

$$(A \#_k B) \#_k C = A \#_k (B \#_k C)$$

for all arrays A, B and C for which $\#_k$ is defined. ■

THEOREM 4.60 (GENERALIZED ASSOCIATIVITY)
If for any operands A, B and C, the operator $\oplus$ satisfies

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

then for any finite sequence $A_1, A_2, \ldots, A_m$ of such operands,

(4.11) $$A_1 \oplus A_2 \oplus \cdots \oplus A_m$$

is defined and yields the same result for arbitrary groupings whenever all operations are defined between pairs.

PROOF (SKETCH) Employ an inductive argument. First show there can be at most two solutions by looking at all possible solutions involving pairs $(A_i \oplus A_{i+1})$. Then show these two solutions must be the same by looking at groupings of three. ■

Associativity is not merely a mathematical notion without practical significance.

THEOREM 4.61 (DUALITY THEOREM)
If $\oplus$ is an associative binary operator then

$$\twoheadrightarrow_k \oplus = /_k \oplus = \twoheadleftarrow_k \oplus.$$

In addition,

$$\mathrm{lom}_k(\#_k, f) = (\!|\#_k, f|\!)_k = \mathrm{rom}_k(\#_k, f).$$

PROOF First, consider 1-arrays. Assume $\oplus$ is associative. Let $\twoheadrightarrow \oplus$, $/ \oplus$ and $\twoheadleftarrow \oplus$ be homomorphisms for some array A. By inspection of equations (4.4), (4.5), (4.6), and generalized associativity, the stated equalities hold. Now

$$\mathrm{lom}(\#, f) = (\#, f) = \mathrm{rom}(\#, f).$$

follows by Theorem 4.23 since $\#$ is associative.

Next, consider n-arrays with $n > 1$. Assume $\oplus$ is associative. Let $\twoheadrightarrow_k \oplus$, $/_k \oplus$ and $\twoheadleftarrow_k \oplus$ be homomorphisms for some array A. By inspection of Theorem 4.36 and generalized associativity, the stated equalities hold. In addition,

$$\mathrm{lom}_k(\#_k, f) = (\#_k, f)_k = \mathrm{rom}_k(\#_k, f)$$

holds by Theorem 4.33 since $\#_k$ is associative. ■

For arrays, there is another "dimension" of associativity. Arrays may be partitioned and then reassembled in different ways. Invoking CI, repeatedly split A at arbitrary positions along any valid k-axis. By generalized associativity, the order of concatenations is irrelevant. So it is possible to write

$$A = A_1 \#_k A_2 \#_k \cdots \#_k A_m.$$

Next, partition each subarray A_i at arbitrary positions along any p-axis , provided the same positions along the p-axis are selected for each subarray. Write

$$A_i = A_{i,1} \#_p A_{i,2} \#_p \cdots \#_p A_{i,n},$$

to obtain

$$A = (A_{1,1} \#_p \cdots \#_p A_{1,n}) \#_k \cdots \#_k (A_{m,1} \#_p \cdots \#_p A_{m,n}). \tag{4.12}$$

A nested tiled array or tiling is a useful representation for an array of blocks

$A_{1,1}$	$A_{1,2}$	$\cdots$	$A_{1,n}$
$A_{2,1}$	$A_{2,2}$	$\cdots$	$A_{2,n}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$A_{m,1}$	$A_{m,2}$	$\cdots$	$A_{m,n}$

where each of the $A_{i,j}$'s are elements of a tiled array.

4.5 EXPRESSIVENESS

Due to familiarity with arrays using indices, it may seem that a n-array is expressive. It is customary to see algorithms and programs written using indices. As

programming has been deferred, it may seem that the primitives are not equally expressive. However, do not be deceived. The primitives are expressive.

Admittedly, $\mathtt{t}_k = 1$ is a fairly concise way to describe a hyperplane. Yet little information is contained in this equation $\mathtt{t}_k = 1$. It is not an array. On the other hand $\mathrm{reax}_k\,\mathtt{C}$ is an array which is also a hyperplane. In fact the expression $\mathtt{t}_k = 1$ has about the same number of symbols as $\mathrm{reax}_k\,\mathtt{C}$, treating reax as one symbol. This is just one example of expressiveness of the constructors.

Consider rotations to demonstrate the expressiveness of the constructors. The reader is invited to express the following statement using a n-array model:

$$\mathrm{reax}_k(\mathtt{A} \#_k \mathtt{B}) = (\mathrm{reax}_k\,\mathtt{A}) \#_{k+1} (\mathrm{reax}_k\,\mathtt{B}),$$

which is immediate from Lemma 4.43. Imagine first stacking two boxes and second rotating them together. The same result would be obtained if the boxes were first rotated before stacking.

EXAMPLE 4.21 If $\mathtt{A} = [\mathtt{a} : 1]$ and $\mathtt{B} = [\mathtt{b} : 1]$, then

$$\mathrm{reax}(\mathtt{A} \# \mathtt{B}) = [\mathtt{c} : 1, 2] = (\mathrm{reax}\ \mathtt{A}) \#_2 (\mathrm{reax}\ \mathtt{B}),$$

where $\mathtt{c}(1,1) = \mathtt{a}_1$ and $\mathtt{c}(1,2) = \mathtt{b}_1$.

EXAMPLE 4.22 Take $\mathtt{A} = [\mathtt{id} : 3]$ and $\mathtt{B} = [\mathtt{b} : 2]$ with $\mathtt{b}(\mathtt{i}) = \mathtt{i} + 3$. Then

$$\mathtt{C} = \mathtt{A} \# \mathtt{B} = [\mathtt{id} : 5] \qquad \text{and} \qquad \mathtt{D} = \mathrm{reax}\ \mathtt{C} = [\mathtt{d} : 1, 5]$$

where $\mathtt{d}(1,\mathtt{i}) = \mathtt{i}$. In addition, if $\mathtt{E} = \mathrm{reax}\ \mathtt{A} = [\mathtt{e} : 1, 3]$ with $\mathtt{e}(1,\mathtt{i}) = \mathtt{i}$ and $\mathtt{F} = \mathrm{reax}\ \mathtt{B} = [\mathtt{f} : 1, 2]$ with $\mathtt{f}(1,\mathtt{i}) = \mathtt{i} + 3$, then

$$\mathtt{G} = \mathtt{E} \#_2 \mathtt{F} = [\mathtt{g} : 1, 5] \qquad \text{and} \qquad \mathtt{D} = \mathtt{G}$$

where $\mathtt{g}(1,\mathtt{i}) = \mathtt{i}$.

4.6 SUMMARY

The consequences of the definitions have been formally investigated. Any array can be expressed in a few simple forms. Explicit forms have been stated. Arrays have been successfully transformed into constructors, i.e., it is possible to define operations and write programs using only the constructors.

List and array homomorphisms have been classified. The tool box is adequate to begin programming. Specifically, the following programming templates are available:

List Homomorphism	Function
left hom	$\mathtt{lom}(\oplus,\mathtt{u})$
right hom	$\mathtt{rom}(\oplus,\mathtt{u})$
catamorphism	$(\!|\oplus,\mathtt{u}|\!)$

List Homomorphism	Operator
left fold	$\twoheadrightarrow$
right fold	$\twoheadleftarrow$
reduce	$/$
map	$*$

Array Homomorphism	Operator
left fold	$\twoheadrightarrow_{\mathtt{k}}$
right fold	$\twoheadleftarrow_{\mathtt{k}}$
reduce	$/_{\mathtt{k}}$
planar map	$*_{\mathtt{k}}$

Total Homomorphism	Operator
total left fold	$\twoheadrightarrow_{*}$
total right fold	$\twoheadleftarrow_{*}$
total reduce	$/_{*}$
axial homomorphism	$\circledast_{\mathtt{k}}$

Array Homomorphism	Function
left array hom	$\mathtt{lom}_{\mathtt{k}}(\oplus,\mathtt{u})$
right array hom	$\mathtt{rom}_{\mathtt{k}}(\oplus,\mathtt{u})$
array catamorphism	$(\!|\oplus,\mathtt{u}|\!)_{\mathtt{k}}$

Array Homomorphism	Operator
ordered left fold	$\gtrsim$
ordered right fold	$\lesssim$
ordered reduce	$\asymp$

Array Homomorphism	Operator
permuted left fold	$\twoheadrightarrow^{\mathtt{p}}$
permuted right fold	$\twoheadleftarrow^{\mathtt{p}}$
permuted reduce	$/^{\mathtt{p}}$

It is always possible to arbitrarily define functions but they cannot always be combined to form a consistent and coherent theory. Sets of equations including many identities have been established. Some identities which have been verified are listed below. For $\mathtt{n}$-arrays with $\mathtt{n} > 1$ these formulas lack most of the indices which are typically needed in conventional notation. These identities are versatile as they express common patterns of computations.

For 1-arrays, the following identities hold:

$$[\mathtt{a} : \mathtt{n}] = [\mathtt{a} \circ \tau(1) : 1] \mathbin{\#} \cdots \mathbin{\#} [\mathtt{a} \circ \tau(\mathtt{n}) : 1],$$

$$\mathtt{lom}(\oplus,\mathtt{u})[\mathtt{a} : \mathtt{n}] = (\cdots((\mathtt{u}(\mathtt{a}_1) \oplus \mathtt{u}(\mathtt{a}_2)) \oplus \mathtt{u}(\mathtt{a}_3))\cdots) \oplus \mathtt{u}(\mathtt{a}_{\mathtt{n}}),$$
$$\mathtt{rom}(\oplus,\mathtt{u})[\mathtt{a} : \mathtt{n}] = \mathtt{u}(\mathtt{a}_1) \oplus (\mathtt{u}(\mathtt{a}_2) \oplus (\cdots(\mathtt{u}(\mathtt{a}_{\mathtt{n}-1}) \oplus \mathtt{u}(\mathtt{a}_{\mathtt{n}}))\cdots)),$$
$$(\!|\oplus,\mathtt{u}|\!)[\mathtt{a} : \mathtt{n}] = \mathtt{u}(\mathtt{a}_1) \oplus \mathtt{u}(\mathtt{a}_2) \oplus \cdots \oplus \mathtt{u}(\mathtt{a}_{\mathtt{n}}),$$

$$(\twoheadrightarrow \oplus)\,[a:n] = (\cdots((a_1 \oplus a_2) \oplus a_3)\cdots) \oplus a_n,$$
$$(\twoheadleftarrow \oplus)\,[a:n] = a_1 \oplus (a_2 \oplus (\cdots(a_{n-1} \oplus a_n)\cdots)),$$
$$(/\oplus)\,[a:n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n,$$

$$(*f)\,[a:n] = [f \circ a \circ \tau(1):1] \# \cdots \# [f \circ a \circ \tau(n):1].$$

For n-arrays A with $n > 1$ the following identities hold:

$$A = [a \circ \tau_k(1) \mid T] \#_k \cdots \#_k [a \circ \tau_k(t_k) \mid T],$$

$$\mathrm{lom}_k(\oplus, u)\ A = (\cdots(u\,[a \circ \tau_k(1) \mid T] \oplus u\,[a \circ \tau_k(2) \mid T])\cdots) \oplus u\,[a \circ \tau_k(t_k) \mid T],$$

$$\mathrm{rom}_k(\oplus, u)\ A = u\,[a \circ \tau_k(1) \mid T] \oplus (\cdots(u\,[a \circ \tau_k(t_k - 1) \mid T] \oplus u\,[a \circ \tau_k(t_k) \mid T])\cdots),$$

$$(\!|\oplus, u|\!)_k\ A = u\,[a \circ \tau_k(1) \mid T] \oplus \cdots \oplus u\,[a \circ \tau_k(t_k) \mid T],$$

$$\twoheadrightarrow_k \oplus\, A = (\cdots([a \circ \tau_k(1) \mid T] \oplus [a \circ \tau_k(2) \mid T])\cdots) \oplus [a \circ \tau_k(t_k) \mid T],$$
$$\twoheadleftarrow_k \oplus\, A = [a \circ \tau_k(1) \mid T] \oplus (\cdots([a \circ \tau_k(t_k - 1) \mid T] \oplus [a \circ \tau_k(t_k) \mid T])\cdots),$$
$$/_k \oplus\, A = [a \circ \tau_k(1) \mid T] \oplus \cdots \oplus [a \circ \tau_k(t_k) \mid T],$$

$$\twoheadrightarrow_k \underset{\longleftrightarrow}{\oplus}\, A = [\smile_k \oplus A \mid T],$$
$$\twoheadleftarrow_k \underset{\longleftrightarrow}{\oplus}\, A = [\frown_k \oplus A \mid T],$$
$$/_k \underset{\longleftrightarrow}{\oplus}\, A = [\asymp_k \oplus A \mid T],$$

$$*_k f\, A = f\,[a \circ \tau_k(1) \mid T], \#_k \cdots \#_k\, f\,[a \circ \tau_k(t_k) \mid T],$$

where $A = [a \mid (t:n)]$ and $T = (t:n; t_k = 1)$.

The following promotion laws hold:

$$*f \circ (/\#_k) = (/\#_k) \circ (**f),$$

and

$$(/_k\oplus) \circ (/\#_k) = (/\oplus) \circ *(/_k\oplus).$$

Some of the important lemmas and theorems that have been proved are listed below:

Promotion Lemma
Map Distributivity
Map Inverses
Homomorphism Lemma
Promotability Theorem
Duality Theorem
Axial Morphism Theorem
Array Morphism Theorem

Although the theory has been sufficiently developed to begin programming, it would be beneficial to find more properties of higher-order functions. Useful theoretical results could be sought especially for border, unborder, cover and untile. In particular, equational transformations are needed for optimization involving block algorithms.

CHAPTER 5

PROGRAMMING EXAMPLES

Regardless of the completeness, depth or elegance of a theory, a yardstick with which to measure a theory is the applicability to programming. Enough primitive operations should be provided to account for common operations so that programming is not too difficult. Examples are needed to show that the theory is relevant to programming practice. Computer scientists know that programmers learn by examples. In this chapter, many examples are given. The main focus is on general programming techniques, not on optimization which is possible using well-known techniques (see Section 1.4.3).

The primitive operations that deal with structural aspects together with their properties form the foundation upon which an algebra of programming is constructed. It is inefficient to go over the primitives in any more detail. They were defined in Chapter 3. Presumably they would be efficiently implemented.

Programming the primitive operations serves to check that an adequate set of primitive operations has been provided. It is possible to focus only on the constructors and to disallow other primitive operations. Programming is easier when more primitives are available. In any case, programs must be written using only the primitives, not in terms of an array model.

Going by the book, whenever two different definitions are provided, a proof of equivalence is required. However, informal definitions are given to ferment understanding. The intention is to show how to write programs, not to mix the programming examples with technical details.

In the sequel, a program consists of a sequence of equations. Any valid expression matches one of the left-hand sides of the listed definitions. For the sake of completeness, conciseness and definiteness, any valid expression should match precisely one of the left-hand sides. However, for optimization purposes there should be as much freedom as possible in matching any of the left-hand sides. A variable is not a constant

and may assume any value of the specified type. If no type is specified, a (generic) type variable matches any valid data value. The **universal set** $\mathtt{U}$ is defined to be a set containing all known data values.

5.1 DESTRUCTORS

Next size is programmed in terms of the constructors.

Program size: $\mathcal{A}^* \to \mathbb{N}_0$;

Let $\mathtt{x}$ be a type variable, $\mathtt{k} \in \mathbb{N}$ and $\mathtt{C}, \mathtt{D} \in \mathcal{A}$.

$\text{size}(\mathtt{C} +\!\!+_{\mathtt{k}} \mathtt{D}) = \text{size}(\mathtt{C}) + \text{size}(\mathtt{D})$,
$\text{size}(\text{reax}_{\mathtt{k}}\,\mathtt{C}) = \text{size}(\mathtt{C})$,
$\text{size}(\cdot)(\mathtt{x}) = \mathtt{1}$,
$\text{size}\,\square = \mathtt{0}$.

Apply a "*collapse rule*" which is explained next. Partition the array into hyperplanes via $+\!\!+$. Then remove an unneeded axis via reax. Repeat the process, eventually obtaining $\mathtt{1}$-arrays that are finally subdivided via $+\!\!+$ until all that remains are singletons. The program is well-defined in view of array forms (see previous chapter).

The solution is a homomorphism. The theory provides the following programming template:

$$\text{size} = (/^* +) \circ (* \mathtt{K}_1)\,.$$

In fact, since addition is commutative and associative,

$$\text{size} = \left(\bigsqcup +\right) \circ (* \mathtt{K}_1)\,.$$

In the program of size the types may be inferred. For instance, there should be no doubt about the possibilities for $\mathtt{C}$ or $\mathtt{k}$ in the expression $\text{reax}_{\mathtt{k}}\ \mathtt{C}$, as according to the definition, $\text{reax}_{\mathtt{k}}$ is defined only for $\mathtt{n}$-arrays (not empty arrays) and $\mathtt{k} \in \mathbb{N}$. In addition, in the expression $\mathtt{A} +\!\!+_{\mathtt{k}} \mathtt{B}$, $\mathtt{A}$ and $\mathtt{B}$ must both be $\mathtt{n}$-arrays for some $\mathtt{n} \in \mathbb{N}$, unless one of them equals the empty array. In discussions, it is convenient to drop subscripts and superscripts that appear in programs, e.g., write reax instead of $\text{reax}_{\mathtt{k}}$.

Program length recursively as follows.

Program length: $\mathbb{N} \times \mathcal{A}^* \to \mathbb{N}_0$;

Let $\mathtt{x}$ be a type variable, $\mathtt{k} \in \mathbb{N}$ and $\mathtt{C}, \mathtt{D} \in \mathcal{A}$.

$\text{length}_{\mathtt{k}}(\mathtt{C} +\!\!+_{\mathtt{k}} \mathtt{D}) = \text{length}_{\mathtt{k}}(\mathtt{C}) + \text{length}_{\mathtt{k}}(\mathtt{D})$,
$\text{length}_{\mathtt{k}}\ \text{reax}_{\mathtt{k}}(\mathtt{C}) = \mathtt{1}$,
$\text{length}\ (\cdot)(\mathtt{x}) = \mathtt{1}$,
$\text{length}\ \square = \mathtt{0}$.

The solution is a homomorphism. Another solution is

$$\text{length}_{\mathtt{k}} = (+, \mathtt{K}_1)_{\mathtt{k}}.$$

Notice this operation involves only one field direction; whence, there is no point in looking further than list homomorphisms. The following program for shape uses length.

Program shape : $\mathcal{A}^* \to \mathbb{A}^*_{\mathbb{N}}$;

Let $\mathtt{x}$ be a type variable, $\mathtt{A} \in \mathcal{A}^*$, and $\mathtt{C} \in \mathcal{A}$.

$\text{shape}((\text{reax } \mathtt{C}) +\!\!+ \mathtt{A}) = (\cdot)(1 + \text{length } \mathtt{A}) +\!\!+ \text{shape}(\mathtt{C})$,
$\text{shape}((\cdot)(\mathtt{x}) +\!\!+ \mathtt{A}) = (\cdot)(1 + \text{length } \mathtt{A})$,
$\text{shape}\,\square = \square$.

This program is completely defined since all possible array forms are covered. Observe that the shape of an array is the same as the shape of a row, except for the number of rows, in accordance with the definition of $+\!\!+$. The hyperplane reax $\mathtt{C}$ has only one row. $\mathtt{A}$ has length $\mathtt{A}$ rows. The fact that destructors are useful for programming furnishes evidence that primitives should not be limited to constructors. Next, a recursive characterization of axes is presented.

Program axes : $\mathcal{A}^* \to \mathbb{N}_0$;

Let $\mathtt{x}$ be a type variable, $\mathtt{k} \in \mathbb{N}$, $\mathtt{A} \in \mathcal{A}^*$, and $\mathtt{C} \in \mathcal{A}$.

$\text{axes}((\text{reax}_{\mathtt{k}}\ \mathtt{C}) +\!\!+_{\mathtt{k}} \mathtt{A}) = 1 + \text{axes}(\mathtt{C})$,
$\text{axes}((\cdot)(\mathtt{x}) +\!\!+ \mathtt{A}) = 1$,
$\text{axes}\,\square = 0$.

Any hyperplane of an array necessarily has the same number of axes as the entire array. Each time an axis is removed, one is added to the number of axes. A 1-array has one axis. Array forms guarantee that all arrays are programmed. Next a recursive characterization of index is given.

Program index : $\mathcal{A}_{\mathtt{Y}} \times \mathbb{A}_{\mathbb{N}} \to \mathtt{Y}$;

Let $\mathtt{x}$ be a type variable, $\mathtt{n} \in \mathbb{N}$, $\mathtt{I} \in \mathbb{A}^*_{\mathbb{N}}$, $\mathtt{A} \in \mathcal{A}^*$, and $\mathtt{C} \in \mathcal{A}$.

$\text{index}((\text{reax } \mathtt{C}) +\!\!+ \mathtt{A})((\cdot)(\mathtt{n}+1) +\!\!+ \mathtt{I}) = \text{index}\,\mathtt{A}((\cdot)(\mathtt{n}) +\!\!+ \mathtt{I})$,
$\text{index}((\text{reax } \mathtt{C}) +\!\!+ \mathtt{A})((\cdot)(1) +\!\!+ \mathtt{I}) = \text{index } \mathtt{C}\ \mathtt{I}$,
$\text{index}((\cdot)(\mathtt{x}) +\!\!+ \mathtt{A})(\cdot)(\mathtt{n}+1) = \text{index } \mathtt{A}\ (\cdot)(\mathtt{n})$,
$\text{index}((\cdot)(\mathtt{x}) +\!\!+ \mathtt{A})\ (\cdot)(1) = \mathtt{x}$.

Every n-array that is not a 1-array can be expressed as reax C ⧺ A. The first index specifies the row number, which means "hyperplane down" to the correct row. When the right hyperplane is reached, only that hyperplane is needed; whence, all other hyperplanes may be discarded. In addition, after reaching the desired hyperplane, collapse and discard the index that is no longer useful. Each time the valency is reduced (i.e., $\text{collapse}_k \circ \text{reax}_k\, C = C$), delete the corresponding index. Observe that all types of generating relations were used in the program above, which implies the number of constructors is not too large. A recursive characterization of block is presented next.

Program block : $\mathbb{A}_{\mathbb{Z}}^* \times \mathbb{A}_{\mathbb{N}}^* \times \mathcal{A}^* \to \mathcal{A}^*$;

Let $n, m, k \in \mathbb{N}$, $S, U, A \in \mathcal{A}^*$, and $C \in \mathcal{A}$.

```
block S U A = blocks (axes A) 1 S U A,

blocks 0 k S U A = A,
blocks n k S U □ = □,
blocks n k (·)(m) ⧺ S U ((reax_k C) ⧺_k A)
                = blocks n k (·)(m−1) ⧺ S U A,
blocks n k (·)(m) ⧺ S U ((·)(x) ⧺ A)
                = blocks n k (·)(m−1) ⧺ S U A,
blocks n k ((·)(0) ⧺ S) U C = blocku n k S U C,
blocks n k □ U C = blocku n k □ U C,

blocku n k S ((·)(m) ⧺ U) A
                = blocks (n−1) (k+1) S U (blockx k m A),
blocku n k S ((·)(0) ⧺ U) A = □,
blocku n k S □ A = blocks (n−1) (k+1) S □ A,

blockx k m □ = □,
blockx k 0 A = □,
blockx k m((reax_k C) ⧺_k A) = (reax_k C) ⧺_k (blockx k (m−1) A),
blockx k m((·)(x) ⧺ A) = ((·)(x)) ⧺ (blockx k (m−1) A).
```

To extract a block, proceed axis by axis in the natural order. Initially store the valency n. If the given array is empty or if all axes were processed already, in which case $n = 0$, return the array which is the argument from which a block was to be extracted. There are two cyclic phases: one shifting and one taking.

In the first phase, shift over to the desired hyperplane in field direction k using blocks. Drop all initial hyperplanes according to the specified shift $m > 0$. If $m = 0$ or $S = \square$ then skip no hyperplanes. It is possible m is too large and then blocks returns an empty array. By the known forms of an array, it suffices to consider two forms for the argument whenever it is nonempty. All recursive calls of blocks

subtract one from the number of hyperplanes. Hence, the recursion must eventually end.

In the second phase, use two subprograms to take the specified number of hyperplanes `m`. The first subprogram `blocku` checks three possibilities. If the initial number of hyperplanes to take is zero, then `blocku` returns an empty array (which consists of zero hyperplanes). In the case `U` is empty, then the subprogram returns to the first phase with the array `A` from which the result is to be extracted, reduces the number of axes remaining to be checked to $\mathtt{n}-1$, and updates the new field direction to $\mathtt{k}+1$. The only other possibility (ignoring $\mathtt{m}<0$) is that the number of hyperplanes to be taken is $\mathtt{m}>0$. In this case, the subprogram `blocku` again returns to the first phase utilizing the subprogram `blockx`, that is called upon to take the first `m` hyperplanes in field direction `k`. If $\mathtt{m}=0$ or $\mathtt{A}=\square$ then `blockx` returns an empty array. Otherwise the subprogram recursively stacks the specified number of hyperplanes or elements.

Next program $\mathrm{take}_{\mathtt{k}}$, assuming `B`, which specifies the hyperplanes to take, has only positive indices in order.

Program take : $\mathbb{N}\times\mathbb{A}^*_{\mathbb{Z}}\times\mathcal{A}^*\to\mathcal{A}^*$;

Let `x` be a type variable, $\mathtt{k}\in\mathbb{N}$ be constant, $\mathtt{m},\mathtt{n}\in\mathbb{N}$, $\mathtt{C}\in\mathcal{A}$, $\mathtt{B}\in\mathbb{A}^*_{\mathbb{N}}$ and $\mathtt{A}\in\mathcal{A}^*$.

$\mathrm{take}_{\mathtt{k}}\ \mathtt{B}\ \mathtt{A}=\mathtt{takes}_{\mathtt{k}}\ 1\ \mathtt{B}\ \mathtt{A}$,

$\mathtt{takes}_{\mathtt{k}}\ \mathtt{m}\ \mathtt{B}\ \square=\square$,
$\mathtt{takes}_{\mathtt{k}}\ \mathtt{m}\ \square\ \mathtt{C}=\square$,
$\mathtt{takes}_{\mathtt{k}}\ \mathtt{m}\ (\cdot)(\mathtt{m})\ +\!\!+_{\mathtt{B}}\ ((\mathrm{reax}_{\mathtt{k}}\ \mathtt{C})\ +\!\!+_{\mathtt{k}}\mathtt{A})$
$\quad=(\mathrm{reax}_{\mathtt{k}}\ \mathtt{C})\ +\!\!+_{\mathtt{k}}\ \mathtt{takes}_{\mathtt{k}}\ (\mathtt{m}+1)\ \mathtt{B}\ \mathtt{A}$,
$\mathtt{takes}_{\mathtt{k}}\ \mathtt{m}\ (\cdot)(\mathtt{m}+\mathtt{n})\ +\!\!+_{\mathtt{B}}\ ((\mathrm{reax}_{\mathtt{k}}\ \mathtt{C})\ +\!\!+_{\mathtt{k}}\mathtt{A})$
$\quad=\mathtt{takes}_{\mathtt{k}}\ \mathtt{m}+1\ ((\cdot)(\mathtt{m}+\mathtt{n})\ +\!\!+_{\mathtt{B}}\ \mathtt{B})\ \mathtt{A}$,
$\mathtt{takes}_{\mathtt{k}}\ \mathtt{m}\ (\cdot)(\mathtt{m})\ +\!\!+_{\mathtt{B}}\ ((\cdot)(\mathtt{x})\ +\!\!+\mathtt{A})=(\cdot)(\mathtt{x})\ +\!\!+\mathtt{takes}_{\mathtt{k}}\ \mathtt{m}+1\ \mathtt{B}\ \mathtt{A}$,
$\mathtt{takes}_{\mathtt{k}}\ \mathtt{m}\ (\cdot)(\mathtt{m}+\mathtt{n})\ +\!\!+_{\mathtt{B}}\ ((\cdot)(\mathtt{x})\ +\!\!+\mathtt{A})$
$\quad=\mathtt{takes}_{\mathtt{k}}\ (\mathtt{m}+1)\ ((\cdot)(\mathtt{m}+\mathtt{n})\ +\!\!+_{\mathtt{B}}\ \mathtt{B})\ \mathtt{A}$.

In the preceding program of take, process all hyperplanes in the natural order. Keep track of the number `m` of the current hyperplane and compare it to the next index of the hyperplane that should be taken, which is specified in `B`. If the hyperplane should be taken, then include it and dispose of the specified index that has been used; otherwise, keep the index as the desired hyperplane has not yet been reached but discard the hyperplane that should not be taken. Next, a recursive characterization of drop is given, assuming `B`, which specifies the hyperplanes to drop, has only positive indices in order.

Program drop : $\mathbb{N} \times \mathbb{A}_{\mathbb{Z}}^{*} \times \mathcal{A}^{*} \to \mathcal{A}^{*}$;

Let $\mathtt{x}$ be a type variable, $\mathtt{k} \in \mathbb{N}$ be constant, $\mathtt{m}, \mathtt{n} \in \mathbb{N}$, $\mathtt{C} \in \mathcal{A}$, $\mathtt{B} \in \mathbb{A}_{\mathbb{N}}^{*}$ and $\mathtt{A} \in \mathcal{A}^{*}$.

$$\mathrm{drop}_{\mathtt{k}}\ \mathtt{B}\ \mathtt{A} = \mathtt{drops}_{\mathtt{k}}\ 1\ \mathtt{B}\ \mathtt{A},$$

$$\mathtt{drops}_{\mathtt{k}}\ \mathtt{m}\ \mathtt{B}\ \square = \square,$$
$$\mathtt{drops}_{\mathtt{k}}\ \mathtt{m}\ \square\ \mathtt{C} = \mathtt{C},$$
$$\mathtt{drops}_{\mathtt{k}}\ \mathtt{m}\ (\cdot)(\mathtt{m}+\mathtt{n}) +\!\!+ \mathtt{B}((\mathrm{reax}_{\mathtt{k}}\ \mathtt{C}) +\!\!+_{\mathtt{k}} \mathtt{A}) = (\mathrm{reax}_{\mathtt{k}}\ \mathtt{C}) +\!\!+_{\mathtt{k}} \mathtt{drops}_{\mathtt{k}}\ (\mathtt{m}+1)\ (\cdot)(\mathtt{m}+\mathtt{n}) +\!\!+ \mathtt{B}\ \mathtt{A},$$
$$\mathtt{drops}_{\mathtt{k}}\ \mathtt{m}\ (\cdot)(\mathtt{m}) +\!\!+ \mathtt{B}((\mathrm{reax}_{\mathtt{k}}\ \mathtt{C}) +\!\!+_{\mathtt{k}} \mathtt{A}) = \mathtt{drops}_{\mathtt{k}}\ (\mathtt{m}+1)\ \mathtt{B}\ \mathtt{A},$$
$$\mathtt{drops}_{\mathtt{k}}\ \mathtt{m}\ (\cdot)(\mathtt{m}+\mathtt{n}) +\!\!+ \mathtt{B}\ ((\cdot)(\mathtt{x}) +\!\!+ \mathtt{A}) = (\cdot)(\mathtt{x}) +\!\!+ \mathtt{drops}_{\mathtt{k}}\ (\mathtt{m}+1)\ (\cdot)(\mathtt{m}+\mathtt{n}) +\!\!+ \mathtt{B}\ \mathtt{A},$$
$$\mathtt{drops}_{\mathtt{k}}\ \mathtt{m}\ (\cdot)(\mathtt{m}) +\!\!+ \mathtt{B}\ ((\cdot)(\mathtt{x}) +\!\!+ \mathtt{A}) = \mathtt{drops}_{\mathtt{k}}\ (\mathtt{m}+1)\ \mathtt{B}\ \mathtt{A}.$$

In the preceding program, process all hyperplanes in the natural order. Keep track of the number of the current hyperplane $\mathtt{m}$ and compare it to the next index of the hyperplane which should be dropped, which is specified in $\mathtt{B}$. If the hyperplane should be dropped, then dispose of it and the specified index which is now in the way. Otherwise, keep both the index and the hyperplane as an unwanted hyperplane has not yet been found.

Compare Definition 3.26 and program drop. Starting from the specification of the problem to drop hyperplanes, it is clear by inspection that the required task is performed via the program. In addition, not knowing the specification of the problem, it is fairly easy to figure out what happens using the program. This advantage is one of the benefits of using an algebraic theory.

5.2 PAIRING OPERATIONS

Consider familiar pairing operations, which combine or pair corresponding entries of two arrays. Such operations are needed to perform common pointwise operations. Such common operations have been included in the set of primitives.

5.2.1 ZIP

It is natural to extend operations on basic data types to arrays. For instance, $+$ is defined for integers and this operation is extended in the usual way to compute the sum of two arrays whose entries are integers. More generally, suppose $\oplus$ is any binary operation defined over some homogeneous set and written in infix form. Next $\underset{\longleftrightarrow}{\oplus}$ is programmed using recursion written in infix form.

Program $\underset{\longleftrightarrow}{\ }: \{X \times X \to Y\} \times \mathcal{A}_X^* \times \mathcal{A}_X^* \to \mathcal{A}_Y^*$;

Let $\oplus \in \{X \times X \to Y\}$, $x_1, x_2 \in X$, $k \in \mathbb{N}$, $C, D \in \mathcal{A}_X$, and $A, B \in \mathcal{A}_X^*$.

$$\square \underset{\longleftrightarrow}{\oplus} \square = \square,$$
$$((\cdot)(x_1) +\!\!+ A) \underset{\longleftrightarrow}{\oplus} ((\cdot)(x_2) +\!\!+ B) = (\cdot)(x_1 \oplus x_2) +\!\!+ \left(A \underset{\longleftrightarrow}{\oplus} B\right),$$
$$((\mathrm{reax}_k C) +\!\!+_k A) \underset{\longleftrightarrow}{\oplus} ((\mathrm{reax}_k D) +\!\!+_k B) = \mathrm{reax}_k \left(C \underset{\longleftrightarrow}{\oplus} D\right) +\!\!+_k \left(A \underset{\longleftrightarrow}{\oplus} B\right).$$

This operation is defined only for arrays with the same shape. In particular, $A \underset{\longleftrightarrow}{\oplus} B$ is undefined whenever $A = \square$ and $B \neq \square$ or $A \neq \square$ and $B = \square$. Since all possible array forms are considered, every array is programmed. Observe that

$$\underset{\longleftrightarrow}{\oplus} = (* \oplus) \circ \mathtt{pair},$$

where `pair` forms an array of pairs of elements from two given arrays. This function is known as `zip` in functional programming (see for instance Axford and Joy, 1993).

EXAMPLE 5.1 Pair two 2-arrays to obtain a 2-array of pairs:

$$\left(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}\right) \to \begin{bmatrix} (1,5) & (2,6) \\ (3,7) & (4,8) \end{bmatrix}$$

Notice `pair` is algebraically an extension of the `identity` function on pairs of elements

$$\mathtt{id}(a,b) = (a,b)$$

to pairs of arrays, i.e.,

$$\mathtt{pair} = \underset{\longleftrightarrow}{\mathtt{id}}$$

whence

$$\underset{\longleftrightarrow}{\oplus} = (* \oplus) \circ \underset{\longleftrightarrow}{\mathtt{id}}.$$

5.2.2 DISTRIBUTE FUNCTIONS

Closely related to `pair` are two functions, namely `distl` and `distr` (Backus, 1978). They deal mainly with structure. They are defined on a pair, say an object `x` and an array `A`.

DEFINITION 5.1 Declare

$$\mathtt{distl}(x, \square) = \square,$$

and for an arbitrary `n`-array $[a|T]$,

$$\mathtt{distl}\ (x, [a|T]) = [(\mathfrak{Y} \mapsto x) \circ a | T],$$

where

$$(\curvearrowright \mathtt{x})\,\mathtt{a}(\mathtt{t}) = (\mathtt{x}, \mathtt{a}(\mathtt{t})).$$

EXAMPLE 5.2 Pair each element of the 2-array $\mathtt{A} = [\mathtt{a} : 2, 4] \sim \left[\begin{smallmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{smallmatrix}\right]$ with the number 9 via

$$\mathtt{distl}(9, \mathtt{A}) \sim \begin{bmatrix} (9,1) & (9,2) & (9,3) & (9,4) \\ (9,5) & (9,6) & (9,7) & (9,8) \end{bmatrix}.$$

Next distl is programmed.

Program $\mathtt{distl} : \mathtt{U} \times \mathcal{A}^* \to \mathcal{A}^*$;

Let x and y be type variables, $\mathtt{k} \in \mathbb{N}$, $\mathtt{C} \in \mathcal{A}$, and $\mathtt{A} \in \mathcal{A}^*$.

$\mathtt{distl}\ \mathtt{x}\ \square = \square,$
$\mathtt{distl}\ \mathtt{x}\ (\cdot)(\mathtt{y}) +\!\!+\ \mathtt{A} = (\cdot)(\mathtt{x}, \mathtt{y}) +\!\!+\ (\mathtt{distl}\ \mathtt{x}\ \mathtt{A}),$
$\mathtt{distl}\ \mathtt{x}\ (\mathrm{reax}_{\mathtt{k}}\ \mathtt{C}) +\!\!+_{\mathtt{k}}\ \mathtt{A} = (\mathrm{reax}_{\mathtt{k}}\ \mathtt{distl}\ \mathtt{x}\ \mathtt{C}) +\!\!+_{\mathtt{k}} (\mathtt{distl}\ \mathtt{x}\ \mathtt{A}).$

DEFINITION 5.2 Declare

$$\mathtt{distr}(\mathtt{x}, \square) = \square,$$

and for an arbitrary n-array $[\mathtt{a} \mid \mathtt{T}]$,

$$\mathtt{distr}\ (\mathtt{x}, [\mathtt{a} \mid \mathtt{T}]) = [(\leftharpoonup \mathtt{x}) \circ \mathtt{a} \mid \mathtt{T}],$$

where

$$(\leftharpoonup \mathtt{x})\,\mathtt{a}(\mathtt{t}) = (\mathtt{a}(\mathtt{t}), \mathtt{x}).$$

The function distr is programmed via recursive equations.

Program $\mathtt{distr} : \mathtt{U} \times \mathcal{A}^* \to \mathcal{A}^*$;

Let x and y be type variables, $\mathtt{k} \in \mathbb{N}$, $\mathtt{C} \in \mathcal{A}$, and $\mathtt{A} \in \mathcal{A}^*$.

$\mathtt{distr}\ \mathtt{x}\ \square = \square,$
$\mathtt{distr}\ \mathtt{x}\ ((\cdot)(\mathtt{y}) +\!\!+ \mathtt{A}) = (\cdot)(\mathtt{y}, \mathtt{x}) +\!\!+\ (\mathtt{distr}\ \mathtt{x}\ \mathtt{A}),$
$\mathtt{distr}\ \mathtt{x}\ ((\mathrm{reax}_{\mathtt{k}}\ \mathtt{C}) +\!\!+_{\mathtt{k}} \mathtt{A}) = (\mathrm{reax}_{\mathtt{k}}\ \mathtt{distr}\ \mathtt{x}\ \mathtt{C}) +\!\!+_{\mathtt{k}}\ \mathtt{distr}\ \mathtt{x}\ \mathtt{A}.$

Actually, distl x and distr x are simple specialized maps:

$$\mathtt{distl}\ \mathtt{x} = *(\curvearrowright \mathtt{x})$$

and

$$\mathtt{distr}\ \mathtt{x} = *(\leftharpoonup \mathtt{x}).$$

Although it is unnecessary to add distl and distr as primitives, it is reasonable to account for them in optimization.

5.3 PARTITIONING OPERATIONS

Consider familiar partitioning operations. Such operations do not perform computations. They select portions of the data structure or partition it.

5.3.1 SPLIT

Next a recursive characterization is presented for split, which is a primitive operation.

Program split: $\mathbb{N} \times \mathbb{Z} \times \mathcal{A}^* \to \mathcal{A}^* \times \mathcal{A}^*$;

Let x be a type variable, $k \in \mathbb{N}$ be constant, $p \in \mathbb{N}$, $q \in \mathbb{Z}$, $C \in \mathcal{A}$, and $A \in \mathcal{A}^*$.

$\text{split}_k\ q\ A = (\text{spl}\ k\ q\ A, \text{spr}\ k\ q\ A)$,

$\text{spl}\ k\ p\ \square = \square$,
$\text{spl}\ k\ 0\ A = \square$,
$\text{spl}\ k\ (-p)\ A = \square$,
$\text{spl}\ k\ p\ (\text{reax}_k\ C) +\!\!+_k A = (\text{reax}_k\ C) +\!\!+_k \text{spl}\ k\ (p-1)\ A$,
$\text{spl}\ 1\ p\ (\cdot)(x) +\!\!+ A = (\cdot)(x) +\!\!+ \text{spl}\ 1\ (p-1)\ A$,

$\text{spr}\ k\ p\ \square = \square$,
$\text{spr}\ k\ 0\ A = A$,
$\text{spr}\ k\ (-p)\ A = A$,
$\text{spr}\ k\ p\ (\text{reax}_k\ C) +\!\!+_k A = \text{spr}\ k\ (p-1)\ A$,
$\text{spr}\ 1\ p\ (\cdot)(x) +\!\!+ A = \text{spr}\ 1\ (p-1)\ A$.

The program for split is partitioned as the argument itself is partitioned. The subprogram spl takes only the first p hyperplanes in field direction k. The subprogram spr drops the first p hyperplanes and returns the rest. Evidently, spl and spr may be executed concurrently.

5.3.2 BORDER

A recursive characterization of border is given next.

Program border: $\mathbb{A}^*_{\mathbb{N}_0} \times \mathcal{A}^* \to \mathcal{A}^*$;

Let x and y be type variables, $k, m, n \in \mathbb{N}$, $C \in \mathcal{A}$, $B \in \mathcal{A}^*_{\mathbb{N}_0}$, and $A \in \mathcal{A}^*$.

$\text{border}\ B\ A = \text{bord}\ 1\ B\ A$,

$\text{bord}\ k\ \square\ A = A$,
$\text{bord}\ k\ B\ \square = \square$,
$\text{bord}\ k\ ((\cdot)(m) +\!\!+ B)\ C = \text{bordl}\ k\ m\ m\ B\ C$,
$\text{bord}\ k\ ((\cdot)(0) +\!\!+ B)\ C = \text{bord}\ (k+1)\ B\ C$,

$\mathtt{bordl\ k\ n\ m\ B}((\mathrm{reax}_{\mathtt{k}}\,\mathtt{C}) +\!\!+_{\mathtt{k}} \mathtt{A})$
$\quad = (\mathrm{reax}_{\mathtt{k}}\,\mathtt{C}) +\!\!+_{\mathtt{k}} \mathtt{bordl\ k\ (n-1)\ m\ B}\ ((\mathrm{reax}_{\mathtt{k}}\,\mathtt{C}) +\!\!+_{\mathtt{k}} \mathtt{A}),$
$\mathtt{bordl\ k\ 0\ m\ B}\ (\mathtt{A} +\!\!+_{\mathtt{k}} (\mathrm{reax}_{\mathtt{k}}\,\mathtt{C}))$
$\quad = \mathtt{bord\ (k+1)\ B}\ ((\mathtt{A} +\!\!+_{\mathtt{k}} (\mathrm{reax}_{\mathtt{k}}\,\mathtt{C})) +\!\!+_{\mathtt{k}} \mathtt{bordr\ k\ m}\ (\mathrm{reax}_{\mathtt{k}}\,\mathtt{C})),$
$\mathtt{bordl\ 1\ n\ m\ B}\ ((\cdot)(\mathtt{x}) +\!\!+ \mathtt{A})$
$\quad = (\cdot)(\mathtt{x}) +\!\!+ \mathtt{bordl\ 1\ (n-1)\ m\ B}\ ((\cdot)(\mathtt{x}) +\!\!+ \mathtt{A}),$
$\mathtt{bordl\ 1\ 0\ m\ B}\ (\mathtt{A} +\!\!+ (\cdot)(\mathtt{y})) = (\mathtt{A} +\!\!+ (\cdot)(\mathtt{y})) +\!\!+ \mathtt{bordr\ 1\ m}\ (\cdot)(\mathtt{y}),$

$\mathtt{bordr\ k\ 0\ B} = \square,$
$\mathtt{bordr\ k\ m\ B} = \mathtt{B} +\!\!+_{\mathtt{k}} \mathtt{bordr\ k\ (m-1)\ B}.$

The program is divided into three subprograms, namely `bord`, `bordl` and `bordr`. Subprogram `bord` takes care of the trivial cases, keeping track of field directions, and if a border is needed for the `k`-axis then `bordl` is called upon to continue processing. If `bord` finds that zero entries are to be duplicated for the `k`-axis, then a recursive call is made, processing the next field direction `k+1`. Note either `B` is empty or `B` contains nonnegative integers.

Subprogram `bordl` adds only the left border and when the job is done, a recursive call to `bord` is made using subprogram `bordr` to append the right border, unless the argument is a `1`-array in which case recursion is halted as there can be no more axes to process and the remainder of the argument is ignored (which may be useful). Note the third parameter `m` for `bordl` is the number of hyperplanes that must be appended. This number does not change (an invariant). Subprogram `bordr` receives only the border that is duplicated, along with the field direction `k` and the number of copies `m` required.

5.3.3 UNBORDER

Next unborder is programmed.

Program unborder: $\mathbb{A}^*_{\mathbb{N}_0} \times \mathcal{A}^* \to \mathcal{A}^*$;

Let `x` be a type variable, $\mathtt{k, m, n} \in \mathbb{N}$, $\mathtt{B} \in \mathbb{A}^*_{\mathbb{N}_0}$, $\mathtt{A} \in \mathcal{A}^*$ and $\mathtt{C} \in \mathcal{A}$.

unborder $\mathtt{B\ A} = \mathtt{unbord\ 1\ B\ A},$

$\mathtt{unbord\ k}\ \square\ \mathtt{A} = \mathtt{A},$
$\mathtt{unbord\ k}\ (\cdot)(\mathtt{m}) +\!\!+ \mathtt{B\ A} = \mathtt{unbordl\ k\ m\ m\ B\ A},$
$\mathtt{unbord\ k}\ (\cdot)(0) +\!\!+ \mathtt{B\ A} = \mathtt{unbord\ (k+1)\ B\ A},$

$\mathtt{unbordl\ k\ n\ m\ B}\ \square = \square,$
$\mathtt{unbordl\ k\ n\ m\ B}\ (\mathrm{reax}_{\mathtt{k}}\,\mathtt{C}) +\!\!+_{\mathtt{k}} \mathtt{A} = \mathtt{unbordl\ k\ (n-1)\ m\ B\ A},$
$\mathtt{unbordl\ 1\ n\ m\ B}\ (\cdot)(\mathtt{x}) +\!\!+ \mathtt{A} = \mathtt{unbordl\ 1\ (n-1)\ m\ B\ A},$
$\mathtt{unbordl\ k\ 0\ m\ B\ A} = \mathtt{unbord\ (k+1)\ B\ (unbordr\ k\ m\ A)},$

```
unbordr k 0 A = A,
unbordr k m □ = □,
unbordr k m (A ⌿k (reaxk C)) = unbordr k (m − 1) A,
unbordr k m (A ⌿ (·) (x)) = unbordr k (m − 1) A.
```

The program is divided into three subprograms, namely `unbord`, `unbordl` and `unbordr`. Subprogram `unbord` takes care of the trivial cases and if a border is to be removed for the `k`-axis then `unbordl` is called upon to continue processing. The field direction `k` along with the number `m` of hyperplanes to drop (repeated twice, once for each side) and the remaining border array which is nonempty is passed to `unbordl`.

Subprogram `unbordl` strips away the left border and then a call to `unbord` is made using subprogram `unbordr` to remove the right border. Note `unbordl` is passed a third parameter which is always a natural number (invariant). Another invariant is that the field direction `k` is unaltered by recursive calls to this subprogram. In addition, the number `n` of remaining hyperplanes to be dropped (second parameter) is nonnegative.

Subprogram `unbordr` receives the remaining array along with the field direction `k` and the number `m` of hyperplanes that will be removed. This subprogram removes the last hyperplane, subtracts one from the number left to remove and continues until either there are no more hyperplanes to remove or there is nothing left of the argument. By inspection of array forms, all cases are covered.

5.4 TRANSFORMATIONS

Next, the following common transformations are programmed: reshape, flatten, reverse, rotate and transpose. These operations restructure their argument, i.e., data is rearranged, not changed.

5.4.1 RAVEL

Another familiar function from APL (`,`) which is included in *A mathematics of arrays* (Mullin, 1988) is `ravel`. It flattens an `n`-array into a `1`-array. This function is defined for all arrays and relies solely on structural aspects.

EXAMPLE 5.3 A squared `2`-array

$$N_3 \sim \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$ is raveled into the `1`-array $\sim [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]$.

EXAMPLE 5.4 Using `1`-order, a `3`-array may be flattened as depicted in Figure 5.1.

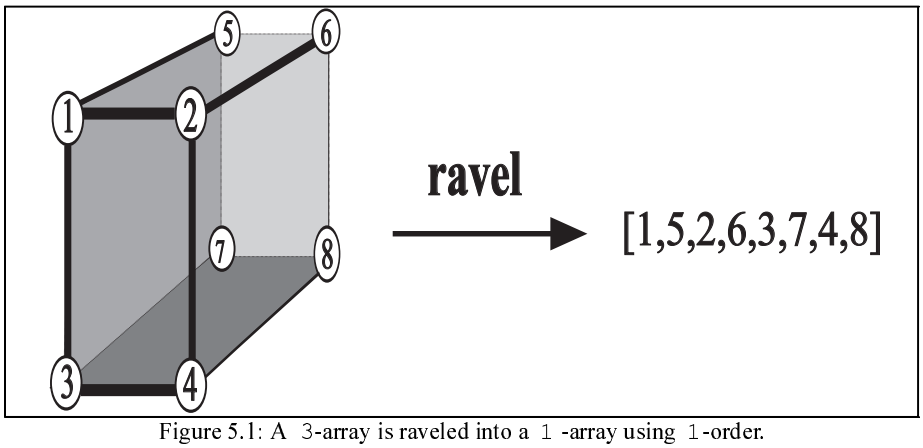


Figure 5.1: A `3`-array is raveled into a `1` -array using `1`-order.

Program ravel : $\mathcal{A}^* \to \mathbb{A}^*$;

Let `x` be a type variable, `k` $\in \mathbb{N}$, and `C, D` $\in \mathcal{A}$.

```
ravel (C ++ D) = (ravel C) ++ (ravel D),
ravel (reax C) = ravel C,
ravel (·) (x) = (·) (x),
ravel □ = □.
```

Apply the collapse rule, always starting with the first axis. Any `n`-array can be expressed as a concatenation `A++B` since every `n`-array has a `1`-axis. Invoking Lemma 4.15, if `n` > 1, then a `n`-array may be decomposed into rows which can be expressed in terms of reax.

Evidently `ravel` is a homomorphism by inspection of the program. It turns out that ravel is the total homomorphism

$$\texttt{ravel} = (\backsim +\!\!+) \circ (* \ (\cdot)).$$

Note `ravel` □ is implicitly defined since □ is the identity element for $+\!\!+$.

5.4.2 RESHAPE

Reshape is a powerful restructuring operation in APL (ρ). The operation is included in *A mathematics of arrays* (Mullin, 1988). Reshape is available in popular mathematical software such as Maple. This operation constructs a new array with the given shape taking the entries from the given array in row order repeatedly until the new array is full. A closely related notion is flattening.

EXAMPLE 5.5 Reshape a 2×3 matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \text{into the } 3 \times 2 \text{ matrix} \quad \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}.$$

DEFINITION 5.3 Informally, `reshape S A` reshapes the second argument using its content and the given shape, which is the first argument. If $\mathtt{S} = \square$ or $\mathtt{A} = \square$ then

$$\mathtt{reshape\ S\ A} = \square.$$

Otherwise for an arbitrary array $\mathtt{A} = [\mathtt{a} \mid \mathtt{T}]$ and shape array $\mathtt{S} = [\mathtt{s} : \mathtt{m}] \in \mathbb{A}_{\mathbb{N}}$, define

$$\mathtt{reshape\ S\ A} = [\mathtt{c} \mid (\mathtt{s} : \mathtt{m})],$$

and

$$\mathtt{c}(\mathtt{i} : \mathtt{m}) = \mathtt{a}\left(\phi^{-1}\left(((\varphi(\mathtt{i} : \mathtt{m}) - 1) \bmod (\text{size } \mathtt{A})) + 1\right)\right)$$

where φ is given by

$$\varphi(\mathtt{i} : \mathtt{m}) = (((\mathtt{i}_1 - 1)\,\mathtt{s}_2 + (\mathtt{i}_2 - 1))\,\mathtt{s}_3 + \cdots + (\mathtt{i}_{\mathtt{m}-1} - 1))\,\mathtt{s}_{\mathtt{m}} + \mathtt{i}_{\mathtt{m}},$$

and ϕ yields the lexicographical order of elements of $\mathtt{A}$. Recall, $\phi : \mathbb{P}(\mathtt{T}) \to \iota\ (\text{size } \mathtt{A})$ is the one-to-one map satisfying $\mathtt{p} <_1 \mathtt{q} \Rightarrow \phi(\mathtt{p}) < \phi(\mathtt{q})$.

If $\mathtt{s}_1 = \text{size } \mathtt{A}$ then

$$\mathtt{reshape}\ [\mathtt{s} : 1]\ \mathtt{A} = \mathtt{ravel\ A}.$$

Next a couple auxiliary programs are given. The size determined by a shape array is computed using `shapesize` defined next.

DEFINITION 5.4 For a shape array $\mathtt{S} = [\mathtt{s} : \mathtt{m}]$,

$$\mathtt{shapesize\ S} = \prod_{\mathtt{i}=1}^{\mathtt{m}} \mathtt{s}_{\mathtt{i}}.$$

A program for `shapesize` is given next.

Program shapesize : $\mathbb{A}_{\mathbb{N}}^{*} \to \mathbb{N}$;

Let $\mathtt{n} \in \mathbb{N}$ and $\mathtt{A}, \mathtt{B} \in \mathbb{A}_{\mathbb{N}}$.

```
shapesize □ = 1,
shapesize(A ⊣ B) = (shapesize A) × (shapesize B),
shapesize(·)(n) = n.
```

Notice all list forms are programmed. To determine a value for `shapesize` $\square$, notice

$$\texttt{shapesize A} = \texttt{shapesize}(\texttt{A} +\!\!+ \square) = (\texttt{shapesize A}) \times (\texttt{shapesize} \square).$$

In particular, $\texttt{shapesize} \square = (\texttt{shapesize} \square)^2$ has only two nonnegative solutions: 0 or 1. The former solution is unacceptable since if $\texttt{shapesize} \square = 0$ then $\texttt{shapesize A} = 0$ for all `A`.

The function `shapesize` is a homomorphism. A simpler program is

$$\texttt{shapesize} = /\times.$$

Alternatively,

$$\texttt{shapesize} = \bigsqcup \times.$$

Reshape involves both shape and content. First form a `1`-array which holds all the elements for the final reshaped array. Second, reshape this `1`-array. To form the former `1`-array, `ravel` the given array and then concatenate the raveled array sufficiently many times so that the resulting `1`-array is large enough to be reshaped. Before reshaping, truncate the large `1`-array so that it has exactly the correct size. The program `resize` forms this `1`-array which will subsequently be reshaped.

Program resize : $\mathcal{A}^* \to \mathbb{A}^*$;

Let $\texttt{m,n,p} \in \mathbb{N}$, $\texttt{T} \in \mathbb{A}_{\mathbb{N}}$, $\texttt{S} \in \mathbb{A}_{\mathbb{N}}^*$, $\texttt{A} \in \mathcal{A}^*$ and $\texttt{B} \in \mathcal{A}$.

```
resize □ A = □,
resize S □ = □,
resize T B = resizer (shapesize T) (size B) (ravel B),

resizer m n B = resizeit m (m − n) n B,

resizeit m p n B = resizeit m (m − 2n) 2n B ++ B,
resizeit m 0 n B = initial m B,
resizeit m (−p) n B = initial m B.
```

The subprogram `resizeit` uses a second parameter to keep track of how many more elements are needed. The third parameter for `resizeit` is used to keep track of the current size of the `1`-array. The first parameter for `resizeit` is the required size (an invariant). During each round the length of the string is doubled via concatenation, and the parameters are updated.

It remains to reshape the array, after flattening and resizing.

Program reshape : $\mathbb{A}_{\mathbb{N}}^* \times \mathcal{A}^* \to \mathcal{A}^*$;

Let $\texttt{m} \in \mathbb{N}$, $\texttt{S} \in \mathbb{A}_{\mathbb{N}}^*$, $\texttt{T} \in \mathbb{A}_{\mathbb{N}}$, $\texttt{A} \in \mathcal{A}^*$, and $\texttt{B,C} \in \mathbb{A}$.

$$\begin{aligned}
&\texttt{reshape}\ \square\ \texttt{A} = \square,\\
&\texttt{reshape}\ \texttt{S}\ \square = \square,\\
&\texttt{reshape}\ \texttt{T}\ \texttt{B} = \texttt{reshaper}\ \texttt{T}\ (\texttt{resize}\ \texttt{T}\ \texttt{B}),\\
&\\
&\texttt{reshaper}\ (\cdot)\ (\texttt{m})\ \texttt{B} = \texttt{B},\\
&\texttt{reshaper}\ ((\cdot)\ (\texttt{m}) +\!\!+ \texttt{T})\ \texttt{B} = \texttt{replane}\ (\texttt{shapesize}\ \texttt{T})\ \texttt{T}\ \texttt{B},\\
&\\
&\texttt{replane}\ \texttt{m}\ \texttt{T} = (\texttt{reaxis}\ \texttt{m}\ \texttt{T}) \circ (\text{split}\ \texttt{m}),\\
&\\
&\texttt{reaxis}\ \texttt{m}\ \texttt{T}\ \texttt{B}\ \square = \texttt{reaxial}\ \texttt{T}\ \texttt{B},\\
&\texttt{reaxis}\ \texttt{m}\ \texttt{T}\ \texttt{B}\ \texttt{C} = (\texttt{reaxial}\ \texttt{T}\ \texttt{B}) +\!\!+ (\texttt{replane}\ \texttt{m}\ \texttt{T}\ \texttt{C}),\\
&\\
&\texttt{reaxial}\ \texttt{T}\ \texttt{B} = \text{reax} \circ (\texttt{reshaper}\ \texttt{T}\ \texttt{B}).
\end{aligned}$$

The argument in the definition of `replane` is omitted, which is a preferable way to define operations. Note the argument to `reshaper` and `replane` is a always a `1`-array (invariant). Each time `reshaper` is called again, the shape array `T` has one less element, and termination occurs when the shape array is a singleton since then the argument is required to be a `1` -array which it is always.

The only way to add an axis is in a call to `reshaper`. It is impossible to concatenate before adding an axis as `replane` terminates only via the subprogram `reaxial`. Just as with ordered reductions, only a single concatenation operator is required as concatenation takes place along the newly formed axis, which may be regarded as collapsing in reverse. The fact that it is possible to reshape using mainly the constructors lends evidence that they have been properly chosen.

EXAMPLE 5.6 In order to duplicate the single row of the array

$$\ulcorner_4 \sim \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

apply

$$\texttt{reshape}\ \overline{2,4} \sim \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}.$$

5.4.3 REVERSE

The function `reverse` is another familiar operation in functional programming. This operation may be programmed so that the order of the hyperplanes in field direction `k` is reversed.

Program reverse : $\mathbb{N} \times \mathcal{A}^* \to \mathcal{A}^*$;

Let x be a type variable, $\texttt{k} \in \mathbb{N}$ be constant, $\texttt{A} \in \mathcal{A}^*$ and $\texttt{C} \in \mathcal{A}$.

$$\texttt{reverse}_{\texttt{k}}\ (\texttt{A} +\!\!+_{\texttt{k}} (\text{reax}_{\texttt{k}}\ \texttt{C})) = (\text{reax}_{\texttt{k}}\ \texttt{C}) +\!\!+_{\texttt{k}} (\texttt{reverse}_{\texttt{k}}\ \texttt{A}),$$

$\texttt{reverse}\,\texttt{A} \mathbin{+\!\!+} (\cdot)(\texttt{x}) = (\cdot)(\texttt{x}) \mathbin{+\!\!+} (\texttt{reverse}\,\texttt{A})$,
$\texttt{reverse}_{\texttt{k}}\,\square = \square$.

It turns out that

$$\texttt{reverse}\ [\texttt{a}:\texttt{n}] = \circlearrowleft \texttt{p}\ [\texttt{a}:\texttt{n}],$$

where $\texttt{p}$ is the $\texttt{n}$-permutation given by

$$\texttt{p}_{\texttt{k}} = \texttt{n} - \texttt{k} + 1\,.$$

5.4.4 TRANSPOSE

Consider the operation `transpose` for $\texttt{n}$-arrays which transposes hyperplanes for the $\texttt{j}$-axis into hyperplanes for the $\texttt{i}$-axis.

Program transpose : $\mathbb{N} \times \mathbb{N} \times \mathcal{A}^* \to \mathcal{A}^*$;

Let $\texttt{i},\texttt{j} \in \mathbb{N}$ be constants, $\texttt{A} \in \mathcal{A}^*$ and $\texttt{C} \in \mathcal{A}$.

$\texttt{transpose}\,\texttt{i}\,\texttt{j}\,\square = \square$,
$\texttt{transpose}\,\texttt{i}\,\texttt{j}\,(\texttt{reax}_{\texttt{j}}\ \texttt{C}) \mathbin{+\!\!+_{\texttt{j}}} \texttt{A} = (\texttt{reax}_{\texttt{i}}\ \texttt{C}) \mathbin{+\!\!+_{\texttt{i}}} \texttt{transpose}\,\texttt{i}\,\texttt{j}\,\texttt{A}$.

In the nontrivial case, the program is a single equation that uses various constructors, showing that they are useful. If fewer constructors were provided, programming in this case would be more difficult.

EXAMPLE 5.7 Let

$$[\phi:2,3] \sim \texttt{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.$$

Then

$$\texttt{transpose}\ 1\ 2\ [\phi:2,3] = [\beta:3,2] \sim \texttt{A}^{\texttt{T}} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

5.5 HIGHER-ORDER FUNCTIONS

The following program recursively defines map.

Program $*$: $\{\texttt{X} \to \texttt{Y}\} \times \mathcal{A}^* \to \mathcal{A}^*$;

Let $\texttt{n} \in \mathbb{N}$ be constant, $\texttt{f} \in \{\texttt{X} \to \texttt{Y}\}$ be any defined function, $\texttt{x} \in \texttt{X}$, $\texttt{k} \in \mathbb{N}$ and $\texttt{A},\texttt{B} \in \mathcal{A}$.

$*\,\texttt{f}\,\square = \square$,
$*\,\texttt{f}\,(\cdot)_{\texttt{n}}(\texttt{x}) = (\cdot)_{\texttt{n}}(\texttt{f}(\texttt{x}))$,
$*\,\texttt{f}\,(\texttt{A} \mathbin{+\!\!+_{\texttt{k}}} \texttt{B}) = (*\,\texttt{f}\,\texttt{A}) \mathbin{+\!\!+_{\texttt{k}}} (*\,\texttt{f}\,\texttt{B})$.

It can be shown that this program is equivalent to Definition 3.28 (see proof in Appendix).

Consider the `scan` operation, sometimes called accumulation or prefix-sums. The scan vector model is a data parallel model based on vectors. A wide range of problems can be solved using this model (Blelloch, 1990). It is possible to replace the vector model by an array model and apply the same algorithm. Essentially, for n-arrays with $n > 1$ replace the elements by hyperplanes.

Program $\backslash : \mathbb{N} \times \{X \times X \to X\} \times \mathcal{A}_X^* \to \mathcal{A}_X^*$;

Let $k \in \mathbb{N}$ be constant, $\oplus \in \{X \times X \to X\}$, $x \in X$, $A \in \mathcal{A}^*$ and $C \in \mathcal{A}$.

$$\backslash_k \oplus \square = \square,$$
$$\backslash_k \oplus ((\text{reax}_k\ C) +\!\!+_k A) = (\text{reax}_k\ C) +\!\!+_k \circledast_k (* (\oplus \circ \rightarrowtail)\ (\text{reax}_k\ C))\ \backslash_k \oplus A,$$
$$\backslash \oplus ((\cdot)\ (x) +\!\!+ A) = (\cdot)\ (x) +\!\!+\ * (\oplus \circ \rightarrowtail x)\ (\backslash \oplus A).$$

EXAMPLE 5.8 For a 2-array

$$A \sim \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \qquad \backslash_2 + A \sim \begin{bmatrix} 1 & 3 & 6 \\ 4 & 9 & 15 \end{bmatrix}.$$

5.6 ARRAY PRODUCTS

Consider functions for products, such as inner product (usually associated with a measure known as the norm) and matrix multiplication. The **inner product** of two vectors is programmed using a homomorphism and an abstractor.

Program $\circledcirc$;

Let $\oplus$ be any associative binary operator and $\otimes$ any binary operator.

$$\circledcirc \oplus \otimes = (/\oplus) \circ \underset{\leftrightarrow}{\otimes}.$$

EXAMPLE 5.9 Calculate the inner product

$$\circledcirc + \times \urcorner_3\ \urcorner_3 = 1^2 + 2^2 + 3^2 = 14.$$

EXAMPLE 5.10 Compute

$$\circledcirc \oplus \otimes\ \overline{a, b, c}\ \overline{d, e, f} = (a \otimes d) \oplus (b \otimes e) \oplus (c \otimes f).$$

A single symbol $\circledcirc$ instead of $\circledcirc + \times$ may be used to denote the usual inner product. Closely related to inner product is outer product, which is not the same as cross product or vector product. The **outer product** of two vectors is programmed

using a couple subprograms in addition to primitives and a homomorphism. The only new subprogram `len` is defined using only primitives.

Program $\odot$;

Let $\otimes$ be any binary operator and $A, B \in \mathcal{A}$.

$$\odot\otimes\ A\,B = *_2\left(\underset{\longleftrightarrow}{\otimes}\circ\rightarrowtail(\text{reax}_2\ A)\right)\ (\texttt{reshape}\ (\texttt{len}\ A\ B)\ B),$$

$$\texttt{len}\ A\ B = (\cdot)(\text{length}\ A) + (\cdot)(\text{length}\ B).$$

EXAMPLE 5.11 Compute the outer product

$$\odot\times\ \urcorner_3\ \overline{5,7,11} \sim \begin{bmatrix}1\\2\\3\end{bmatrix}\begin{bmatrix}5 & 7 & 11\end{bmatrix} = \begin{bmatrix}5 & 7 & 11\\ 10 & 14 & 22\\ 15 & 21 & 33\end{bmatrix}.$$

To obtain this result, multiply $\overline{5,7,11}$ by the entries 1, 2, and 3 of $\urcorner_3$ and concatenate as rows in the same order.

EXAMPLE 5.12 Calculate

$$\odot\otimes\ \overline{a,b,c}\,\overline{d,e,f} \sim \begin{bmatrix}a\otimes d & a\otimes e & a\otimes f\\ b\otimes d & b\otimes e & b\otimes f\\ c\otimes d & c\otimes e & c\otimes f\end{bmatrix}.$$

More (1979) defined two powerful functions: **EACHLEFT** and **EACHRIGHT**. Examples include **Kronecker products** which are useful to build-up matrices (Jain, 1989, p. 30).

EXAMPLE 5.13 Consider

$$\left[K^{\delta}\,|\,\complement_2^2\right] \sim \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \qquad \text{and} \qquad A = \left[a\,|\,\complement_2^2\right] \sim \begin{bmatrix}1 & 0\\ 0 & 2\end{bmatrix}.$$

A Kronecker product is obtained by distributing the identity matrix to each element of A and then performing the usual scalar multiplication yielding a nested 2-array B of 2-arrays:

$$B \sim \begin{bmatrix}\begin{bmatrix}1&0\\0&1\end{bmatrix} & \begin{bmatrix}0&0\\0&0\end{bmatrix}\\ \begin{bmatrix}0&0\\0&0\end{bmatrix} & \begin{bmatrix}2&0\\0&2\end{bmatrix}\end{bmatrix}.$$

The **direct product** of two arrays may be defined as in the following program.

Program $\boxtimes$;

Let $\otimes$ be any binary operator and $A, B \in \mathcal{A}$.

$$\boxtimes\otimes A\,B = *(*(\otimes\circ\rightarrowtail)\circ(\leftarrowtail A))\,B.$$

EXAMPLE 5.14 Take the 2-array A in Example 5.13 and compute

$$\boxtimes\times\left[\mathrm{K}^{\delta}:2,2\right]\mathrm{A}$$

which yields the nested 2-array of 2-arrays in the cited example.

Matrix multiplication and related algorithms are well-known. Typically, an algorithm is designed for small matrices and then a tensor product construction yields an algorithm for larger ones (Coppersmith and Winograd, 1990). Many applications are computationally intensive.

Usually, the product of two square matrices A and B is denoted by $\mathrm{C}=\mathrm{AB}$, where

$$\mathrm{c}_{\mathrm{i},\mathrm{j}}=\sum_{\mathrm{k}=0}^{\mathrm{n}}\mathrm{a}_{\mathrm{i},\mathrm{k}}\mathrm{b}_{\mathrm{k},\mathrm{j}}\qquad\text{for all }\mathrm{i},\mathrm{j}.$$

A special case occurs when B is a vector. Matrix-vector products are often associated with a linear system of equations.

Consider a generalized matrix product defined for any two operators, say $\oplus$ and $\otimes$. In the above discussion, $+$ and $\times$ replace $\oplus$ and $\otimes$, respectively.

EXAMPLE 5.15 The all pairs shortest path problem can be solved by substituting min and $+$ for $\oplus$ and $\otimes$, respectively (JáJá, 1992, p. 252).

Let A and B be arbitrary n-arrays. As inner and outer product have already been programmed, assume $\mathrm{n}>1$. Just as operations like reduce are defined axially, an array product may be defined using axes, appealing to the usual notion of matrix multiplication. To this end, consider rows of the left array to be parallel to the 2-axis and the columns of the right array to be parallel to the 1-axis . Although this choice of axes corresponds to the usual multiplication of matrices, an array product could be defined for any pair of axes.

The next definition for **array product** is intended to show how properties of an operation can be expressed in the formalism.

DEFINITION 5.5 In order to define the higher-order partial function

$$*\oplus\otimes,$$

let $\oplus$ be any associative binary operator and $\otimes$ be any binary operator. For all arrays

$E \in \mathcal{A}^{2*}$,

$$\circledast \oplus \otimes\ E\ \square = \square,$$

$$\circledast \oplus \otimes\ \square\ E = \square.$$

Assume $A, B, C, D \in \mathcal{A}^2$ are suitable arrays for the given operators. The following distributive laws hold:

(5.1) $$\circledast \oplus \otimes\ (A \# B)\, C = (\circledast \oplus \otimes\ A\ C) \# (\circledast \oplus \otimes\ B\ C),$$

(5.2) $$\circledast \oplus \otimes\ A (B \#_2 C) = (\circledast \oplus \otimes\ A\ B) \#_2 (\circledast \oplus \otimes\ A\ C).$$

In addition, if

$$\text{length}_k\ A = \text{length}_k\ C \quad \text{and} \quad i \notin \{1, 2\}$$

then

(5.3) $$\circledast \oplus \otimes\ (A \#_k B)(C \#_k D) = (\circledast \oplus \otimes\ A\ C) \#_k (\circledast \oplus \otimes\ B\ D).$$

Letting $\circledcirc$ denote inner product, if

$$\text{length}_k\ A = 1 \quad \text{for all } k \neq 2,$$

and

$$\text{length}_k\ B = 1 \quad \text{for all } k \neq 1,$$

then

(5.4) $$\circledast \oplus \otimes\ A\ B = \circledcirc \oplus \otimes \left(\text{ax}_2\ A\ \overline{\mathbb{C}^\infty}\right) \left(\text{ax}_1\ B\ \overline{\mathbb{C}^\infty}\right).$$

What sizes are possible? For matrices, the number of columns in the left matrix must equal the number of rows in the right matrix. Similar rules hold for array products. This requirement is implicit in the definition.

Consider the usual properties of matrix multiplication. Equation (5.1) reveals that rows in the resulting array are computed using corresponding rows from the left array together with the right array. Similarly, equation (5.2) shows that columns in the resulting array are computed using corresponding columns from the right array together with the left array. Equation (5.3) shows that a block in the resulting array is computed using corresponding blocks from the left and right arrays, whenever the arrays are partitioned in a different field direction k with $k \neq 1, 2$. Equation (5.4) reveals that an element in the resulting array is computed using appropriate axes from the left and right arrays.

Program $\divideontimes$;

Let $\mathtt{p} \in \mathbb{N}$, $\mathtt{A,B,C,D} \in \mathcal{A}^2$, $\mathtt{E} \in \mathcal{A}^{2*}$, and $\mathtt{F,G} \in \mathbb{A}$.

$$\divideontimes \oplus\otimes \mathtt{E}\ \square = \square,$$
$$\divideontimes \oplus\otimes \square\ \mathtt{E} = \square,$$
$$\divideontimes \oplus\otimes \mathtt{A\,B} = \text{untile} \circ * (\mathtt{multcol} \oplus\otimes \mathtt{A})\ (\text{cover}\ \square\ \overline{2}\ \urcorner_\infty\ \mathtt{B}),$$

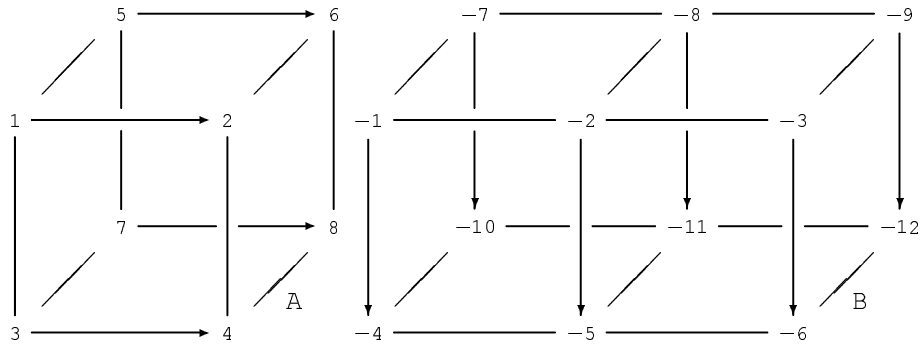
$$\mathtt{multcol} \oplus\otimes \mathtt{A\,B} = \text{untile} \circ * (\mathtt{multrow} \oplus\otimes \mathtt{B})\ (\text{cover}\ \square\ \overline{1}\ \urcorner_\infty\ \mathtt{A}),$$

$$\mathtt{multrow} \oplus\otimes \mathtt{A\,B} = \mathtt{splitplane} \oplus\otimes \mathtt{B\,A},$$

$$\begin{aligned}\mathtt{splitplane} \oplus\otimes\ & ((\text{reax}_{p+2}\,\mathtt{A}) \dashv_{p+2} \mathtt{C})\ ((\text{reax}_{p+2}\,\mathtt{B}) \dashv_{p+2} \mathtt{D}) \\ & = (\mathtt{splitplane} \oplus\otimes\ (\text{reax}_{p+2}\,\mathtt{A})\,(\text{reax}_{p+2}\,\mathtt{B})) \dashv_{p+2} \\ & \qquad (\mathtt{splitplane} \oplus\otimes \mathtt{C\,D}),\end{aligned}$$
$$\mathtt{splitplane} \oplus\otimes \left((\text{reax}_3)^{p-1} \circ (\text{reax})\,\mathtt{F}\right) ((\text{reax}_2)^p\,\mathtt{G}) = \circledcirc \oplus\otimes \mathtt{F\,G}.$$

The program uses the highly parallel abstractors cover and untile. The subprogram `multcol` multiplies the left array by a "column" of the right array, which may actually be a block array. The subprogram `multrow` multiplies a "row" (which may be a block) of the left array by a column of the right array. The subprogram `splitplane` splits the blocks up into hyperplanes which contain the axes that are used in an inner product. For 2-arrays, `splitplane` merely computes an inner product.

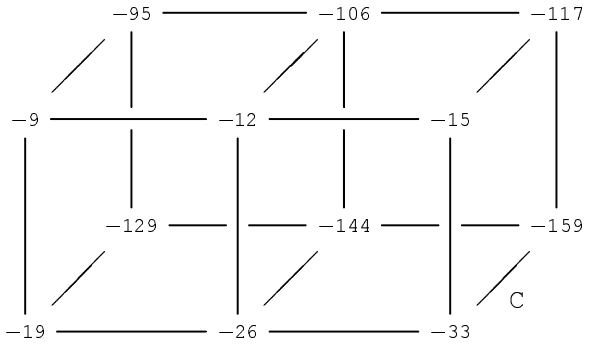
EXAMPLE 5.16 Consider two 3-arrays $\mathtt{A} = [\mathtt{a} : 2,2,2\langle_3 1,2,3,4,5,6,7,8\rangle]$ and $\mathtt{B} = [\mathtt{b} : 2,3,2\langle_3 -1,-2,-3,-4,-5,-6,-7,-8,-9,-10,-11,-12\rangle]$ depicted below.



The horizontal and vertical arrows are suggestive of the rows and columns, respectively, for usual matrix multiplication. The array product $\divideontimes + \times\,\mathtt{A\,B}$ is a 3-array $\mathtt{C} = [\mathtt{c} : 2,3,2\langle_3 -9,-12,-15,-19,-26,-33,-95,-106,-117,-129,-144,-159\rangle]$ obtained by multiplying matrices taken from corresponding hyperplanes

$$\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \begin{bmatrix} -7 & -8 & -9 \\ -10 & -11 & -12 \end{bmatrix} \qquad \begin{bmatrix} -95 & -106 & -117 \\ -129 & -144 & -159 \end{bmatrix}$$
$$\longmapsto \quad ,$$
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \end{bmatrix} \qquad \begin{bmatrix} -9 & -12 & -15 \\ -19 & -26 & -33 \end{bmatrix}$$

and is sketched next.



Next, consider powers of square arrays. Given the usual matrix multiplication operators and a square matrix A, powers of the array may be computed. For instance,

$$A^2 = \divideontimes + \times A A.$$

Recall

$$A^k = \prod_{b_j=1} A^{2^j}$$

where k has the base 2 representation $b_n \cdots b_0$. Any power of an array may be computed by computing all of the factors A^{2^j} and then multiplying them together. Each factor A^{2^j} may be computed via **repeated squaring**

$$A^{2^j} = \left(\left((A)^2 \right)^2 \cdots \right)^2.$$

The following program computes powers using repeated squaring.

Program power;

Let $n \in \mathbb{N}$.

power $2n \oplus\otimes A =$ power $n \oplus\otimes (\divideontimes \oplus\otimes A A)$,
power $1 \oplus\otimes A = A$.

5.7 SUMMARY

Programming techniques, such as the *collapse rule,* which is a standard programming rule to collapse an array, were explained. Array forms were used to guarantee

that all arrays are programmed. Many common operations were programmed, including zip, distl, distr, split, border, unborder, ravel, reshape, reverse, transpose and higher-order operations.

Programming with arrays is possible without handling indices excessively. Using primitive operations, it is easy to write and read programs compared to general programs. The theory is relevant since properties of operations can be studied in the formalism.

Operations may be defined more generally than is usually possible using a general model. For instance, array product was defined for any suitable operators, not only usual matrix multiplication. Software may be developed by building on this foundation, employing the primitives and programming templates from the theory. Future work should focus on optimization techniques.

CHAPTER 6

NEAR HOMOMORPHISMS

The level of abstraction seems to be at the right level. Higher-order functions and homomorphisms have been furnished for generic programming. Cole (1993) showed how it is possible, with the aid of a little extra "baggage," to turn problems that are not genuine homomorphisms into near homomorphisms. The attraction of the approach is twofold. First, the approach offers an intuitive way to partition problems in a way that often leads to a solution. Second, a solution if it can be found is almost always implicitly parallel and efficient. Case studies show that programs can be written at a low-level using fine-grain parallelism.

6.1 MAP AND REDUCE

Recall, list homomorphisms are compositions of a map and a reduction and every such composition is a homomorphism. This means that list homomorphisms can be partitioned into two processes. The intuition is to try to squeeze as much of the solution into the map and the rest into the reduction. If mostly useful work is done in each phase, then the algorithm is likely to be efficient. It is well-known that homomorphisms provide templates for generic programming. Such models are useful for new computing technologies (Trichina and Eriksson-Bique, 1998).

Unfortunately, not all processing problems can be modelled by homomorphisms. The diversity of parallel algorithms attests to this fact. Surprisingly perhaps, it seems not even homomorphisms themselves can always be expressed in a practical way as a composition of a map and a reduction, although proving this fact is tantamount to proving that no such solution exists (perhaps it just has not yet been found). The next example illustrates that it is sometimes difficult to write an efficient program that computes a homomorphism as a composition of a map and a reduction.

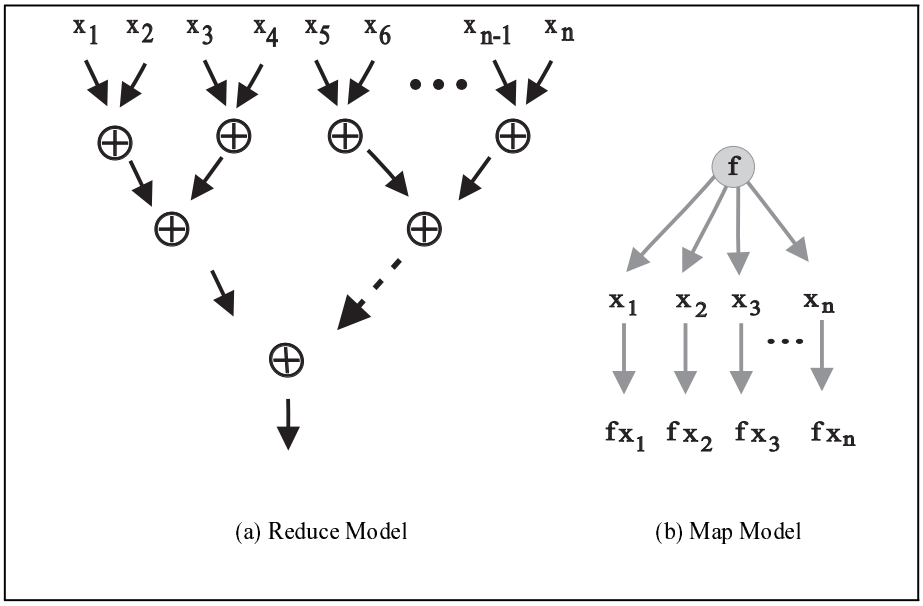


Figure 6.1: (a) Reduce and (b) map models.

EXAMPLE 6.1 The function `rotate` pushes the head of a list to its end:

$$\texttt{rotate}[\texttt{a}:\texttt{n}] = [\texttt{a} \circ \phi : \texttt{n}]$$

where

$$\phi(\texttt{k}) = (\texttt{k} \bmod \texttt{n}) + 1.$$

This function is a homomorphism. Define the operator $\oplus$ so that it pulls the last mem-

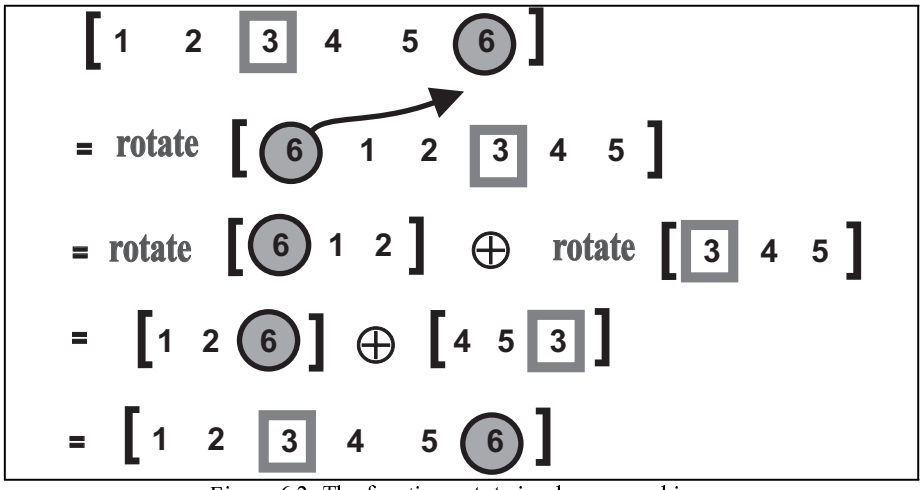


Figure 6.2: The function **rotate** is a homomorphism.

ber of the right list to the front of that list, and then pushes the last member of the left list to the very end of the concatenation of the lists. In addition, define $\texttt{f x} = (\cdot)(\texttt{x})$. Then

$$\texttt{rotate} = /\oplus \circ *\texttt{f}.$$

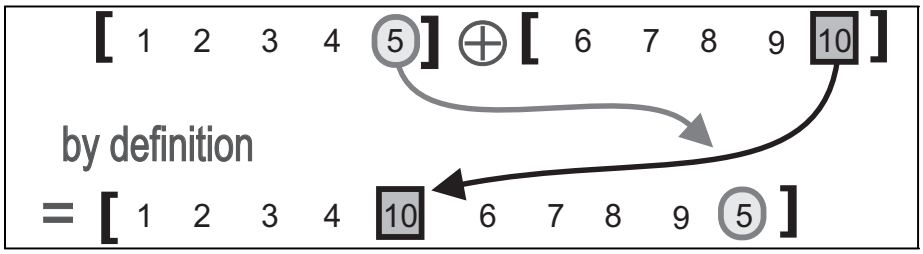


Figure 6.3: Definition of binary operator $\oplus$ for rotate.

Unfortunately, a single application of $\oplus$ is computationally more expensive than the operation `rotate` itself.

Hence near homomorphisms are not always useful. Notwithstanding this fact, many problems admit a satisfactory solution in the homomorphic style. Any parts of the solution which are not homomorphisms, i.e., the baggage, do not contribute significantly to the expected cost of the algorithm using a sufficient degree of parallelism. In the sequel, programs are developed for a few classical problems to show that it is easy and natural to derive nearly homomorphic solutions.

6.2 PARENTHESES MATCHING

Does a given string contain only matching parentheses? This question is solved via a program developed next (Trichina and Eriksson-Bique, 1999). Let $\mathbb{L}$ denote any alphabet with $) \in \mathbb{L}$ and $(\in \mathbb{L}$. For convenience define for $\alpha \in \mathbb{L}$,

$$\rho(\alpha) = \begin{cases} 1 & \text{if } \alpha = (\\ -1 & \text{if } \alpha =) \\ 0 & \text{otherwise} \end{cases} .$$

Strings are essentially lists without any punctuation, i.e., a string is what is left after removal of the brackets and the commas from a list. A string may be identified with a 1-array. The following characterization of strings is useful.

THEOREM 6.1 (MATCHING PARENTHESIS CHARACTERIZATION)

A finite string $[\alpha : \mathtt{n}] \in \mathbb{A}_{\mathbb{L}}$ contains only matching parentheses if and only if the following conditions are satisfied:

$$(/+) \circ (*\rho)\,[\alpha : \mathtt{k}] \geq 0 \qquad \text{for all } \mathtt{k} \in \iota\, \mathtt{n}, \tag{6.1}$$

$$(/+) \circ (*\rho)\,[\alpha : \mathtt{n}] = 0. \tag{6.2}$$

PROOF The theorem holds trivially if $n = 1$. Assume the theorem holds whenever $n \leq q$ for some $q \geq 1$. Let $n = q+1$ and $[\alpha : n] \in \mathbb{A}_{\mathbb{L}}$ be arbitrary. If there are no parentheses in the given string, then every partial sum is zero. If there is exactly one parenthesis, then the total sum is not zero. Suppose that there are at least two parentheses. Let k and j, respectively, be the smallest and largest integers in ιn such that α_k and α_j are parentheses. If α_k is a right parentheses then

$$(/+) \circ (*\rho)\,[\alpha : k] = \rho\alpha_k = -1$$

and the given string is unbalanced. Assume $\alpha_k = ($. If α_j is a left parenthesis then the given string is unbalanced. The conditions of the theorem cannot be satisfied in this case. Suppose by way of contradiction (bwoc) that the conditions of the theorem are satisfied. Since $(/+) \circ (*\rho)$ is a homomorphism,

$$\begin{aligned} 0 &= (/+) \circ (*\rho)\,[\alpha : n] \\ &= (/+) \circ (*\rho)\,[\alpha : j] + (/+) \circ (*\rho)\,[\alpha \circ (+j) : n - j] \\ &= (/+) \circ (*\rho)\,[\alpha : j] \end{aligned}$$

using the fact that $\rho \circ \alpha \circ (p) = 0$ for $p > n - j$ by the choice of j as the largest index for which α_j is a parenthesis. Furthermore

$$\begin{aligned} 0 &= (/+) \circ (*\rho)\,[\alpha : j] \\ &= (/+) \circ (*\rho)\,[\alpha : j-1] + \rho \circ \alpha_j \\ &= (/+) \circ (*\rho)\,[\alpha : j-1] + 1. \end{aligned}$$

But then

$$(/+) \circ (*\rho)\,[\alpha : j-1] = -1,$$

a contradiction. Hence, the conditions of the theorem cannot be satisfied. Now assume $\alpha_j =)$. Suppose the string $[\alpha : n]$ is balanced. Upon removing the leftmost parenthesis which is $\alpha_k = (\,$ and the rightmost parenthesis which is $\alpha_j =)$, the remaining string is balanced and by the inductive hypothesis the conditions of the theorem are met. It follows that

$$\begin{aligned} (/+) \circ (*\rho)\,[\alpha : p] &= 0 \qquad \text{for } p < k, \\ (/+) \circ (*\rho)\,[\alpha : k] &= 1, \\ (/+) \circ (*\rho)\,[\alpha : p] &\geq 1 \qquad \text{for } k \leq p < j, \\ (/+) \circ (*\rho)\,[\alpha : p] &= 0 \qquad \text{for } p \geq j. \end{aligned}$$

Thus, the conditions of the theorem hold for $[\alpha : n]$ in this case. Lastly, assume the string $[\alpha : n]$ is unbalanced. Removing the leftmost parenthesis which is $\alpha_k = (\,$ and the rightmost parenthesis which is $\alpha_j =)$, the remaining string must be unbalanced; whence, by the inductive hypothesis, the conditions of the theorem fail to hold for this

string. It follows that the conditions of the theorem cannot hold for $[\alpha : n]$. ■

The proof is included mainly to show that the formalism is relevant and adequate for theoretical analysis of problems in addition to programming them. If the string $[\alpha : n]$ contains only matching parentheses, it is easy to see that the stated conditions are satisfied. In view of the theorem, it might appear that any algorithm would have to be sequential as every partial sum must be nonnegative. A straightforward solution which directly follows from the theorem is nothing more than a directed reduce operation, which is suitable for a sequential computer. Define the operator $\otimes$ via

$$e \otimes a = \begin{cases} -1 & \text{if } e < 0 \\ e + \rho a & \text{otherwise} \end{cases}.$$

The sequential program is

$$(= 0) \circ (\rightarrowtail \otimes 0).$$

As soon as a negative result is encountered, it is unnecessary to look at additional characters. Hence, it would be beneficial to carry out some method of lazy evaluation. However, the information should be available to all parallel processors, which depends upon the implementation, e.g., the read/write memory model.

To obtain a homomorphic solution, it should be possible to split the input arbitrarily into two blocks and process each block independently. How is it possible to check that every partial sum is nonnegative using only the results from the blocks? Upon reflection, it is enough to keep track of the minimum and the total sum from each block.

LEMMA 6.2 Equation (6.1) fails to hold if and only if

$$\min_{k<n} \{(/+) \circ (*\rho)[\alpha : k]\} < 0. \tag{6.3}$$

PROOF Let k be the smallest integer with $0 \le k < n$ such that $(/+) \circ (*\rho)[\alpha : k] < 0$. If no such integer k exists, then

$$(/+) \circ (*\rho)[\alpha : k] \ge 0 \quad \text{for all } k \in \iota n.$$

Otherwise if there exists such an integer k then

$$\min_{i<n} \{(/+) \circ (*\rho)[\alpha : i]\} \le (/+) \circ (*\rho)[\alpha : k] < 0. \blacksquare$$

The baggage which is needed is a pair (t, m) that stores the result for every block as illustrated in Figure 6.4(b). Letting $[\alpha : k]$ represent a block of characters, the total

sum is

$$t = (/+) \circ (*\rho)\,[\alpha : k] = \rho(\alpha_1) + \rho(\alpha_2) + \cdots + \rho(\alpha_k).$$

The minimum partial sum is the number

$$m = \min_{q<k} \{(/+) \circ (*\rho)\,[\alpha : q]\}.$$

DEFINITION 6.1 Let the pairs (a,b) and (c,d) denote the results after applying the parentheses matching algorithm independently to two adjacent blocks as depicted in Figure 6.4(b). To combine these partial results, define the reduce operator $\oplus$ via

$$(a,b) \oplus (c,d) = (a+c,\ b \downarrow (a+d)), \tag{6.4}$$

where $\downarrow$ denotes the min operator in infix form. Take as the identity element $id_{\oplus} = (0,\infty)$. Let the function f for the map in the near homomorphic solution be given by

$$f(\alpha) = (\rho\alpha,\ 0). \tag{6.5}$$

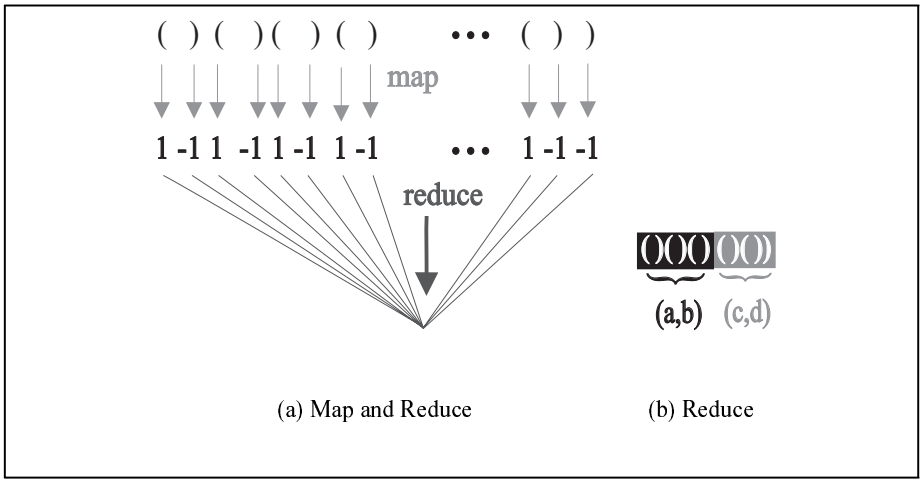


Figure 6.4: (a) The matching parentheses program applies $*f$ producing the baggage and then applies $/\oplus$. (b) The partial results from adjacent blocks are combined thusly: $(a,b) \oplus (c,d)$.

Program MatchingParentheses;

Let the map function f and reduce operator $\oplus$ be as in equations (6.5) and (6.4), respectively.

$$\texttt{MatchingParentheses} = \odot \circ (/\oplus) \circ (*f),$$
$$\odot(a,b) = (a = 0) \wedge (b \geq 0).$$

EXAMPLE 6.2 Calculate $\texttt{MatchingParentheses}\,\overline{(,),(,),(,),)}$:

$$
\begin{aligned}
&\odot\circ(/\oplus)\circ(*\mathtt{f})\overline{(,),(,),(,),)}\\
&=\odot((1,0)\oplus(-1,0))\oplus((1,0)\oplus(-1,0))\oplus((1,0)\oplus(-1,0))\oplus(-1,0)\\
&=\odot((0,0)\oplus(0,0))\oplus((0,0)\oplus(-1,0))\\
&=\odot((0,0)\oplus(-1,0))\\
&=\odot(-1,0)=0\wedge 1=0,
\end{aligned}
$$

which is the correct result as ()()()) is not balanced.

The operator $\oplus$ is well-defined since $+$ and $\downarrow$ are well-defined. In order to apply reduce, it is required to verify that $\oplus$ is associative. Let $\mathtt{a}=(\mathtt{a}_1,\mathtt{a}_2)$, $\mathtt{b}=(\mathtt{b}_1,\mathtt{b}_2)$, and $\mathtt{c}=(\mathtt{c}_1,\mathtt{c}_2)$. It is necessary to verify that

$$(\mathtt{a}\oplus\mathtt{b})\oplus\mathtt{c}=\mathtt{a}\oplus(\mathtt{b}\oplus\mathtt{c}).$$

Since $+$ is an associative operator, it is enough to show that

$$(\mathtt{a}_2\downarrow(\mathtt{a}_1+\mathtt{b}_2))\downarrow((\mathtt{a}_1+\mathtt{b}_1)+\mathtt{c}_2)=\mathtt{a}_2\downarrow(\mathtt{a}_1+(\mathtt{b}_2\downarrow(\mathtt{b}_1+\mathtt{c}_2))).$$

It follows that this equality holds since $\downarrow$ is an associative operator and

$$\mathtt{a}_1+(\mathtt{b}_2\downarrow(\mathtt{b}_1+\mathtt{c}_2))=(\mathtt{a}_1+\mathtt{b}_2)\downarrow(\mathtt{a}_1+(\mathtt{b}_1+\mathtt{c}_2)).$$

After gaining some experience, it is often fairly easy to define operations so that they are associative and to recognize operators that are nonassociative. It may be possible to find an example that confirms an operator is nonassociative. Nevertheless, checking associativity is not always trivial. All cases must be considered, especially when the solution is nearly correct.

THEOREM 6.3 The program `MatchingParentheses` solves the matching parentheses problem.

PROOF In view of Theorem 6.1 and Lemma 6.2, it suffices to show that for any string $[\alpha:\mathtt{n}]\in\mathbb{A}_{\mathbb{L}}$,

$$(/\oplus)\circ(*\mathtt{f})\,[\alpha:\mathtt{n}]=(\mathtt{t},\mathtt{m}) \tag{6.6}$$

where

$$\mathtt{t}=(/+)\circ(*\rho)\,[\alpha:\mathtt{n}]$$

and

$$\mathtt{m} = \min_{\mathtt{k}<\mathtt{n}} \{(/+) \circ (*\rho)\,[\alpha : \mathtt{k}]\}.$$

Proceed by induction on the number $\mathtt{n} \geq 1$. If $\mathtt{n} = 1$ then

$$/\oplus \circ (*\mathtt{f})\,(\cdot)\,(\alpha_1) = /\oplus (\rho\alpha_1, 0) = (\rho\alpha_1, 0) = (\mathtt{t}, \mathtt{m}).$$

Suppose equation (6.6) holds whenever $\mathtt{n} \leq \mathtt{q}$ for some $\mathtt{q} \geq 1$. Put $\mathtt{n} = \mathtt{q}$ and let $[\alpha : \mathtt{n}] \in \mathbb{A}_{\mathbb{L}}$ be arbitrary. Choose $\mathtt{k} \in \mathtt{i}\ \mathtt{n}$. Since $(/\oplus) \circ (*\mathtt{f})$ is a homomorphism,

(6.7) $$(/\oplus) \circ (*\mathtt{f})\,[\alpha : \mathtt{n}] = (/\oplus) \circ (*\mathtt{f})\,[\alpha : \mathtt{k}] \oplus (/\oplus) \circ (*\mathtt{f})\,[\alpha \circ (+\mathtt{k}) : \mathtt{n} - \mathtt{k}].$$

By the inductive hypothesis,

$$(/\oplus) \circ (*\mathtt{f})\,[\alpha : \mathtt{k}] = \left((/+) \circ (*\rho)\,[\alpha : \mathtt{k}]\,,\ \min_{\mathtt{i}<\mathtt{k}} \{(/+) \circ (*\rho)\,[\alpha : \mathtt{i}]\}\right),$$

and

$$(/\oplus) \circ (*\mathtt{f})\,[\alpha \circ (+\mathtt{k}) : \mathtt{n} - \mathtt{k}]$$
$$= \left((/+) \circ *\rho\,[\alpha \circ (+\mathtt{k}) : \mathtt{n} - \mathtt{k}]\,,\ \min_{\mathtt{i}<\mathtt{n}-\mathtt{k}} \{(/+) \circ *\rho\,[\alpha \circ (+\mathtt{k}) : \mathtt{i}]\}\right).$$

Perform the operation $\oplus$ in equation (6.7) after substituting the results from the previous two equations to obtain

$$(/\oplus) \circ (*\mathtt{f})\,[\alpha : \mathtt{n}] = (\mathtt{t}, \mathtt{m}).$$

Note that since $(/+) \circ (*\rho)$ is a homomorphism,

$$\mathtt{t} = (/+) \circ (*\rho)\,[\alpha : \mathtt{k}] + (/+) \circ (*\rho)\,[\alpha \circ (+\mathtt{k}) : \mathtt{n} - \mathtt{k}]$$
$$= (/+) \circ *\rho\,[\alpha : \mathtt{n}].$$

In addition, use the fact

$$\mathtt{m} = \min_{\mathtt{i}<\mathtt{k}} \{/+ \circ *\rho\,[\alpha : \mathtt{i}]\} \downarrow$$
$$\left(/+ \circ *\rho\,[\alpha : \mathtt{k}] + \min_{\mathtt{i}<\mathtt{n}-\mathtt{k}} \{/+ \circ *\rho\,[\alpha \circ (+\mathtt{k}) : \mathtt{n} - \mathtt{k}]\}\right)$$

$$= \min_{\mathtt{i}<\mathtt{n}} \{/+ \circ *\rho\,[\alpha : \mathtt{i}]\}. \blacksquare$$

Consider the number of operations for a string with $\mathtt{n}$ characters. First, consider only the task requirements. Checking all $\mathtt{n} - 1$ partial sums and the final sum, yields

at least n comparisons. To compute the sums requires at least $n-1$ additions. Hence, it is necessary to perform approximately $2n$ operations. Second, analyze the program. Applying map $*f$ involves n operations. Doing the reduction requires about $3n$ additional operations, since each application of $\oplus$ requires three operations (two additions and one minimum). Thus, the program requires approximately $4n$ operations. So the program requires $O(n)$ operations which is the same order of magnitude as the required number of operations.

Consider the time complexity. It depends only on map and reduce since the other operations can be done in constant time. Suppose that an initial string is distributed among n processing elements. Then the map can be done in $O(1)$ time steps. The reduction can be carried via a tree-like structure, yielding in total $O(\log n)$ steps. The solution is as efficient as expected.

6.3 PATTERN MATCHING

Consider the following query: does a given text contain a given string? Such a query is common when searching for documents which contain a key word or pattern. A brute-force approach is taken. Other popular algorithms such as Boyer-Moore, Karp-Robin, and Knuth-Morris-Pratt, are not considered.

Before proceeding with the development of a program, a couple operations are needed. For this application, a more restricted version of the concatenation operator $+\!\!+$ is appropriate. Presumably, if the lengths of all strings are known in advance then this information can be utilized. Although the length of the text may not be known, the length of each block is known since the text is partitioned into equal sized blocks.

DEFINITION 6.2 Let

$$x +\!\!+^0 y = \square$$

and for $k \in \mathbb{N}$,

$$\begin{aligned}
\square +\!\!+^k \square &= \square, \\
\square +\!\!+^k ((\cdot)(a) +\!\!+ y) &= (\cdot)(a) +\!\!+ \left(\square +\!\!+^{k-1} y\right), \\
((\cdot)(a) +\!\!+ x) +\!\!+^k y &= (\cdot)(a) +\!\!+ \left(x +\!\!+^{k-1} y\right).
\end{aligned}$$

The operator $+\!\!+^k$ works like $+\!\!+$ except that the former drops all elements, if any, after the kth one. Interestingly, `initial` is now a homomorphism:

$$(\texttt{initial}\ k)(x +\!\!+ y) = (\texttt{initial}\ k)(x) +\!\!+^k (\texttt{initial}\ k)(y).$$

Another useful operation produces “windows” that overlap in a regular manner.

DEFINITION 6.3 Let $k \geq 0$. The operator $\ominus^k$ is given by

$$x \ominus^k ((\cdot)(a) +\!\!+ ys) = \left(*\left(+\!\!+^k \circ \leftarrow\!\!\rho\, a\right)(x)\right) +\!\!+ ((\cdot)(a) +\!\!+ (ys)).$$

Put $f(x) = (\cdot)(\cdot)(x)$. The operation $\mathtt{window}\, k$ is the homomorphism

$$\mathtt{window}\, k = \left(/\ominus^k\right) \circ (*f).$$

The resulting objects produced by its application are called **windows**.

EXAMPLE 6.3 Compute

$$\begin{aligned}
&(\mathtt{window}\, 3)\, \overline{a_1, a_2, a_3, a_4, a_5} \\
&\quad = \overline{\overline{a_1}} \ominus^3 \overline{\overline{a_2}} \ominus^3 \overline{\overline{a_3}} \ominus^3 \overline{\overline{a_4}} \ominus^3 \overline{\overline{a_5}} \\
&\quad = \overline{\overline{a_1, a_2}, \overline{a_2}} \ominus^3 \overline{\overline{a_3, a_4}, \overline{a_4}} \ominus^3 \overline{\overline{a_5}} \\
&\quad = \overline{\overline{a_1, a_2, a_3}, \overline{a_2, a_3, a_4}, \overline{a_3, a_4}, \overline{a_4}} \ominus^3 \overline{\overline{a_5}} \\
&\quad = \overline{\overline{a_1, a_2, a_3}, \overline{a_2, a_3, a_4}, \overline{a_3, a_4, a_5}, \overline{a_4, a_5}, \overline{a_5}}.
\end{aligned}$$

Consider a program to compare two strings. Informally, two strings match if one string is the initial part of another. The strings a and b match if there exists a string c (possibly $\square$) such that either

$$a = b +\!\!+ c \qquad \text{or} \qquad b = a +\!\!+ c.$$

The primitive matching operation is defined next.

Program match;

Let $\wedge$ and $==$ denote the usual Boolean AND operation and equality test, respectively.

$$\mathtt{match} = (/\wedge) \circ \underset{\longleftrightarrow}{==}.$$

The program match is a near homomorphism. Rewriting by Lemma 4.58,

$$\mathtt{match} = (/\wedge) \circ (*==) \circ \underset{\longleftrightarrow}{\mathtt{id}}.$$

Instead of searching from the beginning to the end of the given text, split the text into approximately equal blocks and perform a brute-force search concurrently on each block. The only potential problem is that the given pattern may overlap two adjacent blocks. After checking each block, it is necessary to employ an additional reduction to determine if there is a match between two adjacent blocks.

The results from a block will be stored in a triple (α, β, δ). The string α is the leftmost window in the block. This string must be kept in case there is a partial match

from the adjacent block on the left side. The string β is the matching string if there is one. The Boolean value δ equals 1 if and only if there is a matching string.

Denote the given pattern by $\mathtt{P}$. The strategy is to split the given text into large blocks with $\mathtt{k}$ characters each and then form windows of size

$$\mathtt{WS} = \text{length } \mathtt{P}$$

in each such block. Assume $\mathtt{k} >> \mathtt{WS}$. In the first round, try to match each window with the given pattern. In the second round, search for a match between blocks. Put

$$\mathtt{PM_P} = (\mathtt{id}_{\mathbb{L}^*}, \mathtt{id}_{\mathbb{L}^*}, \mathtt{match} \circ \mathtt{P}), \tag{6.8}$$

where $\mathbb{L}$ denotes any alphabet, and

$$(\mathtt{f}, \mathtt{g}, \mathtt{h})(\mathtt{x}) = (\mathtt{f}(\mathtt{x}), \mathtt{g}(\mathtt{x}), \mathtt{h}(\mathtt{x}))$$

is the usual construction (Backus, 1978). Applying $\mathtt{PM_P}$ to a window α yields a triple (α, α, δ) where δ is the Boolean value resulting from comparing the strings α and $\mathtt{P}$. Note $\mathtt{WS}$ and $\mathtt{PM_P}$ are part of the pre-processing phase.

DEFINITION 6.4 The operator $\oplus$ is given by

$$(\alpha, \beta, \delta) \oplus (\mathtt{a}, \mathtt{b}, \mathtt{d}) = \begin{cases} (\alpha, \mathtt{b}, \mathtt{d}) & \text{if } \delta = 0 \\ (\alpha, \beta, \delta) & \text{if } \delta = 1 \end{cases}, \tag{6.9}$$

and is used to combine results within a block.

If a matching string from any block has the same length as the pattern, then the pattern occurs within the text. Otherwise, it will be necessary to check if the pattern overlaps two adjacent blocks. For this purpose, the operator $\odot_{\mathtt{P}}$ is used.

DEFINITION 6.5 The operator $\odot_{\mathtt{P}}$ is given by

$$(\alpha, \beta, \delta) \odot_{\mathtt{P}} (\mathtt{a}, \mathtt{b}, \mathtt{d}) \tag{6.10}$$

$$= \begin{cases} (\alpha, \mathtt{b}, \mathtt{d}) & \text{if } \delta = 0 \\ (\alpha, \mathtt{P}, 1) & \text{if } (\delta = 1 \wedge \text{length}\,\beta = \mathtt{WS}) \vee (\mathtt{d} = 1 \wedge \text{length}\,\mathtt{b} = \mathtt{WS}) \\ \left(\alpha, \widehat{\mathtt{b}}, \widehat{\mathtt{d}}\right) & \text{otherwise} \end{cases},$$

where

$$\left(\widehat{\mathtt{b}}, \widehat{\mathtt{d}}\right) = \begin{cases} (\mathtt{P}, 1) & \text{if } \widetilde{\mathtt{d}} = 1 \\ (\mathtt{b}, \mathtt{d}) & \text{otherwise} \end{cases},$$

and $\widetilde{\mathtt{d}}$ is the Boolean result of searching for the pattern between the adjacent blocks via

$$\left(\widetilde{\alpha}, \widetilde{\mathtt{b}}, \widetilde{\mathtt{d}}\right) = (/\oplus) \circ (*\mathtt{PM}_{\mathtt{P}}) \circ (\mathtt{initial}\ \mathtt{length}\beta) \circ (\mathtt{window}\,\mathtt{WS})(\beta +\!\!+ (\mathtt{init}\,\mathtt{a})).$$

In the preceding definition, the program used to compute $\widetilde{\mathtt{d}}$ uses mainly subprograms of the following pattern matching program.

Program MatchPattern;

The operations $\mathtt{PM}_{\mathtt{P}}$, $\oplus$, and $\odot_{\mathtt{P}}$ are defined in equations (6.8), (6.9) and (6.10), respectively.

```
MatchPattern
        = last ∘ (/⊙P) ∘ * ((/⊕) ∘ (*PMP) ∘ (window WS)) (dice k),
last (e, f, g) = g.
```

EXAMPLE 6.4 Does the text acbacbacbacabacbac contain the pattern cab? Setting $\mathtt{k} = 6$, compute

$$(\mathtt{dice}\ 6)\,\overline{\mathtt{a,c,b,a,c,b,a,c,b,a,c,a,b,a,c,b,a,c}} = \overline{\overline{\mathtt{a,c,b,a,c,b}}, \overline{\mathtt{a,c,b,a,c,b}}, \overline{\mathtt{a,c,b,a,c,a}}, \overline{\mathtt{b,a,c,b,a,c}}}.$$

As the pattern has three characters, obtain (using map distributivity)

$$*(\mathtt{window}\ 3)\,\overline{\overline{\mathtt{a,c,b,a,c,b}}, \overline{\mathtt{a,c,b,a,c,b}}, \overline{\mathtt{a,c,b,a,c,a}}, \overline{\mathtt{b,a,c,b,a,c}}} = \overline{\mathtt{W}_1, \mathtt{W}_2, \mathtt{W}_3},$$

where

$$\mathtt{W}_1 = \overline{\overline{\mathtt{a,c,b}}, \overline{\mathtt{c,b,a}}, \overline{\mathtt{b,a,c}}, \overline{\mathtt{a,c,b}}, \overline{\mathtt{c,b}}, \overline{\mathtt{b}}},$$
$$\mathtt{W}_2 = \overline{\overline{\mathtt{a,c,b}}, \overline{\mathtt{c,b,a}}, \overline{\mathtt{b,a,c}}, \overline{\mathtt{a,c,a}}, \overline{\mathtt{c,a}}, \overline{\mathtt{a}}},$$
$$\mathtt{W}_3 = \overline{\overline{\mathtt{b,a,c}}, \overline{\mathtt{a,c,b}}, \overline{\mathtt{c,b,a}}, \overline{\mathtt{b,a,c}}, \overline{\mathtt{a,c}}, \overline{\mathtt{c}}}.$$

Next calculate

$$*((/\oplus) \circ (*\mathtt{PM}_{\mathtt{cab}}))\,\overline{\mathtt{W}_1, \mathtt{W}_2, \mathtt{W}_3} = \overline{(\overline{\mathtt{a,c,b}}, \overline{\mathtt{b}}, 0), (\overline{\mathtt{a,c,b}}, \overline{\mathtt{c,a}}, 1), (\overline{\mathtt{b,a,c}}, \overline{\mathtt{c}}, 1)}.$$

Then

$$(/\odot_{\mathtt{cab}})\,\overline{(\overline{\mathtt{a,c,b}}, \overline{\mathtt{b}}, 0), (\overline{\mathtt{a,c,b}}, \overline{\mathtt{c,a}}, 1), (\overline{\mathtt{b,a,c}}, \overline{\mathtt{c}}, 1)} = (\overline{\mathtt{a,c,b}}, \overline{\mathtt{c,a,b}}, 1),$$

and lastly

$$\mathtt{last}([\mathtt{a,c,b}], [\mathtt{c,a,b}], 1) = 1,$$

which is the correct result as cab does appear in the given text.

The operator $\oplus$ is well-defined (since $\delta \in \{0,1\}$ always) and associative. After reducing any block via $/\oplus$, only the leftmost window that matches the given pattern will be preserved. Consequently, a matching string, if any, must have the longest length in the block since windows only decrease in size.

The operator $\odot_P$ is well-defined as $(\alpha,\beta,\delta)\odot_P(a,b,d)=(\alpha,b,d)$ unless a complete matching string exists and in that case $(\alpha,\beta,\delta)\odot_P(a,b,d)=(\alpha,P,1)$. It follows that $\odot_P$ is associative. Note that it is necessary only to find a single occurrence of the given pattern.

In the worst case scenario, it will be necessary to compare the pattern P against every possible window, involving at most $O(nm)$ comparisons where n denotes the length of the text, which is also the number of windows, and $m = \text{length}\,P$ is the maximum number of comparisons for each window. Assuming that splitting the original text, forming windows and applying maps requires only constant time, the time complexity will depend mainly on the reductions, which involve only a portion of the text.

6.4 DIGIT SERIAL COMPUTATIONS

Consider programming usual addition, multiplication and convolution of strings of digits (Eriksson-Bique, 1998). Fix $b \in \mathbb{N}-\{1\}$ with $b \geq 2$. Consider base b computations and let $\mathbb{L}=\{0,1,2,\ldots,b-1\}$. As usual, any arbitrary finite sequence of digits $a_n a_{n-1}\cdots a_0 \in \mathbb{L}*$ represents the number $\sum_{i=0}^{n} a_i b^i$. This representation is unique since $\sum_{i=0}^{n} a_i b^i = \sum_{i=0}^{n} \hat{a}_i b^i$ implies $\sum_{i=1}^{n}(a_i-\hat{a}_i)b^i = \hat{a}_0 - a_0$ must be divisible by b forcing $\hat{a}_0 = a_0$, and dividing through by b and repeating the argument, forces $\hat{a}_i = a_i$ for all i. Adding two digits always yields a string with at most two digits and any carry is always equal to 1 since whenever $\alpha,\beta \in \mathbb{L}$,

$$\alpha+\beta \leq (b-1)+(b-1) = 2b-2 < 2b-2+1 = 1b^1+(b-1)b^0,$$

which means

$$\alpha+\beta = \sum_{i=0}^{1} a_i b^i \tag{6.11}$$

for some unique $a_0, a_1 \in \mathbb{L}$ with $a_1 \leq 1$ whenever $\alpha,\beta \in \mathbb{L}$.

DEFINITION 6.6 In equation (6.11), call the coefficient a_0 the "digit part" and the coefficient a_1 the "carry part." Using the same equation, define the "carry add"

operator $+_c$ via

$$\alpha +_c \beta = a_1 \in \{0,1\}$$

and define the "digit add" operator $+_d$ via

$$\alpha +_d \beta = a_0.$$

By uniqueness of the coefficients in equation (6.11), these operations are well-defined.

6.4.1 USUAL ADDITION

The problem is to add two long strings in $\mathbb{L}*$. First analyze the problem and second develop a homomorphic solution. Let $a_m a_{m-1} \cdots a_0,\ \hat{a}_n \hat{a}_{n-1} \cdots \hat{a}_0 \in \mathbb{L}*$. Without loss of generality, assume $m \le n$ (otherwise relabel using commutativity of addition). Set $a_i = 0$ for all $m < i \le n$. Then

$$\sum_{i=0}^{n} a_i b^i + \sum_{i=0}^{n} \hat{a}_i b^i = \sum_{i=0}^{n} (a_i + \hat{a}_i) b^i. \tag{6.12}$$

It is necessary to investigate how the carries propagate from right to left, i.e., from the ith term (with coefficient $a_i + \hat{a}_i$) to the $(i+1)$ st term in the right hand side of equation (6.12).

First, the number of digits in the result is at most $n+2$ as

$$\sum_{i=0}^{n} a_i b^i + \sum_{i=0}^{n} \hat{a}_i b^i < b^{n+1} + b^{n+1} = 2b^{n+1} \le bb^{n+1} = b^{n+2} \tag{6.13}$$

implying that

$$\sum_{i=0}^{n} a_i b^i + \sum_{i=0}^{n} \hat{a}_i b^i = \sum_{i=0}^{n+1} \tilde{a}_i b^i$$

for some $\tilde{a}_i$'s in $\mathbb{L}$. The task is to compute these $\tilde{a}_i$'s.

Second, the maximum carry is at most 1, as initially ($i = 0$) the carry is zero and for $i \ge 0$

$$\mathrm{carry}_i + a_i + \hat{a}_i \le 1 + (b-1) + (b-1) = 1b + (b-1),$$

which shows that the carry for the $(i+1)$ st term is at most 1. In view of equation (6.12) and equation (6.13) and uniqueness of representation, it follows that $\tilde{a}_i = a_i +_d \hat{a}_i +_d \mathrm{carry}_i$ where $\mathrm{carry}_i \in \{0,1\}$.

Consider a string adder as a homomorphism. Partition the input strings into blocks with corresponding blocks having the same size. After adding all corresponding pairs of blocks, a reduce operation is applied to combine the results.

Two corresponding substrings are added together, doing computations from right to left, as depicted below.

$\longleftarrow$

$\widehat{a}_n$	$\cdots$	$\widehat{a}_{j*+2}$	$\widehat{a}_{j*+1}$	$\widehat{a}_{j*}$	$\cdots$	$\widehat{a}_j$	$\widehat{a}_{j-1}$	$\cdots$	$\widehat{a}_0$
a_n	$\cdots$	a_{j*+2}	a_{j*+1}	a_{j*}	$\cdots$	a_j	a_{j-1}	$\cdots$	a_0
$\overline{a}_n$	$\cdots$	$\overline{a}_{j*+2}$	$\overline{a}_{j*+1}$	$\overline{a}_{j*}$	$\cdots$	$\overline{a}_j$	$\overline{a}_{j-1}$	$\cdots$	$\overline{a}_0$

Assume the strings are partitioned into blocks and each block is added concurrently. In any block

$\widehat{a}_{j*+1}$	$\widehat{a}_{j*}$	$\cdots$	$\widehat{a}_j$
a_{j*+1}	a_{j*}	$\cdots$	a_j
$\overline{a}_{j*+1}$	$\overline{a}_{j*}$	$\cdots$	$\overline{a}_j$

.

if there will be no carry from the right, then all of the computed $\overline{a}_j$'s are correct. What happens if there is a carry of 1 from the right? It must be added to $\overline{a}_j$. If

$$\overline{a}_j +_c 1 = 0,$$

then the remaining $\overline{a}_j$'s are correct; otherwise, add the carry to the next digit and if

$$\overline{a}_{j+1} +_c 1 = 1,$$

it will be necessary to repeat the process. When will the carry stop propagating to the left? Precisely when $\overline{a}_{\widehat{j}} < b-1$ is encountered since this inequality holds if and only if $\overline{a}_{\widehat{j}} + 1 \leq b-1$. Hence, either all the digits in the block are correct or it is only necessary to add 1 to all the digits to the right of and including the rightmost digit less than $b-1$.

Before receiving any carry from the right, the resulting string for each block may be split into two parts: the left part, say l, and the right part, say r. The string l (possibly empty) is guaranteed to be correct and the string r is either correct or it will be necessary to add 1 to all of the digits in this substring. Actually, it is necessary to add one only to the leftmost digit of r and set all remaining digits to zero. A tag, say t, is needed to indicate whether or not the right string r actually contains a digit less than $b-1$, as it is possible r will not stop a carry.

The baggage consists of quadruples of the form (c, l, r, t) where

$$c \# l \# r$$

is the correct result of adding two given substrings. In particular, if there is a carry from adding the given substrings, then $c = \neg$; otherwise, $c = \square$. Moreover, the string l is known to be correct and the right string r will stop any carry whenever l has any digits. All of the digits in string r are correct if there will be no carry from other blocks; otherwise, it will be necessary to modify the digits in r as discussed in the

preceding paragraph. The tag $t \in \{0,1\}$ will be nonzero if and only if the string r has a digit less than $b-1$.

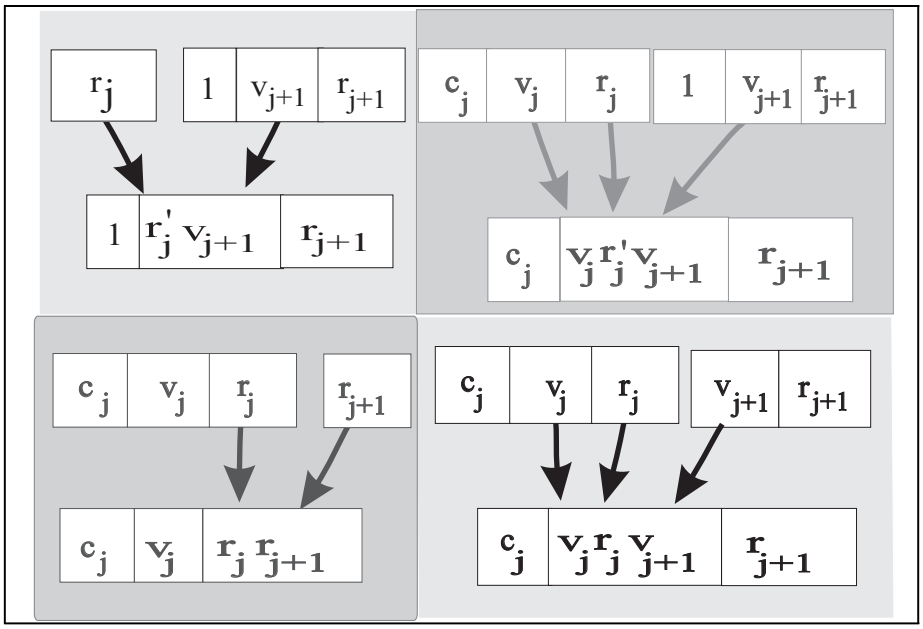


Figure 6.5: Different cases for combining strings added independently.

DEFINITION 6.7 For any two digits α, β let

$$\begin{aligned} d &= \alpha +_d \beta, \\ t &= d < b-1. \end{aligned}$$

Define the function f by

$$f(\alpha,\beta) = \begin{cases} (\neg, \square, \overline{d}, t) & \text{if } \alpha +_c \beta = 1 \\ (\square, \square, \overline{d}, t) & \text{otherwise} \end{cases}. \tag{6.14}$$

DEFINITION 6.8 The binary operator $\oplus$ is given by

$$(c_1, l_1, r_1, t_1) \oplus (c_2, l_2, r_2, t_2)$$

$$= \begin{cases} (c_2, r_1' \# l_2, r_2, t_2) & \text{if } c_2 \neq \square,\ t_1 = 0 \quad (6.15a) \\ (c_1, l_1 \# r_1' \# l_2, r_2, t_2) & \text{if } c_2 \neq \square,\ t_1 = 1 \quad (6.15b) \\ (c_1, l_1, r_1 \# r_2, t_1) & \text{if } c_2 = \square,\ t_2 = 0 \quad (6.15c) \\ (c_1, l_1 \# r_1 \# l_2, r_2, t_2) & \text{if } c_2 = \square,\ t_2 = 1 \quad (6.15d) \end{cases} \tag{6.15}$$

where $\mathtt{r}'$ is defined by

$$\mathtt{r} = (\cdot)(\mathtt{a}) +\!\!+ \mathtt{xs} \quad \Longrightarrow \quad \mathtt{r}' = (\cdot)(\mathtt{a} +_{\mathtt{d}} \mathtt{1}) +\!\!+ (*\mathtt{K}_0)\,\mathtt{xs}.$$

For completeness, the identity element is $\mathtt{id}_{\oplus} = (\Box, \Box, \Box, \mathtt{0})$.

The digit serial adder program is given next.

Program AddStrings;

The map function $\mathtt{f}$ and the binary operator $\oplus$ are defined in equations (6.14) and (6.15), respectively.

$$\begin{aligned}
\mathtt{AddStrings} &= \mathtt{unbag} \circ (/\oplus \circ * \mathtt{f}) \circ \underset{\longleftrightarrow}{\mathtt{id}} \\
&= \mathtt{unbag} \circ /\oplus \circ \underset{\longleftrightarrow}{\mathtt{f}}, \\
\mathtt{unbag}(\mathtt{c}, \mathtt{l}, \mathtt{r}, \mathtt{t}) &= \mathtt{c} +\!\!+ \mathtt{l} +\!\!+ \mathtt{r}.
\end{aligned}$$

If one string is shorter, prefix sufficiently many zeros so that they both have the same length. Although AddStrings is not a pure homomorphism that applies to a single argument, the pre-processing $\underset{\longleftrightarrow}{\mathtt{id}}$ is an implicitly parallel abstractor, and the post-processing operation unbag only concatenates the resulting strings.

THEOREM 6.4 The program AddStrings computes the sum of two strings.

PROOF It turns out that the baggage can only appear in certain forms. In order to show the program is correct, it is necessary to investigate all possible forms. The function $\mathtt{f}$ and the binary operator $\oplus$ are well-defined by inspection. It is impossible to obtain a quadruple of the form

$$(\mathtt{c}, \mathtt{l}, \Box, \mathtt{t})$$

by the construction of $\mathtt{f}$ and $\oplus$. In addition, it is impossible to obtain a quadruple

$$(\mathtt{c}, \mathtt{l}, \mathtt{r}, \mathtt{t}) = (\neg, \Box, \mathtt{r}, \mathtt{0}) \tag{6.16}$$

which follows from the fact that whenever a carry is produced, a digit less than $\mathtt{b}-\mathtt{1}$ must also be produced since, for any $\alpha, \beta \in \mathbb{L}$,

$$\alpha + \beta \leq 2(\mathtt{b}-\mathtt{1}) = \mathtt{b} + (\mathtt{b}-\mathtt{2}).$$

By inspection $(\neg, \Box, \mathtt{r}, \mathtt{0})$ cannot be produced via $\mathtt{f}$ or

$$(\mathtt{c}_1, \mathtt{l}_1, \mathtt{r}_1, \mathtt{t}_1) \oplus (\mathtt{c}_2, \mathtt{l}_2, \mathtt{r}_2, \mathtt{t}_2)$$

as neither $(\mathtt{c}_1, \mathtt{l}_1, \mathtt{r}_1, \mathtt{t}_1)$ nor $(\mathtt{c}_2, \mathtt{l}_2, \mathtt{r}_2, \mathtt{t}_2)$ can have the form $(\neg, \Box, \mathtt{r}, \mathtt{0})$. For $\mathtt{l} = \Box$ and $\mathtt{t} = \mathtt{0}$ is possible only via equation $\mathtt{6.15(c)}$ since in all other cases $\mathtt{r}_1 \neq \Box$ forces $\mathtt{l} \neq \Box$. But then $\mathtt{c}_1 = \neg$, $\mathtt{l}_1 = \Box$, and $\mathtt{t}_1 = \mathtt{0}$, a contradiction since a carry implies $\mathtt{t}_1 = \mathtt{1}$.

For any valid quadruple $(\mathtt{c},\mathtt{l},\mathtt{r},\mathtt{t})$, if $\mathtt{l} \neq \square$ then $\mathtt{r}$ must stop any carry, i.e.,

$$\mathtt{l} \neq \square \Rightarrow \mathtt{t} = 1. \tag{6.17}$$

Assume $(\mathtt{c},\mathtt{l},\mathtt{r},\mathtt{t})$ is valid and

$$\mathtt{l} \neq \square. \tag{6.18}$$

By inspection, this quadruple cannot be produced by an application of $\mathtt{f}$ (as $\mathtt{f}$ produces $\mathtt{l} = \square$). Suppose (bwoc) that $\mathtt{t} = 0$ and the quadruple is obtained via

$$(\mathtt{c}_1,\mathtt{l}_1,\mathtt{r}_1,\mathtt{t}_1) \oplus (\mathtt{c}_2,\mathtt{l}_2,\mathtt{r}_2,\mathtt{t}_2) = (\mathtt{c},\mathtt{l},\mathtt{r},\mathtt{t})$$

where (by supposition) the implication (6.17) holds for the operands $(\mathtt{c}_1,\mathtt{l}_1,\mathtt{r}_1,\mathtt{t}_1)$ and $(\mathtt{c}_2,\mathtt{l}_2,\mathtt{r}_2,\mathtt{t}_2)$. By next examining all possibilities, it is possible to arrive at a contradiction in every case, thereby verifying implication (6.17).

CASE $\mathtt{t}_1 = 0, \mathtt{t}_2 = 0$: By supposition, $\mathtt{l}_1 = \mathtt{l}_2 = \square$ as the implication 6.17 holds for the operands. If $\mathtt{c}_2 = \square$, then equation $6.15(\mathtt{c})$ applies and $\mathtt{l} = \square$, which violates the assumption (6.18). So $\mathtt{c}_2 \neq \square$, which is impossible as it implies an invalid form for the second operand (6.16).

CASE $\mathtt{t}_1 = 1, \mathtt{t}_2 = 0$: By hypothesis, $\mathtt{l}_2 = \square$ (otherwise $\mathtt{t}_2 = 1$). If $\mathtt{c}_2 = \square$, then equation $6.15(\mathtt{c})$ applies and $\mathtt{t} = 1$, which contradicts the supposition. So $\mathtt{c}_2 \neq \square$, which is impossible as it implies an invalid form for the second operand (6.16).

CASE $\mathtt{t}_1 = 0, \mathtt{t}_2 = 1$: By premise, $\mathtt{l}_1 = \square$ (otherwise $\mathtt{t}_1 = 1$). If $\mathtt{c}_2 = \square$, then equation $6.15(\mathtt{d})$ applies and $\mathtt{t} = 1$, which contradicts the supposition. So $\mathtt{c}_2 \neq \square$ and equation $6.15(\mathtt{a})$ applies, yielding $\mathtt{t} = 1$, a contradiction.

CASE $\mathtt{t}_1 = 1, \mathtt{t}_2 = 1$: It is immediate that $\mathtt{t} = 1$ as the tag is always equal to one of the tags, which violates the supposition.

By inspection, the solution is correct provided the correct baggage $(\mathtt{c},\mathtt{l},\mathtt{r},\mathtt{t})$ is produced so that

$$\mathtt{c} \# \mathtt{l} \# \mathtt{r}$$

is the correct result. This can be established in two phases, just as the solution is divided into two phases (map and reduction). First show that the function $\mathtt{f}$ produces only correct baggage. Second show that the binary operator $\oplus$ produces correct baggage whenever the operands are correct.

Consider a single application of the function $\mathtt{f}$, which computes the sum $\alpha + \beta$ of two digits $\alpha, \beta \in \mathbb{L}$. Put

$$\mathtt{a}_1 = \alpha +_{\mathtt{c}} \beta \qquad \text{and } \mathtt{a}_0 = \alpha +_{\mathtt{d}} \beta$$

and $\mathtt{t} = \mathtt{a}_0 < \mathtt{b} - 1$. The correct result is

$$\begin{array}{ll} \overline{\mathtt{a}_1,\mathtt{a}_0} & \text{if } \mathtt{a}_1 = 1 \\ \overline{\mathtt{a}_0} & \text{if } \mathtt{a}_1 \neq 1 \end{array}$$

If $a_1 = 1$, then

$$f(\alpha, \beta) = (\neg, \square, \overline{a_0}, t)$$

and

$$\neg \# \square \# \overline{a_0} = \overline{a_1, a_0}.$$

If $a_1 \neq 1$, then

$$f(\alpha, \beta) = (\square, \square, \overline{a_0}, t)$$

and

$$\square \# \square \# \overline{a_0} = \overline{a_0}.$$

Hence, every application of f produces a quadruple which yields the correct string as well as the correct tag.

It remains to show that

$$(c_1, l_1, r_1, t_1) \oplus (c_2, l_2, r_2, t_2) = (c, l, r, t).$$

is valid whenever (c_1, l_1, r_1, t_1) and (c_2, l_2, r_2, t_2) are valid. Either $c_2 = \square$ or $c_2 \neq \square$.

CASE $c_2 = \square$: As $c_2 = \square$, r_1 is not modified. It follows that the correct string for the combined result is

$$c_1 \# l_1 \# r_1 \# l_2 \# r_2.$$

By inspection of the relevant cases (equation $6.15(c)$ and $6.15(d)$), it is evident that the correct string will be produced. Note in equation $6.15(c)$, $l_2 = \square$ since otherwise $t_2 = 1$ (6.17). The carry $c = c_1$ is correct since $c_2 = \square$ implies no new carry is produced. For equation $6.15(d)$,

$$l = l_1 \# r_1 \# l_2,$$

$r = r_2$, and $t = t_2$ are correct, since l_1 and l_2 are correct, $t_2 = 1$ implies $r = r_2$ will stop any carry and t depends only on $r = r_2$. For equation $6.15(c)$, $l = l_1$,

$$r = r_1 \# r_2,$$

and $t = t_1$ are correct, since l_1 is correct initially, and as $t_2 = 0$, r_2 will not stop any carry; consequently, t depends only on r_1.

CASE $c_2 \neq \square$: As there is a carry, r_1 must be replaced by r_1'. In all relevant cases (equation 6.15(a) and equation $6.15(b)$), $r = r_2$ and $t = t_2$. Note $t_2 = 0$ is impossible since otherwise in order that the second operand has a valid form (see (6.16)) $l_2 \neq \square$ which forces $t_2 = 1$ (6.17), a contradiction. The tag is correct because it was correct initially (depends only on r_2). If equation $6.15(a)$ applies, then $c = c_2$ as $l_1 = \square$ (otherwise $t_1 = 1$), $t_1 = 0$ implies r_1 will not stop the carry, and $c_1 = \square$ since the first operand cannot have an invalid form (6.16). In addition, by the implication (6.17), $t_2 = 1$ and r_2 will stop any future carry implying that $r = r_2$. The resulting string

$$c_2 \# r_1' \# l_2 \# r_2$$

is correct since $c_1 = l_1 = \square$ and r_1 does not stop the carry. If equation 6.15(b) applies, then $t_1 = 1$ implies r_1 stops the carry; consequently, the carry $c = c_1$ is unchanged. It follows that the resulting string

$$c_1 \# l_1 \# r_1' \# l_2 \# r_2$$

is correct. ■

EXAMPLE 6.5 Consider $99999 + 99199 = 199198$ (decimal system). Calculate

$$\begin{aligned}
&\mathtt{unbag} \circ (/\oplus \circ * \mathtt{f}) \circ \underset{\longleftrightarrow}{\mathtt{id}} (\overline{9,9,1,9,9}, \overline{9,9,9,9,9}) \\
=&\mathtt{unbag} \circ (/\oplus \circ * \mathtt{f}) \overline{(9,9),(9,9),(1,9),(9,9),(9,9)} \\
=&\mathtt{unbag} \circ /\oplus \\
&[(\urcorner, \square, \overline{8}, 1), (\urcorner, \square, \overline{8}, 1), (\urcorner, \square, \overline{0}, 1), (\urcorner, \square, \overline{8}, 1), (\urcorner, \square, \overline{8}, 1)] \\
=&\mathtt{unbag}((\urcorner, \overline{9}, \overline{8}, 1) \oplus (\urcorner, \urcorner, \overline{8}, 1) \oplus (\urcorner, \square, \overline{8}, 1)) \text{ by equation } 6.15(b) \\
=&\mathtt{unbag}((\urcorner, \overline{9,9,1}, \overline{8}, 1) \oplus (\urcorner, \square, \overline{8}, 1)) \text{ by equation } 6.15(b) \\
=&\mathtt{unbag}(\urcorner, \overline{9,9,1,9}, \overline{8}, 1) \text{ by equation } 6.15(b) \\
=&\overline{1,9,9,1,9,8}.
\end{aligned}$$

Adding all the digits ignoring the carries requires n additions and then adding in the carries requires approximately n additions, yielding a total of about $2n$ additions. From the definition of f, at least n additions are performed plus n comparisons for computing the tags, and using the unit cost model, the cost of f alone is about $2n$. Looking at the reduce operation $\oplus$, there are at most five operations involving comparisons, concatenations or "toggling" and assuming approximately equal cost of these operations the total reduction costs about $5n$ (tree algorithm). Note that although toggling is actually a map, it is never possible to toggle more than a cumulative total of n digits. Hence the computational complexity is $O(n)$. The reduction takes $O(\log n)$ time steps via a binary tree algorithm and the map as well as other processing takes only constant time.

Consider the same problem using a sequential model of computation.

DEFINITION 6.9 The binary operator $\uplus$ obeys

$$(\alpha, \beta) \uplus ((\cdot)(c) \# x) = \overline{\alpha +_c \beta +_c c, \alpha +_d \beta +_d c} \# x. \tag{6.19}$$

A sequential digit adder is defined next.

Program AddStringsSEQ;

The function $\uplus$ is defined in equation (6.19).

$$\mathtt{AddStringsSEQ} = (\leftharpoonup \uplus) \circ \underset{\longleftrightarrow}{\mathtt{id}}.$$

6.4.2 USUAL MULTIPLICATION

The problem is to multiply two strings from $\mathbb{L}*$, say

$$\alpha_m\alpha_{m-1}\cdots\alpha_0 \qquad \text{and} \qquad \beta_n\beta_{n-1}\cdots\beta_0.$$

Without loss of generality, assume $m \le n$ (otherwise relabel using commutativity of multiplication). It is necessary to investigate how the carries propagate from right to left. The number of digits in the result is at most $m+n+2$ as

$$\sum_{i=0}^{m}\alpha_i b^i \sum_{i=0}^{n}\beta_i b^i \le \left(b^{m+1}-1\right)\left(b^{n+1}-1\right) = b^{m+n+2}-b^{m+1}-b^{n+1}+1$$
$$< b^{m+n+2}$$

implies

$$\sum_{i=0}^{m}\alpha_i b^i \sum_{i=0}^{n}\beta_i b^i = \sum_{k=0}^{m+n+1}\lambda_k b^k$$

for some λ_k's in $\mathbb{L}$. The task is to calculate these λ_k's.

Consider the net depicted in Figure 6.6(a). The αs are passed to the right and the βs are passed down the net. A horizontal row in this net corresponds to multiplying some digit α_i by all of the digits β_j, $0 \le j \le n$, just as rows are computed using the usual multiplication algorithm. Similarly, diagonals ($\searrow$) in this net correspond to the columns obtained using the usual multiplication algorithm by aligning the calculated rows.

In particular, each node in Figure 6.6(c) computes $c'_{i,j}$ and $d'_{i,j}$ by setting $c_{i,j}+d_{i,j}+\alpha_i\beta_j$ equal to $c'_{i,j}b+d'_{i,j}$ as illustrated in Figure 6.6(a). The sum computed by a node will produce exactly two digits since

$$c_{i,j}+d_{i,j}+\alpha_i\beta_j \le (b-1)+(b-1)+(b-1)^2 = (b-1)b+(b-1).$$

A node in Figure 6.6(d) corresponds to a quintuple (β, α, t, c, d). The digit β comes from the right string and the digit α from the left string. The tag t serves a dual purpose. First, it records state information indicating whether or not a node is ready to perform a computation. Second, the tag records information about the spread of data in order to guarantee associativity. Viewing the spread of data through the net for quintuples as a reduction, for the reduce operation to be associative, it must not matter whether β and c are passed before or after α. Accordingly, a tag $t = 0$ indicates a node is inactive whereas a tag $t = 1$ indicates a node is ready to update the carry c and the cumulative sum d as in Figure 6.6(c). In addition, a tag $t = 2$ indicates β and c must be passed to the left node and α to the right node. A tag

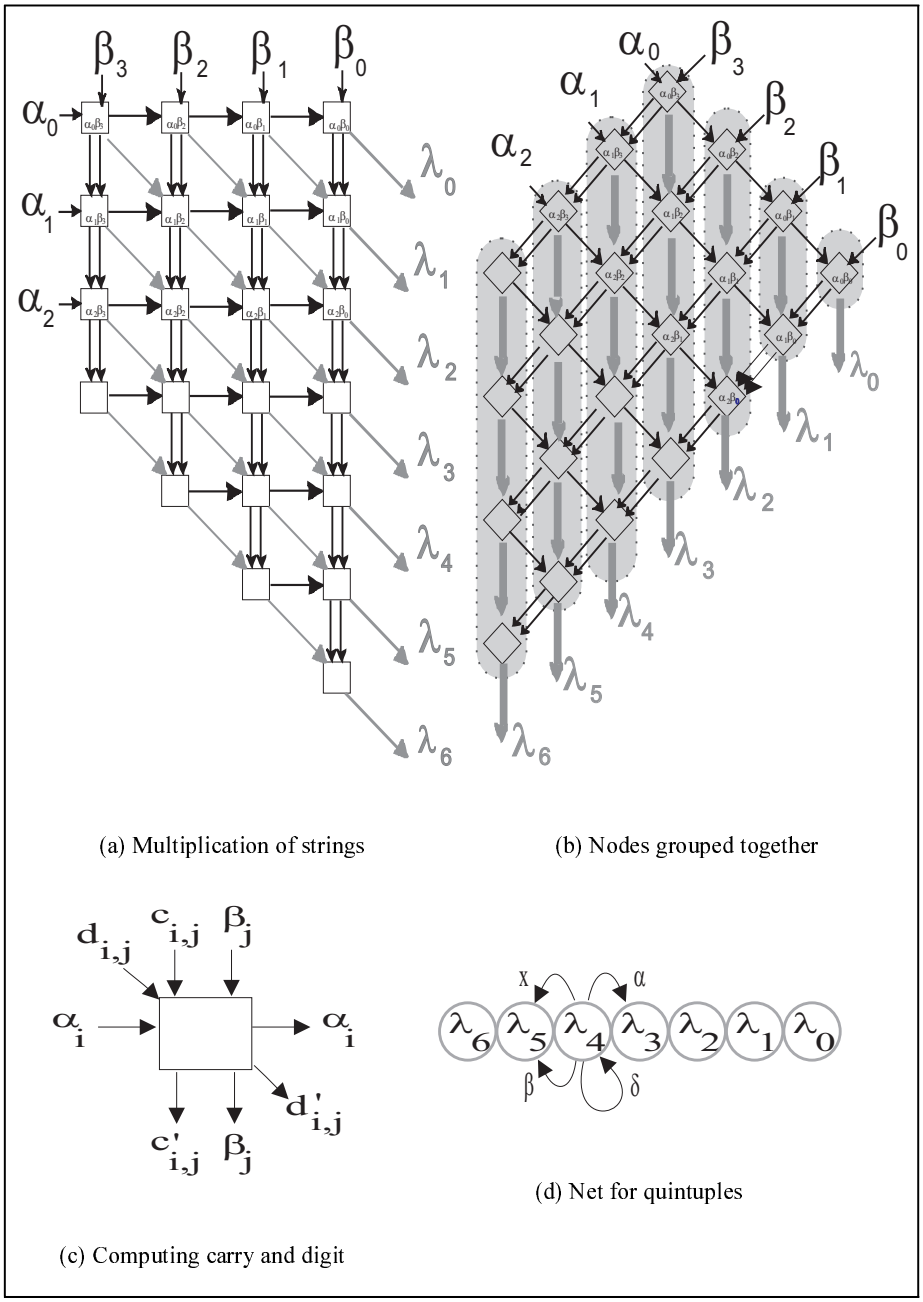


Figure 6.6: Rotate the diagram (a) clockwise so that the diagonals are vertical as shown in diagram (b). In the latter diagram the nodes are grouped vertically and then "collapsed" to obtain the new graph (d), which shows only the edges for one node. Each node computes the carry part $c'_{i,j}$ and the digit part $d'_{i,j}$ as illustrated in (c).

$t = 3$ indicates β and c must be passed to the left node and α has already been passed to the right node. A tag $t = 4$ indicates β and c have already been passed

to the left node and α must still be passed to the right node. The pre-processing is done via dscat which initializes the components of the quintuples using the inputs indicated in Figure 6.6(a) and which is defined next.

DEFINITION 6.10 Setting

$$\theta_k = \begin{cases} (0,0,0,0,0) & \text{for } k = m+1 \\ (0,\alpha_k,0,0,0) & \text{for } 1 \le k \le m \\ (\beta_n,\alpha_0,1,0,0) & \text{for } k = 0 \\ (0,\beta_{k+n},0,0,0) & \text{for } -n \le k \le -1 \end{cases},$$

put

$$\text{dscat}\left(\overline{\alpha_m,\ldots,\alpha_0},\overline{\beta_n,\ldots,\beta_0}\right) = \overline{\theta_{m+1},\theta_m,\ldots,\theta_{-n}}. \tag{6.20}$$

The main operations in the homomorphic solution are defined next.

DEFINITION 6.11 The binary operator $\oplus$ is given by

$$\begin{aligned} &x \# \overline{(\beta_1,\alpha_1,t_1,c_1,d_1)} \oplus \overline{(\beta_2,\alpha_2,t_2,c_2,d_2)} \# y \\ &= \begin{cases} x \# \overline{(\beta_1,0,3,c_1,d_1),(\beta_2,\alpha_1,1,c_2,d_2)} \# y & \text{if } t_1 = 2 \\ x \# \overline{(\beta_1,0,0,c_1,d_1),(\beta_2,\alpha_1,1,c_2,d_2)} \# y & \text{if } t_1 = 4 \\ x \# \overline{(\beta_2,\alpha_1,1,c_2,d_1),(0,\alpha_2,4,0,d_2)} \# y & \text{if } t_2 = 2 \\ x \# \overline{(\beta_2,\alpha_1,1,c_2,d_1),(0,\alpha_2,0,0,d_2)} \# y & \text{if } t_2 = 3 \\ x \# \overline{(\beta_1,\alpha_1,t_1,c_1,d_1),(\beta_2,\alpha_2,t_2,c_2,d_2)} \# y & \text{otherwise} \end{cases} \end{aligned} \tag{6.21}$$

The function for the map in the homomorphic solution is $f = (\cdot) \circ g$ forms singletons using g:

$$f(x) = \overline{g(x)}. \tag{6.22}$$

The operator $\oplus$ is well-defined and associative. Some post-processing is done using the function g and ap.

DEFINITION 6.12 The function g does the computations in Figure 6.6(c):

$$g(\beta,\alpha,t,c,d) = \begin{cases} (\beta,\alpha,2,c',d') & \text{if } t = 1 \\ (\beta,\alpha,t,c,d) & \text{otherwise} \end{cases}, \tag{6.23}$$

where c' and d' satisfy

$$\beta\alpha + c + d = c'b + d'.$$

The function ap applies a given function to the first element of a 1-array:

$$\texttt{ap g}\ (\cdot)(\texttt{a}) +\!\!+ \texttt{xs} = (\,(\cdot)\,\texttt{g}(\texttt{a})) +\!\!+ \texttt{xs}. \tag{6.24}$$

The digit serial multiplication program is defined next.

Program dsmult;

Let ap, g, f, $\oplus$ and dscat be given by equations (6.24), (6.23), (6.22), (6.21) and (6.20), respectively. The post-processing function dz drops the initial zero if there is one, *lst selects the last digit from each quintuple, and $\texttt{p} = \texttt{m} + 2\texttt{n} - 2$.

$$\texttt{dsmult} = \texttt{dz} \circ (*\texttt{lst}) \circ (\texttt{ap g}) \circ (/\oplus \circ * \texttt{f})^{\texttt{p}} \circ \texttt{dscat}.$$

A proof that the program is correct is geometrical. The arrows in Figure 6.6(a) reveal that initially only one node (which has α_0 and β_n) is active. Subsequently, every adjacent node (i.e., nodes that were inactive in the last round) of an active node performs an update. Observe that

$$\texttt{t}_1 \in \{2, 4\} \Longrightarrow \texttt{t}_2 \notin \{2, 3\}, \tag{6.25}$$

whenever $\texttt{t}_1$ is a valid tag for some node and $\texttt{t}_2$ is the tag of the adjacent node on the right. This follows from the net and the semantics as a node on the right must be inactive whenever a node computes and is ready to pass an α to the right.

After applying dscat, the quintuples satisfy the stated semantics and all tags are valid. Moreover, implication (6.25) holds as all tags are zero except for one tag which is one. By construction, f performs the computation depicted in Figure 6.6(c). After applying the map *f, only those quintuples with tags that were equal to one ($\texttt{t} = 1$) will get their tags modified and the new value for such tags is two ($\texttt{t} = 2$). Before the first application of the reduction $/\oplus$, the tags possess the following two properties:

1. all internal nodes have tags in $\{0, 2\}$ and any internal node having tag equal to 2 has neighbors with tags in $\{0, 3, 4\}$;
2. the leftmost node has tag in $\{0, 2, 3\}$ and the rightmost node has tag in $\{0, 2, 4\}$.

Two tags are said to be adjacent if their corresponding nodes are adjacent. Consider a node whose tag is equal to two. After applying $\oplus$ on the left side of such a node, the tag is replaced by $\texttt{t} = 4$ and the adjacenttag on the left is set equal to one. After applying $\oplus$ on the right side of a node whose tag is equal to two, the tag is replaced by $\texttt{t} = 3$ and the adjacent tag on the right is set equal to one. In both cases, after evaluating on the other side, the tag is replaced by $\texttt{t} = 0$. After applying the map *f again, it follows that the same properties (1 and 2) must hold.

From Figure 6.6(a), it is clear that there are $n-1$ times that the αs must be passed to neighbors and only after α_0 is passed to the rightmost node, β_0 is propagated along. There are $n-1+m$ times that β_0 is passed until the last update is done via ap g.

EXAMPLE 6.6 Compute $99 \times 99 = 9801$(base 10). Let $h = \not\oplus \circ * f$, $p = 4$ and calculate as follows:

$$
\begin{aligned}
& h^4 \circ dscat\left(\overline{9,9},\overline{9,9}\right) \\
&= h^4\overline{(0,0,0,0,0),(0,9,0,0,0),(9,9,1,0,0),(9,0,0,0,0)} \\
&= h^3\left(\overline{(0,0,0,0,0)} \oplus \overline{(0,9,0,0,0)} \oplus \overline{(9,9,2,8,1)} \oplus \overline{(9,0,0,0,0)}\right) \\
&= h^3\left(\overline{(0,0,0,0,0),(0,9,0,0,0)} \oplus \overline{(9,0,3,8,1),(9,9,1,0,0)}\right) \\
&= h^3\overline{(0,0,0,0,0),(9,9,1,8,0),(0,0,0,0,1),(9,9,1,0,0)} \\
&= h^2\left(\overline{(0,0,0,0,0)} \oplus \overline{(9,9,2,8,9)} \oplus \overline{(0,0,0,0,1)} \oplus \overline{(9,9,2,8,1)}\right) \\
&= h^2\left(\overline{(9,0,1,8,0),(0,9,4,0,9)} \oplus \overline{(9,0,1,8,1),(0,9,4,0,1)}\right) \\
&= h^2\overline{(9,0,1,8,0),(0,0,0,0,9),(9,9,1,8,1),(0,9,4,0,1)} \\
&= h\left(\overline{(9,0,2,0,8)} \oplus \overline{(0,0,0,0,9)} \oplus \overline{(9,9,2,9,0)} \oplus \overline{(0,9,4,0,1)}\right) \\
&= h\left(\overline{(9,0,3,0,8),(0,0,1,0,9)} \oplus \overline{(9,0,3,9,0),(0,9,1,0,1)}\right) \\
&= h\,\overline{\underline{(9,0,3,0,8)},\underline{(9,0,1,9,9)},\underline{(0,0,0,0,0)},\underline{(0,9,1,0,1)}} \\
&= \underline{(9,0,3,0,8)} \oplus \underline{(9,0,2,1,8)} \oplus \underline{(0,0,0,0,0)} \oplus \underline{(0,9,2,0,1)} \\
&= \overline{(9,0,1,1,8),(0,0,4,0,8)} \oplus \overline{(0,0,1,0,0),(0,9,4,0,1)} \\
&= \overline{(9,0,1,1,8),(0,0,0,0,8),(0,0,1,0,0),(0,9,4,0,1)}.
\end{aligned}
$$

Lastly,

$$
\begin{aligned}
& dz \circ (*\,lst) \circ ap\ g \\
& \qquad \overline{(9,0,1,1,8),(0,0,0,0,8),(0,0,1,0,0),(0,9,4,0,1)} \\
&= dz \circ (*\,lst)\overline{(9,0,2,0,9),(0,0,0,0,8),(0,0,1,0,0),(0,9,4,0,1)} \\
&= dz\ \overline{9,8,0,1} = \overline{9,8,0,1}.
\end{aligned}
$$

6.4.3 CONVOLUTION

The problem is to compute

$$\sum_{i=0}^{\min\{j,m\}} a_i x_{j-i} = b_j,$$

which resembles a linear system of equations except the x_j's are known. Instead of solving for the x_j's, it is desired to simply calculate the b_j's. Essentially the

x_j's are replaced by a weighted average of their neighbors. A valid interpretation of the computation is to view the weights as a window that slides over the data. Only the data in the window is used. Consider the net depicted on the left side of Figure 6.7. The weights (a_i's) are passed to the right. The data (x_j's) flows down the net. A horizontal row in this net corresponds to multiplying some weight a_i by all of the x_j's, $0 \leq j \leq n$. Diagonals in this net correspond to a sum b_j. A particular node in the net computes a product $a_i x_{j-i}$ and adds it to the sum b_j as depicted in Figure 6.7(c).

A node in Figure 6.7(d) corresponds to a quadruple (a, x, t, b). The data x comes from the right node and the weight a from the left node. The tag t serves a dual purpose. First, it records state information indicating whether or not a node is ready to perform a computation. Second, the tag records information in order to guarantee associativity. In order that the reduce operation is associative, it should be possible to pass the weight a before or after transmitting the data x.

Accordingly, let a tag $t = 0$ indicate that a node is inactive and a tag $t = 1$ indicate that a node is ready to update the cumulative sum b as in Figure 6.7(c). A tag $t = 2$ indicates x must be passed to the left node and weight a to the right node. A tag $t = 3$ indicates x must be passed to the left node and weight a has already been passed to the right node. Lastly, a tag $t = 4$ indicates x has already been passed to the left node and weight a must still be passed to the right node.

The pre-processing is accomplished via $\mathtt{preconv}$ defined next.

DEFINITION 6.13 Set

$$\theta_k = \begin{cases} (0, x_k, 0, 0) & \text{if } 1 \leq k \leq n \\ (a_m, x_0, 1, 0) & \text{if } k = 0 \\ (a_{m+k}, 0, 0, 0) & \text{if } -m \leq k \leq -1 \end{cases},$$

and

$$\mathtt{preconv}(\overline{a_0, \ldots, a_m}, \overline{x_0, \ldots, x_n}) = \overline{\theta_{-m}, \theta_{-m+1}, \ldots, \theta_n}. \tag{6.26}$$

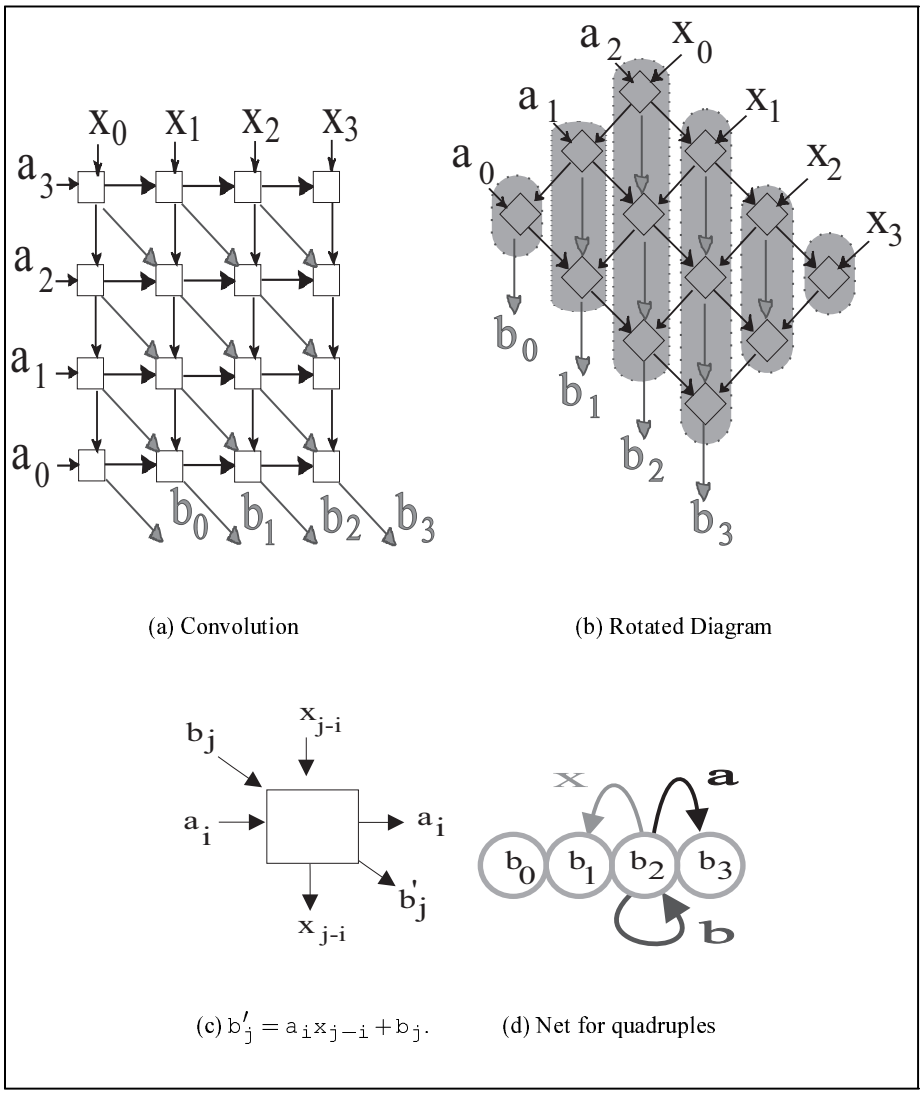


Figure 6.7: Rotate the diagram on the top left (a) clockwise so that the diagonals are vertical as shown on the right side (b). The nodes in (b) are grouped vertically. The diagram (b) is "collapsed" to obtain the new graph (d), i.e., a node in diagram (d) represents all of the nodes in a group (b). In diagram (d) only the edges for one node are shown.

DEFINITION 6.14 The binary operator $\oplus$ in the homomorphic solution is given by

$$x \# \overline{(a_1, x_1, t_1, b_1)} \oplus \overline{(a_2, x_2, t_2, b_2)} \# y \tag{6.27}$$

$$= \begin{cases} x \# \overline{(0, x_1, 3, b_1), (a_1, x_2, 1, b_2)} \# y & \text{if } t_1 = 2 \\ x \# \overline{(0, x_1, 0, b_1), (a_1, x_2, 1, b_2)} \# y & \text{if } t_1 = 4 \\ x \# \overline{(a_1, x_2, 1, b_1), (a_2, 0, 4, b_2)} \# y & \text{if } t_2 = 2 \\ x \# \overline{(a_1, x_2, 1, b_1), (a_2, 0, 0, b_2)} \# y & \text{if } t_2 = 3 \\ x \# \overline{(a_1, x_1, t_1, b_1), (a_2, x_2, t_2, b_2)} \# y & \text{otherwise} \end{cases}.$$

Post-processing involves a function $\mathtt{g}$ which is defined next.

DEFINITION 6.15 The function $\mathtt{g}$ is given by

$$\mathtt{g(a,x,t,b)} = \begin{cases} \mathtt{(a,x,2,b+ax)} & \text{if } \mathtt{t}=1 \\ \mathtt{(a,x,t,b)} & \text{if } \mathtt{t}\neq 1 \end{cases}. \tag{6.28}$$

The multiplication program for convolution is defined next.

Program conv;

The operations preconv, $\oplus$ and g are given by (6.26), (6.27) and (6.28), respectively, $*\,\mathtt{lst}$ selects the $\mathtt{b}$'s from each quadruple, $\mathtt{f}=(\cdot)\circ\mathtt{g}$ and $\mathtt{p}=\mathtt{m}+\mathtt{n}$.

$$\mathtt{conv} = (\mathtt{initial\ n}+1)\circ *(\mathtt{lst}\circ\mathtt{g})\circ(/\oplus\circ *\mathtt{f})^{\mathtt{p}}\circ\mathtt{preconv}$$

EXAMPLE 6.7 Compute the convolution using weights $\mathtt{a}_0=1,\ \mathtt{a}_1=2,\ \mathtt{a}_2=1$ and data $\mathtt{x}_0=2,\ \mathtt{x}_1=3,\ \mathtt{x}_2=5,\ \mathtt{x}_3=7$. Taking $\mathtt{p}=5$ and $\mathtt{h}=/\oplus\circ *\mathtt{f}$, calculate

$$\begin{aligned}
&\mathtt{h}^5\circ\mathtt{preconv}(\overline{1,2,1},\overline{2,3,5,7})\\
=&\ \mathtt{h}^5\,\overline{(1,0,0,0),(2,0,0,0),(1,2,1,0),(0,3,0,0),(0,5,0,0),(0,7,0,0)}\\
=&\ \mathtt{h}^4(\overline{(1,0,0,0)}\oplus\overline{(2,0,0,0)}\oplus\overline{(1,2,2,2)}\oplus\overline{(0,3,0,0)}\oplus\overline{(0,5,0,0)}\oplus\overline{(0,7,0,0)})\\
=&\ \mathtt{h}^4(\overline{(1,0,0,0),(2,0,0,0)}\oplus\overline{(0,2,3,2),(1,3,1,0)}\oplus\overline{(0,5,0,0),(0,7,0,0)})\\
=&\ \mathtt{h}^4(\overline{(1,0,0,0),(2,2,1,0),(0,0,0,2),(1,3,1,0)}\oplus\overline{(0,5,0,0),(0,7,0,0)})\\
=&\ \mathtt{h}^4\overline{(1,0,0,0),(2,2,1,0),(0,0,0,2),(1,3,1,0),(0,5,0,0),(0,7,0,0)}\\
=&\ \mathtt{h}^3(\overline{(1,0,0,0)}\oplus\overline{(2,2,2,4)}\oplus\overline{(0,0,0,2)}\oplus\overline{(1,3,2,3)}\oplus\overline{(0,5,0,0)}\oplus\overline{(0,7,0,0)})\\
=&\ \mathtt{h}^3\overline{(1,2,1,0),(0,0,0,4),(2,3,1,2),(0,0,0,3),(1,5,1,0),(0,7,0,0)}\\
=&\ \mathtt{h}^2(\overline{(1,2,2,2)}\oplus\overline{(0,0,0,4)}\oplus\overline{(2,3,2,8)}\oplus\overline{(0,0,0,3)}\oplus\overline{(1,5,2,5)}\oplus\overline{(0,7,0,0)})\\
=&\ \mathtt{h}^2\overline{(0,2,3,2),(1,3,1,4),(0,0,0,8),(2,5,1,3),(0,0,0,5),(1,7,1,0)}\\
=&\ \mathtt{h}(\overline{(0,2,3,2)}\oplus\overline{(1,3,2,7)}\oplus\overline{(0,0,0,8)}\oplus\overline{(2,5,2,13)}\oplus\overline{(0,0,0,5)}\oplus\overline{(1,7,2,7)})\\
=&\ \mathtt{h}\,\overline{(0,3,1,2),(0,0,0,7),(1,5,1,8),(0,0,0,13),(2,7,1,5),(1,0,4,7)}\\
=&\ \overline{(0,3,2,2)}\oplus\overline{(0,0,0,7)}\oplus\overline{(1,5,2,13)}\oplus\overline{(0,0,0,13)}\oplus\overline{(2,7,2,19)}\oplus\overline{(1,0,4,7)}\\
=&\ \overline{(0,3,3,2),(0,5,1,7),(0,0,0,13),(1,7,1,13),(0,0,0,19),(1,0,4,7)}.
\end{aligned}$$

Using map distributivity,

$$\begin{aligned}
&*(\mathtt{lst}\circ\mathtt{g})\,\overline{(0,3,3,2),(0,5,1,7),(0,0,0,13),(1,7,1,13),(0,0,0,19),(1,0,4,7)}\\
=&\ *\,\mathtt{lst}\,\overline{(0,3,3,2),(0,5,2,7),(0,0,0,13),(1,7,2,20),(0,0,0,19),(1,0,4,7)}\\
=&\ \overline{2,7,13,20,19,7},
\end{aligned}$$

and lastly,

$$(\mathtt{initial\ 4})\,\overline{2,7,13,20,19,7} = \overline{2,7,13,20}.$$

The correct solution is

$$\mathtt{b}_0=2,\ \mathtt{b}_1=7,\ \mathtt{b}_2=13,\ \text{and}\ \mathtt{b}_3=20.$$

6.5 SUMMARY

It seems the programming templates are useful. Frequently, the programs are as efficient as expected. It is possible to analyze costs using any appropriate model. The complexity analysis serves to develop measures for a particular program. Such analysis, however, cannot be used to evaluate the methodology. The complexity analysis reveals whether or not a program has the potential to meet the requirements and which parts of a program are relatively expensive and should be optimized. If the cost analysis is unsatisfactory, it does not necessarily mean that the methodology is wrong. It simply means another solution should be sought, perhaps investigating other models, computational patterns and algorithms.

Using complexity analysis, an implementation can be chosen and adapted to yield an optimal program. Increasingly in the future, computers might be custom-built based on such programs and analysis of them. By studying the programs, it is possible to discover applicable methods such as lazy evaluation which could be utilized to improve performance.

Optimization may be viewed as two phases. The first phase is accomplished via the programs. In this phase, the task is studied and models and templates are chosen to write an optimal program. In the second phase, the task is removed from the picture and only the programming solution is studied to do further optimizations.

It is easy and natural to develop programs using near homomorphisms. The problems can be partitioned in a natural way using the templates for map and reduce. The formalism furnishes the needed tools for analysis. Experience solving one problem like multiplication of strings helps when solving another problem like convolution. It seems homomorphisms offer quite versatile applications in computer science.

CHAPTER 7

FUTURE WORK

A concise calculus for generic programming of multidimensional arrays has been presented. A model, terminology and simple notation to facilitate reasoning about multidimensional arrays have been introduced. A set of primitive operations has been constructed. A methodology to build such a set has been prescribed. Higher-order functions have been provided to recognize a variety of program structures for the particular data structure. Homomorphisms have been classified. A number of explicit formulas that lack many of the indices typically required have been stated. Programming examples have been given.

Mainly, research focuses on developing complete theories for any particular data type. An existing theory might be modified to solve particular classes of problems. Higher-order functions that model different computational patterns should be investigated.

Consider the following problem: find the rightmost list element that is smaller than its neighbor on the right.

EXAMPLE 7.1 In $\overline{1,2,3,4,8,7,6,5}$, the element 4 is the desired element. In the worst case, it is necessary to compare all adjacent elements. To do Therefore, a comparison operator must be placed between all adjacent elements. This implies that the interior elements are used twice whereas the exterior elements are used only once.

EXAMPLE 7.2 For $\overline{1,2,3,4,8,7,6,5}$, all of the following comparisons may be performed:

$$1 < 2, 2 < 3, 3 < 4, 4 < 8, 8 < 7, 7 < 6, 6 < 5.$$

This problem motivates a new splitting defined next.

DEFINITION 7.1 The join operator $\|$ obeys

$$x \| y = xs +\!\!+ (\cdot)(a) +\!\!+ ys$$

provided

$$\begin{aligned} x &= xs +\!\!+ (\cdot)(a) \neq (\cdot)(a), \\ y &= (\cdot)(a) +\!\!+ ys \neq (\cdot)(a). \end{aligned}$$

Empty strings $xs = \square$ and $ys = \square$ are disallowed in the definition of the join operator. The join operator is a well-defined associative operator.

EXAMPLE 7.3 $\overline{1,2,3,4} = \overline{1,2} \| \overline{2,3,4} = \overline{1,2,3} \| \overline{3,4}$.

EXAMPLE 7.4 $\left(\overline{1,2} \| \overline{2,3}\right) \| \overline{3,4} = \overline{1,2,3,4} = \overline{1,2} \| \left(\overline{2,3} \| \overline{3,4}\right)$.

A new operation $=\!\!=$ places a given operator between all the elements.

DEFINITION 7.2 For any binary operator $\oplus$,

$$\begin{aligned} =\!\!= \oplus [a : 2] &= (\cdot)(a_1 \oplus a_2), \\ =\!\!= \oplus (x \| y) &= (=\!\!= \oplus x) +\!\!+ (=\!\!= \oplus y). \end{aligned}$$

The function $=\!\!=$ is not defined for 1-arrays with fewer than two elements. The cost of $=\!\!=$ is the same as the cost of `map` since all binary operations may be computed concurrently.

DEFINITION 7.3 A function h is a $\|$**-homomorphism** if there exists an associative binary operator $\oplus$ such that

$$h(x \| y) = h(x) \oplus h(y)$$

for all valid x and y.

EXAMPLE 7.5 Define $\oplus$ by

$$a \oplus b = a + b - 1 \text{ for all } a, b \in \mathbb{Z}.$$

Then length is a $\|$-homomorphism since

$$\text{length}\,(x \| y) = \text{length}\ x \oplus \text{length}\ y.$$

EXAMPLE 7.6 The function $*\mathtt{f}$ is a $\|$-homomorphism:

$$*\mathtt{f}(\mathtt{x} \parallel \mathtt{y}) = (*\mathtt{f}\ \mathtt{x}) \parallel (*\mathtt{f}\ \mathtt{y}).$$

Assume the desired result may be found by comparing adjacent elements and then searching through the results of these comparisons. Various algorithms, including linear and binary tree algorithms are well-known. Is there a satisfactory solution using algebraic structures? Consider the following programming template:

$$(/\oplus) \circ (=\!\!\odot),$$

in which the operation $=\!\!\odot$ places the comparison operator between all the elements and then the reduction $/\oplus$ performs the search. Assuming $=\!\!\odot$ can be performed in constant time and $/\oplus$ is implemented using a binary tree, the above program has time complexity $\mathtt{O}(\log \mathtt{n})$, where $\mathtt{n}$ denotes the length of the argument.

A "binary linear search from the right" is defined next.

DEFINITION 7.4

$$\mathtt{blsr} \odot \mathtt{x} +\!\!\!+ [\mathtt{a} : 2] = \begin{cases} \mathtt{a}_1 & \text{if } \mathtt{a}_1 \odot \mathtt{a}_2 \\ \mathtt{blsr} \odot \mathtt{x} +\!\!\!+ [\mathtt{a} : 1] & \text{otherwise} \end{cases}.$$

EXAMPLE 7.7 Compute

$$\mathtt{blsr} < \overline{\mathtt{1,2,3,4,8,7,6,5}} = 4.$$

In order to fit into the specified programming template, the baggage needs to be set up properly.

DEFINITION 7.5 The binary comparison operator $\prec$ is given by

$$\mathtt{a} \prec \mathtt{b} = (\mathtt{a}, \mathtt{a} < \mathtt{b}).$$

The search is performed via a reduction using the binary operator defined next.

DEFINITION 7.6 The binary operator $\trianglelefteq$ is given by

$$(\mathtt{a},\mathtt{b}) \trianglelefteq (\mathtt{c},\mathtt{d}) = \begin{cases} (\mathtt{c},\mathtt{d}) & \text{if } \mathtt{d} = 1 \\ (\mathtt{a},\mathtt{b}) & \text{otherwise} \end{cases}.$$

The program $\mathtt{blsrc} = \mathtt{blsr} \prec$ is defined next.

Program blsrc;

The operations $\prec$ and $\trianglelefteq$ are given by (7.5) and (7.6), respectively and $\mathtt{fst}(\mathtt{a},\mathtt{b}) = \mathtt{a}$.

$$\mathtt{blsrc} = \mathtt{fst} \circ (/ \trianglelefteq) \circ (= \prec).$$

EXAMPLE 7.8 Compute

$$\begin{aligned}
& (/ \trianglelefteq) \circ (= \prec)\ \overline{1,2,3,4,8,7,6,5} \\
= \ & (/ \trianglelefteq)\ \overline{(1,1),(2,1),(3,1),(4,1),(8,0),(7,0),(6,0)} \\
= \ & ((1,1) \trianglelefteq (2,1)) \trianglelefteq ((3,1) \trianglelefteq (4,1)) \trianglelefteq ((8,0) \trianglelefteq (7,0)) \trianglelefteq (6,0) \\
= \ & ((2,1) \trianglelefteq (4,1)) \trianglelefteq ((8,0) \trianglelefteq (6,0)) \\
= \ & (4,1) \trianglelefteq (8,0) \\
= \ & (4,1)
\end{aligned}$$

and

$$\mathtt{fst} \circ (/ \trianglelefteq) \circ (= \prec)\ \overline{1,2,3,4,8,7,6,5} = \mathtt{fst}\ (4,1) = 4.$$

EXAMPLE 7.9 Suppose that instead of finding the element 4 in the previous example, it is desired to find the index of the element, which is 4. The baggage is

$$(\mathtt{lf},\mathtt{lt})$$

where $\mathtt{lf}$ denotes the length found and $\mathtt{lt}$ denotes the total length. Redefine the binary operators so that

$$\mathtt{a} \prec \mathtt{b} = (\mathtt{a} < \mathtt{b}, 1)$$

and

$$(\mathtt{a},\mathtt{b}) \trianglelefteq (\mathtt{c},\mathtt{d}) = \begin{cases} (\mathtt{b}+\mathtt{c},\mathtt{b}+\mathtt{d}) & \text{if } \mathtt{c} \neq 0 \\ (\mathtt{a},\mathtt{b}+\mathtt{d}) & \text{otherwise} \end{cases}.$$

Then

$$\begin{aligned}
& \mathtt{fst} \circ (/ \trianglelefteq) \circ (= \prec)\ \overline{1,2,3,4,8,7,6,5} \\
= & \mathtt{fst} \circ (/ \trianglelefteq)\ \overline{(1,1)(1,1)(1,1)(1,1)(0,1)(0,1)(0,1)} \\
= & \mathtt{fst}(((1,1) \trianglelefteq (1,1)) \trianglelefteq ((1,1) \trianglelefteq (1,1)) \trianglelefteq ((0,1) \trianglelefteq (0,1)) \trianglelefteq (0,1)) \\
= & \mathtt{fst}((2,2) \trianglelefteq (2,2) \trianglelefteq (0,2) \trianglelefteq (0,1)) \\
= & \mathtt{fst}((4,4) \trianglelefteq (0,3)) \\
= & \mathtt{fst}(4,7) \\
= & 4.
\end{aligned}$$

This overlapping approach based on the join operator could be applied to a class of problems. Other constructors might be invented which would solve other classes of

problems. In principle, many computational patterns could be incorporated into useful programming templates (involving appropriate constructors).

An empty object was given as a generator. Generators are adequate whenever dealing with only one object. It is unnecessary to invent a function merely to manage a single object. An empty array was employed for algebraic reasons. A benefit is the simplification of definitions and programs. For instance, any nonempty 1-array could be written in the form

$$(\cdot)(x) +\!\!+ A$$

and so definitions and programs for any nonempty 1-array could be written using only this form. In practice there are trade-offs and an empty object is not strictly required.

An empty object is a useful object. When instructed to remove all the items, it is appropriate to return an empty object as opposed to an undefined one. There may be more tolerance for unexpected results as operations should always be defined for an empty object whenever it is admitted into a formalism.

Although arrays possess a single array type, it is possible to discuss particular classes of arrays in useful ways. Many sorted arrays or different types could be defined. For instance, it is possible to define an infinite family of empty arrays $\{[\ ]_n\}$ so that $[\ ]_n$ has the new type n-array (instead of defining just one empty array with a single type), an infinite family of empty functions

$$\varepsilon_n : \text{unit} \longrightarrow n\text{-array}$$

by $\varepsilon_n(1) = [\ ]_n$, reax_k as $\text{reax}_{k,n}$, which applies only to n-arrays, and $+\!\!+_k$ as $+\!\!+_{k,n}$ which applies only n-arrays. Using more than one type should be avoided, since the notation may become more cumbersome while the power of the formalism may not be significantly improved. Simple notation and unifying ideas are beneficial.

Notwithstanding the fact that an array model is a versatile model, it is not all encompassing. For example, an infinite sequence,

$$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \ldots$$

does not fit into an array model. Nevertheless, the specification of infinite arrays which could be used for parameters to operations was discussed.

EXAMPLE 7.10 The infinite 1-array

$$[f : \infty]$$

cannot be an argument to a map since evidently the result would be an infinite array.

However, this infinite array could be a parameter to $\multimap$ (bicomp) provided the argument is a valid 1-array.

A lesson from mathematics is that it is not wise to mix finite and infinite cases. Although only the finite case was studied, it is possible to develop an algebra for infinite arrays. An array may regarded as infinite if it can always be extended. This idea seems reasonable. Limits and sequences may be approached in the same way. A closely related notion is lazy evaluation for infinite lists. In any case, an array can always be extended.

Arrays were endowed with regular structures that resemble boxes. Operations may utilize irregular structures within an array. Theories may be developed for irregular structures. Instead of trying to construct one universal model, build different models by finding the right match between the objects and the operations on them.

If several ADTs are available, choose one based upon the common operations that are performed. The choice hopefully reduces the overall cost(s) in a significant way. For any particular ADT, only a couple access functions may be relatively inexpensive compared to other primitives. When developing an algebraic theory, look at models and primitives based on the common operations that are needed. After analysis of the kind of operations that are needed, try to find similarities between the data structure and the program structures.

An ADT solves only a small part of the software dilemma. Essentially the abstraction is not high enough away from implementation. The data type is hidden behind access functions so that it is possible to easily rewrite operations or modify the implementation. However, no tools are provided to help program in the first place, which is really what is needed.

Consider sets. Various ADTs for sets are well-known. Moreover, numerous implementations are available. Actually the algebra of sets is not complicated and many ADTs incorporate useful functions (such as union and intersection). Therefore, ADTs can be successful. Yet the diversity of existing data types attests to the fact that not all algebras are so simple.

Programming is at the right level for formal program development. In principle, solutions, especially implicitly parallel ones, may be calculated starting from the specification of a task. The efficiency of a solution may be improved or geared for a particular computer model by rewriting the program using laws, which is another benefit of the approach. In short, the formalism is at the right level for software development because it provides a useful bridge between tasks and computers.

There are many factors to consider besides particular programming solutions. Any

part of a program can be optimized for existing or future computers. Cost analysis is possible. By studying the programs instead of the tasks, it is possible to optimize for any particular implementation. While the programs are completely specified, they allow the freedom to optimize without imposing unneeded details. Engineers can build computers based on the programs, instead of trying to program based on existing computers or languages. The freedom to implement in different ways is important for optimization (cf. for example Axford and Joy, 1993, p. 16 on list processing).

There are existing software packages that work quite well. It may be possible to map primitive operations in the theory to an ADT, or adapt an existing ADT, and find a corresponding implementation in PASCAL or some other desired language. It is always possible to modify or extend an existing theory. Thus, while much work lies ahead, it is likely that there are many libraries, packages and tools that can be used.

There are challenges ahead if algebraic theories are going to be used on a large scale. There may be resistance to moving away from traditional languages. It is unlikely that there is some panacea and that the problems are simply going to be resolved by teaching new languages or even finding more applications.

It is important to learn to get away from the mind-set of the way research is done, especially the way functions and sets are employed, not just the way programming is done. Employing other formal notions is one of the attractions of category theory. Nevertheless, the proper framework for an algebraic theory does not lie so much in any general theory but rather in the discovery of the right category for each theory.

Be wary of any claims of deep results using category theory as pure theories are too general. Homomorphisms by definition concern mainly objects and functions that are equivalent in some sense. If the theory is in fact so general then it is unlikely that adequate tools are provided which reveal how the theory can be applied in practice. It is paramount that the theory is relevant and aids in programming.

The programming language should be sufficiently developed. Do not look for practical results based on a theory that is not specialized enough. Especially, it does not seem promising if the foundation is based on a complex model or if the theory does not account many common operations that are appropriate for the data structure.

A satisfactory theory balances abstraction and application. A lesson for computer scientists from the previous millennium is that software should be designed at the right level of abstraction. Poor design involves only programming languages, “hacking” and reverse engineering. On the other hand, theories are overrated, despite their elegance, whenever they subsume the data type. A favorable proposition is progressive software design that begins with a data type and ends with an algebraic theory which matches

suitable program structures with the data structure.

Take the most general point of view possible, even though the model is specific. In other words, try to get the most out of a specific model. At the same time, avoid introducing notation and classifying every algebraic structure whenever doing so comes at too high a price relative to the benefits that are reaped. It seems a daunting task to attempt to classify every operation and every possible structure. Moreover, the benefits of doing so are unclear, especially in a larger framework that is manageable in size. The goal is not to develop algebra or category theory but instead to apply the relevant ideas whenever beneficial to do so.

Using the same basic model of an array it is possible to choose different constructors. There may be good reasons for using different constructors. It is unlikely that one theory would suffice as costs often depend on the common operations that are performed.

The primitives should be carefully chosen since the whole theory is built on this foundation. It is especially important to choose suitable constructors so that it is possible to obtain useful programming templates for generic programming. A benefit of using algebra, mainly homomorphisms, is all the parts of the theory that come for free. Once the primitives have been chosen, much of the programming falls into place. Some trial and error will likely be needed. There is this controlling mechanism so that the language cannot expand in some chaotic fashion. When developing a formalism and it is not possible to define something in terms of the primitives as desired, it is time to go back and think more carefully about them. When defining a function, in order for it to fit into the theory some choices are often fairly clear right from the start. Even when it is not immediately clear how to proceed, most or all issues are eventually resolved. Many functions are automatically programmed by merely filling-in the necessary parts. For instance, given the equations

$$\begin{aligned} \mathtt{h}((\cdot)_{\mathtt{n}}(\mathtt{x})) &= \mathtt{u}(\mathtt{x}), \\ \mathtt{h}(\mathtt{A} +\!\!+_{\mathtt{k}} \mathtt{B}) &= \mathtt{h}(\mathtt{A}) \oplus \mathtt{h}(\mathtt{B}), \end{aligned}$$

by the unique extension property, it suffices to specify $\mathtt{u}$ and $\oplus$ in order to program $\mathtt{h}$. It is difficult or impossible to provide such programming templates in a low-level language such as PASCAL as it is necessary to specify types and details that are nonessential. On the other hand, such templates can be implemented in any language such as PASCAL. Due to type constraints of a particular implementation it may be necessary to copy the code for a particular template for various types; however, all such procedures are copies except for types.

Homomorphisms are not the only templates for generic programming. The homo-

morphism lemma has practical application because it provides a way to partition the problem and thereby obtain a solution. Perhaps other higher-order functions can be found which yield a different type of solution. Any parallel algorithm could be studied in this context.

Despite all of the problems, general systems have proven successful. Nevertheless, it is important to recognize the limitations and settling only for the current state of affairs is a delaying action. The choices made today will doubtless have a profound impact on the future.

The specification of a task is not always trivial compared to the task itself. For some primitive operations, it is possible to allow some flexibility in the specification of a task. All operations entail some cost under some model and assumptions. It is vital that the expected cost not be altered significantly by changing only the specification. A long definition does not necessarily imply an expensive operation. For example, it may be more expensive to relabel all indices than extract a small block, though the definition of reax is much shorter than the definition of block. In a particular application, it might be desired to eliminate some flexibility which has been introduced if it is not needed as anyway such revisions do not change the operation fundamentally. Definitions may be specialized for particular applications.

Any flexibility which might be permissible could be added at the implementation level or in a particular application. These problems could be totally ignored. However, the theory should be balanced with practice and experience. Such details should not be left entirely to an implementation as an understanding of the theory is desirable to advantageously control software development. On the other hand, there are practical matters which are better suited to an implementation and for which the theory is irrelevant and one reason why the theory is independent of any implementation.

A valid approach might be to define operations disallowing any flexibility. At least it would be easier to define operations and prove that they are correct. It is beneficial if some guidelines are provided for an implementation that are appropriate for the data type and the given operation.

Arrays could be expressed via a small number of forms. A measure of success for any formalism is the number of possible forms. If after experimentation, it is discovered that there are too many forms, then it may be worthwhile to investigate other primitives before proceeding with development.

Are constructors useful? Besides the fact that useful operations can be defined in terms of the constructors, perhaps the most convincing evidence in this work is the lack of indices in programs. It should not be necessary to separate theory from pro-

gramming practice. The problems in software engineering are well-known. When a program is needed, it should not be necessary to look for theoretical results, translate the theory into some algorithm, rewrite in some language, prove it is all correct, and start all over when porting to a different system. Scientists still write in different languages and use different software. Naturally this causes communication problems and makes it difficult to compare and to check results. By taking the proposed approach to programming, it is possible for scientists to communicate more effectively. They can share not only the programs but also the proofs. While there are many challenges ahead, the potential benefits of using an algebraic theory should not be dismissed as there are important ramifications for both software development and research in the future.

REFERENCES

Aho, A.
1982. *Data Structures and Algorithms*. Reading, Massachusetts: Addison-Wesley.

Alexander, B., D. Englehardt, and A. Wendelborn
1995. An overview of the Adl language project. In *High Performance Functional Computing*, A. P. W. Bohm and J. T. Feo, eds., Pp. 73–82.

Arvind, R. Nikhil, and K. Pangali
1989. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):589–632.

Axford, T. and M. Joy
1993. List processing primitives for parallel computation. *Comput. Lang.*, 19(1):1–17.

Backhouse, R.
1989. An exploration of the Bird-Meertens Formalism. In *International Summer School on Constructive Algorithmics*, Ameland, The Netherlands.

Backhouse, R. and P. Hoogendijk
1993. Elements of a relational theory of datatypes. In *Formal Program Development*, B. Möller, H. Partsch, and S. Schuman, eds., volume 755 of *Lecture Notes in Computer Science*, Pp. 7–42. IFIP TC2/WG 2.1, Springer-Verlag. State of the Art Seminar, Rio de Janeiro, Jan. 1992.

Backhouse, R. C., P. J. de Bruin, P. Hoogendijk, G. Malcolm, T. S. Voermans, and J. C. S. P. van der Woude
1991. Relational catamorpisms. In *Constructing Programs from Specifications*, B. Möller, ed., Pp. 287–318.

Backus, J.
1978. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641.

Backus, J.
1981. The algebra of functional programs: Function level reasoning, linear equations, and extended definitions. In *Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, Pp. 1–43. Springer.

Banger, C.
1992. Arrays with categorical type constructors. In *ATABLE'92*, Pp. 105–121.

Banger, C.
1994. *Construction of multidimensional arrays as Categorical Data Types*. PhD thesis, Queen's University, Kingston, Canada.

Banger, C. R. and D. B. Skillicorn
1992. Flat arrays as a categorical data type. unpublished.

Banger, C. R. and D. B. Skillicorn
1993. A foundation for theories of arrays. unpublished.

Barth, P. S., R. Nikhil, and Arvind
1991. M-Structures: Extending a Parallel, Non-Strict Functional Language with State. In *FPCA'91, Cambrige, Massachusetts*, volume 523 of *Lecture Notes in Computer Science*, Pp. 538–568. Springer-Verlag.

Bird, R.
1984. The promotion and accumulation strategies in transformational programming. *acm Transactions on Programming Languages and Systems*, 6(4):487–504.

Bird, R.
1986. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6(2):159–189.

Bird, R.
1988. Lectures on constructive functional programming. Technical Monograph PRG 69, Oxford University.

Bird, R.
1990. A calculus of functions for program derivation. In *Research Topics in Functional Programming*, D. A. Turner, ed., University of Texas at Austin Year of Programming Series, Pp. 287–307. New York, N.Y.: Addison-Wesley.

Bird, R. and O. de Moor
1997. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall.

Bird, R., O. de Moor, and P. Hoogendijk
1996. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28.

Bird, R. and L. Meertens
1987. Two exercises found in a book on algorithmics. In *Program Specification and Transformations*, L. Meertens, ed., Pp. 451–457. North Holland.

Bird, R. and L. Meertens
1998. Nested datatypes. In *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, J. Jeuring, ed., volume 1422 of *Lecture Notes in Computer Science*, Pp. 52–67.

Bird, R. and R. Paterson
1999a. de bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91.

Bird, R. and R. Paterson
1999b. Generalized folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222.

Bird, R. S.
1987. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, M. Broy, ed., volume 36 of *NATO ASI Series F*, Pp. 3–42. Springer-Verlag.

Bird, R. S.
1989a. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126.

Bird, R. S.
1989b. Lectures on constructive functional programming. In *Constructive Methods in Computing Science*, M. Broy, ed., volume 55 of *NATO ASI Series F*, Pp. 151–216. Springer-Verlag.

Blelloch, G.
1990. *Vector Models for Data-Parallel Computing*. Cambridge, Massachusetts: The MIT Press.

Bratvold, T.
1994. Parallelising a functional program using a list-homomorphism skeleton. In *Proceedings of PaSCo94*, H. Long, ed., Pp. 44–53, Linz, Austria. World Scientific Publishing Company.

Bratvold, T. A.
1992. Determining Useful Parallelism in Higher Order Functions. In *4th International Workshop on the Parallel Implementation of Functional Languages*, H. Kuchen and R. Loogen, eds., Aachener Informatik-Berichte Nr 92-19, Pp. 213–226, Aachen, Germany.

Burstall, R. and P. Landin
1969. Programs and their proofs: an algebraic appoach. In *Machine Intelligence*, volume 4, Pp. 17–43. American Elsevier.

Campbell, D. K. G.
1997. A survey of models of parallel computation. Technical Report YCS-278, University of York.

Chamberlain, B. L., S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby
1998. The case for high level programming in zpl. volume 5, Pp. 76–86.

Chamberlain, B. L., E. C. Lewis, C. Lin, and L. Snyder
1999. Regions: An abstraction for expressing array computation. In *1999 ACM SIGAPL/SIGPLAN International Conference on Array Programming Languages*.

Codd, E. F.
1970. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Cole, M.
1993. Parallel programming, list homomorphisms and the maximum segment sum problem. Technical Monograph CSR-25-93, The University of Edinburgh.

Cole, M.
1995. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203.

Coppersmith, D. and S. Winograd
1990. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280.

Davis, P. and R. Hersh
1981. *The Mathematical Experience*. Boston, Massachusetts: Birkhäuser.

Duff, Z. N. G. and P. Harrison
1996. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295.

Eriksson-Bique, S.
1998. Digit serial computations via homomorphisms. In *Proceedings of PART '98: The 5th Australasian Conference on Parallel and Real-Time Systems*, K. Hawick and H. James, eds., Pp. 389–400, University of Adelaide, Adelaide, Australia. Springer-Verlag Singapore Pte. Ltd.

Fokkinga, M. M.
1996. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6:1–32.

Goguen, J.
1977. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95.

Gorlatch, S.
1996a. Systematic efficient parallelization of scan and other list homomorphisms. In *Annual European Conference on Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, Pp. 401–408. Springer-Verlag. Lyon, France.

Gorlatch, S.
1996b. Systematic extraction and implementation of divide-and-conquer parallelism. In *Conference on Programming Languages: Implementation, Logics and Programs*, volume 1140 of *Lecture Notes in Computer Science*, Pp. 274–288. Springer-Verlag.

Gorlatch, S. and C. Lenguaer
1995. Parallelisation of divide-and-conquer in the Bird-Meertens Formalism. *Formal Aspects of Computing*, 7(6):663–682.

Hagino, T.
1987. *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh.

Hains, G. and L. M. R. Mullin
1991. An algebra of multidimensional arrays. Technical Report 782.

Harrison, P.
1992. Higher order approach to parallel algorithms. *The Computer J.*, 35(6):555–566.

Hinze, R.
1999. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, 3(2):159–180.

Hu, Z., H. Iwasaki, and M. Takeichi
1994. Catamorphism based transformation of functional programs. In *Technical Report of IEICE*, Pp. 49–56.

Hu, Z. and M. Takeichi
1999. Calculating an optimal homomorphic algorithm for bracket matching. *Parallel Processing Letters*, 9(1).

Hu, Z., M. Takeichi, and W. N. Chin
1998a. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, Pp. 316–328, San Diego, California, USA.

Hu, Z., M. Takeichi, and H. Iwasaki
1997. Formal derivation of efficient parallel programs by construction of list homomorphisms. 19(3):444–461.

Hu, Z., M. Takeichi, and H. Iwasaki
1998b. Towards polytypic parallel programming. Technical Report METR 98-09, Department of Information Engineering, University of Tokyo, Japan.

Hudak, P.
1986. Para-functional programming. *Computer*, 19(8):60–70.

Hughes, J.
1991. Functional programming with bananas, lenses, envelopes, and barbed wire. In *FPCA'91: Functional Programming Languages and Computer Architecture*, E. Meijer, M. M. Fokkinga, and R. Paterson, eds., volume 523 of *Lecture Notes in Computer Science*, Pp. 124–144. Springer-Verlag.

Hutton, G.
1999. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372.

Iverson, K.
1962. *A Programming Language*. New York: John Wiley and Sons.

Jain, A.
1989. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall.

JáJá, J.
1992. *An Introduction to Parallel Algorithms*. Reading, Massachusetts: Addison-Wesley Publishing Company.

Jansson, P.
2000. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden.

Jay, C., M. Cole, M. Sekanna, and P. Steckler
1997. A monadic calculus for parallel costing in a functional language of arrays. In *Euro-Par'97 Parallel Processing*, C. Lengauer, M. Griebl, and S. Gorlatch, eds., volume 1300 of *Lecture Notes in Computer Science*, Pp. 650–651. Springer.

Jay, C. B. and J. Cockett
1994. Shapely types and shape polymorphism. In *Programming Languages and Systems - ESOP'94*, D. Sannella, ed., volume 788 of *Lecture Notes in Computer Science*, Pp. 302–316. Springer-Verlag.

Jenkins, M., J. Glasgow, and C. McCrosky
1986. Programming styles in nial. *IEEE Software*, 3(1):46–55.

Kolesnikov, A. and E. Trishina
1995. A parallel algorithm for thinning of binary images. In *Optoelectronics, Instrumentation and Data Processing*, number 6, Pp. 7–13.

Kumar, K. and D. Skillicorn
1995. Data parallel geometric operations on lists. *Parallel Computing*, 21:447–459.

Lambeck, J.
1968. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161.

Lehmann, D. J. and M. B. Smyth
1981. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, 14:97–139.

Lin, C. and L. Snyder
1993. Zpl: An array sublanguage. In *Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds., Pp. 96–114.

Malcolm, G.
1989. Homomorphism and promotability. In *Mathematics of Program Construction 1989*, J. van de Snepscheut, ed., Lecture Notes in Computer Science 375, Pp. 335–347. Springer-Verlag.

Malcolm, G.
1990a. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen.

Malcolm, G.
1990b. Data structures and program transformation. *Science of Computer Programming*, 14:255–279.

Malcolm, G.
1997. Hidden algebra and systems of abstract machines. In *Proceedings New Models for Software Architecture (IMSA97)*. Information-technology Promotion Agency.

Manes, E. G. and M. A. Arbib
1986. *Algebraic approaches to Program Semantics*, Text and Monographs in Computer Science. Berlin: Springer-Verlag.

Meertens, L.
1986. Algorithmics - towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, J. de Bakker, M. Hazewinkel, and J. Lenstra, eds., volume 1 of *CWI Monographs*, Pp. 289–334. North Holland.

Meertens, L.
1996. Calculate polytypically! In *PLILP'96*, volume 1140 of *Lecture Notes in Computer Science*, Pp. 1–16. Springer-Verlag.

Meijer, E. and J. Jeuring
1995. Merging monads and folds for functional programming. In *Advanced Functional Programming, AFP'95*, volume 925 of *Lecture Notes in Computer Science*, Pp. 228–266. Springer-Verlag.

Miller, R.
1993. *A Constructive Theory of Multidimensional Arrays*. PhD thesis, Lady Margaret Hall, Oxford.

More, T.
1973. Axioms and theorems for a theory of arrays. Technical Report 2, IBM Journal of Research and Development.

More, T.
1979. The nested rectangular array as a model of data. In *APL 79, APL Quote Quad*, number 4, Pp. 55–73. ACM, Inc.

More, T.
1986a. Notes on the diagrams, logic, and operations of array theory. Technical Report G230-2137, IBM Cambridge Scientific Center Technical Report.

More, T.
1986b. On the development of array theory. Technical report, IBM Cambridge Scientific Center Technical Report.

Mou, Z.
1990a. Divacon: A parallel language for scientific computing based on divide and conquer. In *The 3rd Symposium on the Frontiers of Massively Parallel Computation*, J. JáJá, ed., Pp. 451–461, Los Almitos, CA. IEEE Computer Society.

Mou, Z.
1990b. *A Formal Model for Divide-and-Conquer and Its Parallel Realization*. PhD thesis, Yale University.

Mou, Z. and P. Hudak
1988. An algebraic model for divide-and-conquer algorithms and its parallelism. *Journal of Supercomputing*, 2(3):257–278.

Mullin, L. and M. Jenkins
1991. A comparison of array theory and a mathematics of arrays. In *Arrays, Functional Languages and Parallel Systems.*, Pp. 237–269. Kluwer Academic Publishers.

Mullin, L. M. R.
1988. *A Mathematics of Arrays*. Doctoral dissertation, Syracuse University, Syracuse, New York.

Noble, B. and J. Daniel
1992. *Applied Linear Algebra*, third edition. Englewood Cliffs, New Jersey: Prentice-Hall.

Okasaki, C.
1998. *Purely Functional Data Structures*. Cambridge, UK: Cambridge University Press.

Paige, R.
1983. Transformational programming — applications to algoritms and systems. In *Conference Record of the 10th Annual ACM Symposium on POPL*, Pp. 73–87.

Pfeifer, H. and H. Rueá
1999. Polytypic proof construction. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, Y. Brelot, G. Dowek, A. Hirschowitz, C. Paulin, and L. ThĊry, eds., volume 1690 of *Lecture Notes in Computer Science*, Pp. 55–72. Springer-Verlag.

Pierce, B.
1991. *Basic Category Theory for Computer Scientists*. Cambridge, MA: The MIT Press.

Reynolds, J.
1979. Reasoning about arrays. *Communications of the ACM*, 22(5):290–299.

Skillicorn, D.
. *Foundations of Parallel Programming*, Cambridge Series in Parallel Computation 6. Cambridge University Press.

Skillicorn, D.
1990. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51.

Skillicorn, D.
1993. The Bird-Meertens Formalism as a parallel model. In *Software for Parallel Computation*, J. Kowalik and L. Grandinetti, eds., volume 106 of *NATO ASI Series F*. NATO Workshop on Software for Parallel Computation, Springer-Verlag.

Skillicorn, D.
1994a. Categorical data types. In *Second Workshop on Abstract Models for Highly Parallel Computation*. Oxford University Press. Leeds, April 1993.

Skillicorn, D.
1994b. *Foundations of Parallel Programming*, Cambridge Series in Parallel Computation 6. Cambridge University Press.

Skillicorn, D.
1994c. Questions and answers about categorical data types. invited paper at Meeting on Bulk Data Types for Architecture Independence, Parallel Processing Specialist Group, British Columbia Society, University of London.

Skillicorn, D.
1995. Towards a higher level of abstraction in parallel programming. In *Proceedings of the Programming Models for Massively Parallel Computers (PMMP '95)*, Pp. 78–85. IEEE CS Press. Berlin, Germany, October 1995.

Skillicorn, D. and W. Cai
1994. Equational code generation: Implementing categorical data types for data parallelism. In *Proceedings of TENCON'94*, Singapore.

Skillicorn, D. and W. Cai
1995. A cost calculus for parallel functional programming. *J. Parallel and Distributed Computing*, 28(1):65–83.

Skillicorn, D. B.
1992. *Parallelism and the Bird-Meertens Formalism*. Kingston, Ontario: Department of Computing and Information Science, Queen's University.

Skillicorn, D. B. and D. Talia
1998. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169.

Smith, M. and G. Plotkin
1982. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783.

Spindler, K.
1994. *Abstract Algebra with Applications: Vector spaces and groups*, volume 1. New York: Marcel Dekker, Inc.

Spivey, M.
1989. A categorical approach to the theory of lists. In *Mathematics of Program Construction*, J. L. A. van de Snepscheut, ed., volume 375 of *Lecture Notes in Computer Science*, Pp. 399–408.

Trichina, E. and S. Eriksson-Bique
1998. Calculus of locality based computations for custom computing machines. In *Proceedings of the 8th International Parallel Computing Workshop (PCW '98)*, Y. Teo, M. Feng, T. Ling, and E. Loh, eds., Pp. 259–266, School of Computing, National University of Singapore. Fujitsu Computers (Singapore) Pte. Ltd.

Trichina, E. and S. Eriksson-Bique
1999. Parallel solution to parentheses and pattern matching via near homomorphisms. *Australian Computer Science Communications*, 21(1):205–216.

Tu, H.-C.
1986. Fac: A functional apl language. *IEEE Software*, 3(1):36–45.

Wadler, P.
1981. Applicative style programming, program transformation, and list operators. In *Proceedings ACM Conference on Functional Programming Languages and Computer Architecture*, Pp. 25–32. ACM. Portsmouth, New Hampshire, Oct. 18-22.

Walters, R.
1991. *Categories and Computer Science*. Cambridge University Press.

APPENDIX

This appendix contains rigorous proofs of statements that were made in the main text. The proofs are arranged in the same order that they appear in the text.

CLAIM Any $\mathtt{n}$-array may be transformed into a $\mathtt{n}$-array with unique elements.

PROOF Let $\mathtt{A} = [\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})]$ be any nonempty array. Employ the usual method to transform a multiset into a set. Put $\widehat{\mathtt{A}} = [\widehat{\mathtt{a}}\,|\,(\mathtt{t}:\mathtt{n})]$ where

$$\widehat{\mathtt{a}}(\mathtt{i}:\mathtt{n}) = ((\mathtt{i}:\mathtt{n}),\mathtt{a}(\mathtt{i}:\mathtt{n})).$$

Every element of $\widehat{\mathtt{A}}$ is unique as every element is associated with a unique label. It is not possible to distinguish between $\mathtt{A}$ and $\widehat{\mathtt{A}}$ by inspection of fundamental properties such as the number of elements, the valency, the shapes, or the length of any axis.

In addition, there is a one-to-one correspondence between subarrays of $\mathtt{A}$ and subarrays of $\widehat{\mathtt{A}}$. In particular, if $\mathtt{B} = [\mathtt{b}\,|\,(\mathtt{s}:\mathtt{n})]$ is a subarray of $\mathtt{A}$, then $\widehat{\mathtt{B}} = \left[\widehat{\mathtt{b}}\,|\,(\mathtt{s}:\mathtt{n})\right]$ is a subarray of $\widehat{\mathtt{A}}$ where $\widehat{\mathtt{b}}(\mathtt{i}:\mathtt{n}) = ((\mathtt{i}:\mathtt{n}),\mathtt{b}(\mathtt{i}:\mathtt{n}))$. Moreover, the process is reversible: if $\widehat{\mathtt{B}} = \left[\widehat{\mathtt{b}}\,|\,(\mathtt{s}:\mathtt{n})\right]$ is a subarray of $\widehat{\mathtt{A}}$ with $\widehat{\mathtt{b}}(\mathtt{i}:\mathtt{n}) = ((\mathtt{i}:\mathtt{n}),\mathtt{c}(\mathtt{i}:\mathtt{n}))$, then $\mathtt{B} = [\mathtt{c}\,|\,(\mathtt{s}:\mathtt{n})]$ is a subarray of $\mathtt{A}$.

There is a naturality condition determined by applying any function $\mathtt{f}:\alpha\to\beta$ to the elements as indicated by the following commuting diagram

$$\begin{array}{ccc} \mathtt{A} & \xrightarrow{*\mathtt{f}} & \mathtt{B} \\ \big\downarrow{\scriptstyle\widehat{\ }} & & \big\downarrow{\scriptstyle\widehat{\ }} \\ \widehat{\mathtt{A}} & \xrightarrow[\widehat{*\mathtt{f}}]{} & \widehat{\mathtt{B}} \end{array}$$

where $\widehat{\ }$ is a natural transformation of an array $\mathtt{B}$ into an array $\widehat{\mathtt{B}}$ with unique elements and

$$*\mathtt{f}[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})] = [\mathtt{b}\,|\,(\mathtt{t}:\mathtt{n})],$$
$$\mathtt{b}(\mathtt{i}:\mathtt{n}) = \mathtt{f}(\mathtt{a}(\mathtt{i}:\mathtt{n})),$$

$$\widehat{*f}[\widehat{a}\,|\,(t:n)] = \left[\widehat{b}\,|\,(t:n)\right],$$

and

$$\widehat{b}(i:n) = ((i:n), f(a(i:n))).$$

Hence, there is an equivalence between a multiarray and a `map` with a unique array and another `map`. ■

LEMMA The tau shift, a conditional n-tuple and delete are well-defined. In addition, move right is injective and for all $p \in S$

$$\between_k \circ \sqsupset_k p = \mathrm{id}_{\mathcal{F}_S^k}.$$

PROOF Let $k \in \mathbb{N}$, $p \in \mathbb{Z}$ and take $T \in \mathbb{Z}^n$. Write $T = (t:n)$. If $k \leq n$ then $t_k + p - 1 \in \mathbb{Z}$ and so define

$$\tau_k(p)T = (f:n)$$

where

$$f_p = \begin{cases} t_p & \text{if } p \neq k \\ t_k + p - 1 & \text{if } p = k \end{cases},$$

which is well-defined since it is a linear function of the t_i's. If $k > n$ then

$$\tau_k(p)T = T.$$

The component functions are well-defined since they are linear (identity functions) by inspection. It has been established that τ_k is well-defined.

If $m \in \mathbb{Z}$ and $T = (t:n)$ is a n-tuple for $\mathbb{Z}$ then

$$\begin{aligned}(T; t_k = m) &= (t:n; t_k = m) \\ &= \tau_k(m+1-t_k)(t:n)\end{aligned}$$

is a valid n-tuple for $\mathbb{Z}$ since τ_k is well-defined.

Suppose

$$\sqsupset_k p\,(s:m) = \sqsupset_k (q)(t:n).$$

If $k \in \iota n$, by definition

$$\sqsupset_k p\,(s:m) = (g:m+1)$$

where

$$g_i = \begin{cases} s_i & \text{if } i < k \\ p & \text{if } i = k \\ s_{i-1} & \text{if } i > k \end{cases}.$$

Similarly, if $k \in \iota n$, by definition

$$\sqsupset_k q\,(t:n) = (h:n+1)$$

where

$$h_i = \begin{cases} t_i & \text{if } i < k \\ q & \text{if } i = k \\ t_{i-1} & \text{if } i > k \end{cases}.$$

As

$$\sqsupset_k p\ (s : m) = \sqsupset_k q\ (t : n),$$

it follows that

$$(g : m+1) = (h : n+1).$$

Hence, $m+1 = n+1$ and $g_i = h_i$, which implies $m = n$ and $s_i = t_i$ whence

$$(s : n) = (t : n).$$

If $k > n$ then

$$\sqsupset_k p\ (s : m) = (g : m+1)$$

where

$$g_i = \begin{cases} s_i & \text{if } i \neq m+1 \\ p & \text{if } i = m+1 \end{cases}.$$

Similarly, for $k \in \iota n$ by definition if $k > n$ then

$$\sqsupset_k q\ (t : n) = (h : n+1)$$

where

$$g_i = \begin{cases} t_i & \text{if } i \neq n+1 \\ q & \text{if } i = n+1 \end{cases}.$$

As

$$\sqsupset_k p\ (s : m) = \sqsupset_k q\ (t : n)$$

it follows that

$$(g : m+1) = (h : n+1).$$

Thus, $m+1 = n+1$ and $g_i = h_i$, which implies $m = n$ and $s_i = t_i$; whence,

$$(s : n) = (t : n).$$

This proves that $\sqsupset_k$ is injective.

Let $(t : n) \in S^n$ with $n \geq 2$. If $k \in \iota n$ then

$$\between_k (t : n) = (\rho : n-1)$$

where

$$\rho_q = \begin{cases} t_q & \text{if } q < k \\ t_{q+1} & \text{if } q \geq k \end{cases},$$

which is a well-defined linear function. If $k > n$ then

$$\between_k T = T$$

is well-defined. This shows that $\wr_k$ is well-defined.

If $k \in \iota n$ then by definition

$$(\wr_k \circ \sqsupset_k p)\,(t:n) = \wr_k\,(t_1,\ldots,t_{k-1},p,t_k,t_{k+1},\ldots,t_n) = (t:n).$$

If $k > n$ then by definition

$$\begin{aligned}(\wr_k \circ \sqsupset_k p)\,(t:n) &= \wr_k\,(t:n)\\ &= (t:n).\end{aligned}$$

This verifies that for all $p \in S$,

$$\wr_k \circ \sqsupset_k p = \mathrm{id}_{\mathcal{T}_S^k}$$

and completes the proof. ■

CLAIM Two locations L_1 and L_2 are adjacent for k if and only if for some $T = (t:n)$,

$$L_1 = \tau_k(p)\,T, \quad L_2 = \tau_k(q)\,T \quad \text{and} \quad |p-q| = 1.$$

PROOF Suppose $L_1 = (i:n)$ and $L_2 = (j:n)$ are adjacent for k. Then

$$(j:n) = (i:n; i_k = j_k).$$

with $|i_k - j_k| = 1$. By TM,

$$T = (i:n; i_k = 1),$$

and

$$\begin{aligned}\tau_k(i_k)\,T &= (i:n),\\ \tau_k(j_k)\,T &= (j:n).\end{aligned}$$

This proves the necessity. For the sufficiency, suppose for some $T = (t:n)$,

$$\tau_k(p)\,T = L_1, \quad \tau_k(q)\,T = L_2 \quad \text{and} \quad |p-q| = 1.$$

Write

$$\begin{aligned}L_1 &= (i:n),\\ L_2 &= (j:n).\end{aligned}$$

Then

$$L_1 = \tau_k(p)T = (t : n; t_k = t_k + p - 1),$$
$$L_2 = \tau_k(q)T = (t : n; t_k = t_k + q - 1).$$

By inspection,

$$i_m = t_m = j_m$$

for all $m \neq k$ and

$$\begin{aligned} |i_k - j_k| &= |(t_k + p - 1) - (t_k + q - 1)| \\ &= |p - q| \\ &= 1. \end{aligned}$$

This completes the proof. ■

LEMMA The tau identities hold always.

PROOF If $T = (t : n)$ is any n-tuple for $\mathbb{Z}$ then

$$\begin{aligned} \tau_k(1)T &= (t : n; t_k = t_k + 1 - 1) \\ &= (t : n; t_k = t_k) \\ &= T, \end{aligned}$$

which shows $\tau_k(1) = id_{\mathbb{Z}^n}$. In particular,

$$a \circ \tau_k(1)(i : n) = a(i : n).$$

This proves $[a \mid \tau_k(1)T] = [a \mid T] = [a \circ \tau_k(1) \mid T]$.

If $k \in \iota n$, then $\tau_k(p) \circ \tau_k(q)(t : n)$
$= \tau_k(p)(t : n; t_k = t_k + q - 1)$ by definition of τ
$= (t : n; t_k = t_k + q - 1 + p - 1)$ by definition of τ
$= (t : n; t_k = t_k + (p + q - 1) - 1)$ by commutativity
$= \tau_k(p + q - 1)(t : n)$ by definition of τ.

If $k > n$, then

$$\begin{aligned} \tau_k(p) \circ \tau_k(q)(t : n) &= \tau_k(p)(t : n) \text{ by definition of } \tau \\ &= (t : n) \text{ by definition of } \tau \\ &= \tau_k(p + q - 1)(t : n) \text{ by definition of } \tau. \end{aligned}$$

This proves $\tau_k(p)\circ\tau_k(q)=\tau_k(p+q-1)$. Calculate

$$\tau_k(q)\circ\tau_k(2-q)=\tau_k(2-q)\circ\tau_k(q) \text{ by TIP.}$$

Hence, $(\tau_k(q))^{-1}=\tau_k(2-q)$. Consequently,

$$\tau_k(p)\circ(\tau_k(q))^{-1}=(\tau_k(p)\circ\tau_k(2-q))=\tau_k(p+1-q) \text{ by TIP.}$$

If $k\in\iota\, n+1$ then

$$\begin{aligned}\tau_k(p)\circ\sqsupset_k(1)(t:n) &= \tau_k(p)(t_1,\ldots,t_{k-1},1,t_k,\ldots,t_n)\\ &=(t_1,\ldots,t_{k-1},p,t_k,\ldots,t_n)\\ &=\sqsupset_k(p)(t:n).\end{aligned}$$

Lastly,

$$\tau(n)1=1+n-1=n.$$

Observe that $\tau_k(p)$ always shifts the kth component whenever it is 1 to p.■

LEMMA For any n-array $[a\,|\,(t:n)]$ and for any m with $1\le m<t_k$,

$$[a\,|\,(t:n)] = [a\,|\,\tau_k(m)T]\dashv\!\vdash_k[a\circ\tau_k(m+1)\,|\,\tau_k(t_k-m)T].$$

where $T=(t:n;t_k=1)$.

PROOF Choose m with $1\le m<t_k$. Then

$$\tau_k(m)T=(t:n;t_k=m) \quad\text{and}\quad \tau_k(t_k-m)T=(t:n;t_k=t_k-m) \text{ by TM.}$$

By definition,

$$\begin{aligned}&[a\,|\,\tau_k(m)T]\dashv\!\vdash_k[a\circ\tau_k(m+1)\,|\,\tau_k(t_k-m)T]\\ &=[d\,|\,(t:n;t_k=m+t_k-m)]\\ &=[d\,|\,(t:n)]\end{aligned}$$

where

$$d(i:n)=\begin{cases}a(i:n) & \text{if } 1\le i_k\le m\\ a\circ\tau_k(m+1)(i:n;i_k=i_k-m) & \text{if } m<i_k\le t_k\end{cases}.$$

Evidently,

$$\begin{aligned}a\circ\tau_k(m+1)(i:n;i_k=i_k-m) &= a(i:n;i_k-m+(m+1-1))\\ &=a(i:n),\end{aligned}$$

and $[d\,|\,(t:n)] = [a\,|\,(t:n)]$. ■

CLAIM The scalar function $(\cdot)_n$ is injective for every $n \in \mathbb{N}$.

PROOF Choose $n \in \mathbb{N}$. Suppose $(\cdot)_n(x) = (\cdot)_n(y)$. Write $(\cdot)_n(x) = \left[a\,|\,\mathbb{C}^n\right]$ with $a\left(\mathbb{C}^n\right) = x$ and $(\cdot)_n(y) = \left[b\,|\,\mathbb{C}^n\right]$ with $b\left(\mathbb{C}^n\right) = y$. By the definition of equality for arrays, $a\left(\mathbb{C}^n\right) = b\left(\mathbb{C}^n\right)$; whence $x = a\left(\mathbb{C}^n\right) = b\left(\mathbb{C}^n\right) = y$. ■

CLAIM The function reax_k is injective for every $k \in \mathbb{N}$.

PROOF Let $k \in \mathbb{N}$. Assume

(1) $$\mathrm{reax}_k A = \mathrm{reax}_k B.$$

Write $A = [a\,|\,(t:n)]$ as reax_k is defined only for n-arrays. By definition,

$$\mathrm{reax}_k\,[a\,|\,(t:n)] = [a \circ \wr_k \mid \sqsupset_k(1)(t:n)].$$

As $\mathrm{reax}_k B = \mathrm{reax}_k A$,

(2) $$\mathrm{reax}_k B = \left[\widetilde{b}\,|\,S\right]$$

with

$$S = \sqsupset_k(1)(t:n)$$

and

(3) $$\widetilde{b} \circ \sqsupset_k(1)(i:n) = a \circ \wr_k \circ \sqsupset_k(1)(i:n).$$

Write $B = [b\,|\,(s:m)]$. By definition,

(4) $$\mathrm{reax}_k\,[b\,|\,(s:m)] = [b \circ \wr_k \mid \sqsupset_k(1)(s:m)]$$

In view of equations (2) and (4),

(5) $$\sqsupset_k(1)(s:m) = \sqsupset_k(1)(t:n) \qquad \text{and} \qquad \widetilde{b}(q) = b \circ \wr_k(q)$$

for all valid locations q. As $\sqsupset_k$ is injective,

$$m = n \qquad \text{and} \qquad s_q = t_q.$$

Now $B = [b \mid (t : n)]$ and

$$\begin{aligned} b(i : n) &= b \circ \langle\!\rangle_k \circ \sqsupset_k (1)(i : n) \text{ by equation (4)} \\ &= \widetilde{b} \circ \sqsupset_k (1)(i : n) \text{ by equation (5)} \\ &= a \circ \langle\!\rangle_k \circ \sqsupset_k (1)(i : n) \text{ by equation (3)} \\ &= a(i : n) \text{ by DR}. \end{aligned}$$

These equations show that $B = A$, which completes the proof. ■

CLAIM Concatenation is well-defined.

PROOF Suppose $A, B \in \mathcal{A}^*$, choose $k \in \mathbb{N}$ and consider $A +\!\!+_k B$. If $A = \square$ and $B \neq \square$, then

$$A +\!\!+_k B = B.$$

If $A \neq \square$ and $B = \square$, then

$$A +\!\!+_k B = A.$$

If $A = \square$ and $B = \square$ then the result is either A or B. Since $A = B$, in each case, the result is unique. Assume $A \neq \square$ and $B \neq \square$.

Write $A = [a \mid (t : n)]$. Now by inspection of the definition, concatenation is defined only if B can be expressed as $B = [b \mid (t : n; t_k = m)]$ for some $m \in \mathbb{N}$. By definition,

$$A +\!\!+_k B = [c \mid (t : n; t_k = t_k + m)],$$

where

$$c(i : n) = \begin{cases} a(i : n) & \text{if } 1 \leq i_k \leq t_k \\ b(i : n; i_k = i_k - t_k) & \text{if } t_k < i_k \leq t_k + m \end{cases}.$$

Evidently, $(t : n; t_k = t_k + m)$ is a valid n-tuple since $t_k + m \in \mathbb{N}$ and $t_q \in \mathbb{N}$ for $q \neq k$ by assumption. Let $(i : n) \in \mathbb{P}(t : n; t_k = t_k + m)$. If $i_k \leq t_k$ then by inspection,

$$(i : n) \in \mathbb{P}(t : n)$$

whence $c(i : n) = a(i : n)$ is well-defined. If $i_k > t_k$ then by inspection

$$(i : n) \in \mathbb{P}(t : n; t_k = t_k + m) \Rightarrow t_k < i_k \leq t_k + m,$$

whence

$$0 = t_k - t_k < i_k - t_k \leq t_k + m - t_k = m,$$

which implies

$$1 \leq i_k - t_k \leq m.$$

By inspection, $c(i : n) = b(i : n)$ is well-defined. ■

CLAIM The function size is well-defined.

PROOF Let $A \in \mathcal{A}^*$. If $A = \square$ then

$$\text{size } A = \text{size } \square = 0$$

is constant. Assume $A \neq \square$. Write $A = [a \mid (t : n)]$. Then

$$\text{size } A = \text{size } [a \mid (t : n)] = \prod_{i=1}^{n} t_i,$$

which is known to be well-defined. ■

CLAIM The function shape is well-defined.

PROOF Let $A \in \mathcal{A}^*$. If $A = \square$ then

$$\text{shape } A = \square$$

is constant. Assume $A \neq \square$. Write $A = [a \mid (t : n)]$. Then,

$$\text{shape } A = [t : n],$$

which is well-defined since t is a sequence and n is a 1-tuple. ■

CLAIM The function length is well-defined.

PROOF Let $k \in \mathbb{N}$. Let $A \in \mathcal{A}^*$. If $A = \square$ then

$$\text{length}_k \ A = \text{length}_k \ \square = 0$$

is constant. Assume $A \neq \square$. Write $A = [a \mid (t : n)]$. If $k > n$, then

$$\text{length}_k \ A = 0$$

is constant; otherwise,

$$\text{length}_k \ A = \text{length}_k \ [a \mid (t : n)] = t_k,$$

which is well-defined since it is a linear combination of the t_i's. ■

CLAIM The function axes is well-defined.

PROOF Let $\mathtt{A} \in \mathcal{A}^*$. If $\mathtt{A} = \square$ then

$$\text{axes } \mathtt{A} = \text{axes } \square = 0$$

is constant. Assume $\mathtt{A} \neq \square$. Write $\mathtt{A} = [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})]$. By definition

$$\text{axes } \mathtt{A} = \text{axes } [\mathtt{a} \mid (\mathtt{t} : \mathtt{n})] = \mathtt{n},$$

which is well-defined since the number of components in a $\mathtt{n}$-tuple is well-defined. ■

CLAIM The function selector fn is well-defined.

PROOF Let $\mathtt{A} = [\mathtt{a} \mid \mathtt{T}]$. Let $(\mathtt{i} : \mathtt{n}) \in \mathbb{P}(\mathtt{T})$. Then

$$\begin{aligned} (\text{fn } \mathtt{A})(\mathtt{i} : \mathtt{n}) &= \text{fn}[\mathtt{a} \mid \mathtt{T}](\mathtt{i} : \mathtt{n}) \\ &= \mathtt{a}|_{\mathbb{P}(\mathtt{T})}(\mathtt{i} : \mathtt{n}) \\ &= \mathtt{a}(\mathtt{i} : \mathtt{n}), \end{aligned}$$

which is well-defined since $[\mathtt{a} \mid \mathtt{T}]$ is a $\mathtt{n}$-array. ■

LEMMA The function collapse is well-defined. Furthermore, if $\mathtt{A}$ is any hyperplane for the $\mathtt{k}$-axis, then

$$\text{reax}_{\mathtt{k}} \circ \text{collapse}_{\mathtt{k}}\ \mathtt{A} = \mathtt{A}.$$

If $\mathtt{A}$ is any $\mathtt{n}$-array and $\mathtt{k} \in \iota(\mathtt{n}+1)$, then

$$\text{collapse}_{\mathtt{k}} \circ \text{reax}_{\mathtt{k}}\, \mathtt{A} = \mathtt{A}.$$

PROOF Let $\mathtt{k} \in \mathbb{N}$ and let $\mathtt{A} \in \mathbb{H}$. Using Definition 3.20, write

$$\mathtt{A} = [\mathtt{a} \mid \sqsupset_{\mathtt{k}}(1)(\mathtt{t} : \mathtt{n})].$$

Then

$$\text{collapse}_{\mathtt{k}}\ \mathtt{A} = [\mathtt{a} \circ \sqsupset_{\mathtt{k}}(1) \mid (\mathtt{t} : \mathtt{n})]. \tag{6}$$

Evidently the result is well-defined since $\sqsupset_{\mathtt{k}}$ is well-defined and $\mathtt{a}$ is an array function which is defined on $\mathbb{P}(\sqsupset_{\mathtt{k}}(1)(\mathtt{t} : \mathtt{n}))$.

In addition,

$$\begin{aligned} \text{reax}_{\mathtt{k}} \circ \text{collapse}_{\mathtt{k}}\ \mathtt{A} &= \text{reax}_{\mathtt{k}}[\mathtt{a} \circ \sqsupset_{\mathtt{k}}(1) \mid (\mathtt{t} : \mathtt{n})] \\ &= [\mathtt{a} \circ \sqsupset_{\mathtt{k}}(1) \circ \wr_{\mathtt{k}} \mid \sqsupset_{\mathtt{k}}(1)(\mathtt{t} : \mathtt{n})] \end{aligned}$$

where

$$\begin{aligned}&(\mathtt{a}\circ\beth_{\mathtt{k}}(1)\circ\gimel_{\mathtt{k}})\circ\beth_{\mathtt{k}}(1)(\mathtt{i}:\mathtt{n})\\&=\mathtt{a}\circ\beth_{\mathtt{k}}(1)\circ(\gimel_{\mathtt{k}}\circ\beth_{\mathtt{k}}(1))(\mathtt{i}:\mathtt{n})\ \text{since}\ \circ\ \text{is associative}\\&=\mathtt{a}\circ\beth_{\mathtt{k}}(1)(\mathtt{i}:\mathtt{n})\ \text{by DR}\end{aligned}$$

whence $\mathrm{reax}_{\mathtt{k}}\circ\mathrm{collapse}_{\mathtt{k}}\,\mathtt{A}=\mathtt{A}$.

Let $\mathtt{A}$ be any $\mathtt{n}$-array. If $\mathtt{k}\in\iota\,(\mathtt{n}+1)$, then by Definition 3.12,

$$\mathrm{reax}_{\mathtt{k}}\,[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})]=[\mathtt{a}\circ\gimel_{\mathtt{k}}\mid\beth_{\mathtt{k}}(1)(\mathtt{t}:\mathtt{n})]$$

By Definition 3.20,

$$\begin{aligned}\mathrm{collapse}_{\mathtt{k}}\,[\mathtt{a}\circ\gimel_{\mathtt{k}}\mid\beth_{\mathtt{k}}(1)(\mathtt{t}:\mathtt{n})]&=[\mathtt{a}\circ\gimel_{\mathtt{k}}\circ\beth_{\mathtt{k}}(1)\mid(\mathtt{t}:\mathtt{n})]\\&=[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})]\ \text{by DR}\end{aligned}$$

wherefore

$$\mathrm{collapse}_{\mathtt{k}}\circ\mathrm{reax}_{\mathtt{k}}\,\mathtt{A}=\mathtt{A},$$

which completes the proof. ■

CLAIM The function index is well-defined.

PROOF Suppose $\mathtt{A}\in\mathcal{A}_{\mathtt{Y}}$. Write $\mathtt{A}=[\mathtt{a}\,|\,(\mathtt{t}:\mathtt{n})]$. Let $[\mathtt{i}:\mathtt{m}]\in\mathbb{A}_{\mathbb{N}}$. If $\mathtt{m}\neq\mathtt{n}$ then

$$(\mathtt{i}:\mathtt{m})\notin\mathbb{P}(\mathtt{t}:\mathtt{n})$$

since $(\mathtt{t}:\mathtt{n})\in\mathbb{N}^{\mathtt{n}}$ has precisely $\mathtt{n}$ components. In this case,

$$\text{index}\ \mathtt{A}\ [\mathtt{i}:\mathtt{m}]$$

is not defined. Assume $\mathtt{m}=\mathtt{n}$. If

$$(\mathtt{i}:\mathtt{n})\notin\mathbb{P}(\mathtt{t}:\mathtt{n})$$

then again index $\mathtt{A}[\mathtt{i}:\mathtt{n}]$ is not defined. If $(\mathtt{i}:\mathtt{n})\in\mathbb{P}(\mathtt{T})$ then

$$\text{index}\ \mathtt{A}[\mathtt{i}:\mathtt{n}]=\mathtt{a}(\mathtt{i}:\mathtt{n}),$$

which is well-defined since $\mathtt{A}\in\mathcal{A}_{\mathtt{Y}}$. ■

CLAIM The function ax is well-defined.

PROOF Let $k \in \mathbb{N}$. Suppose $A = [a \mid (t : n)] \in \mathcal{A}$. By definition

$$\mathrm{ax}_k\ [a \mid (t : n)]\ [\ell : n] = [a \circ \boxminus : t_k],$$

The destructor ax is defined only if $k \in \iota\, n$. Plainly, $\ell_k = 1$ and

$$\ell_q \leq \begin{cases} t_q & \text{if } 1 \leq q < k \\ t_{q+1} & \text{if } k < q < n \end{cases}.$$

The result $[a \circ \boxminus : t_k] \in \mathbb{A}$ provided $a \circ \boxminus$ is defined on $\mathbb{P}(t_k)$ since $t_k \in \mathbb{N}$. If $1 \leq j \leq t_k$ then

$$\begin{aligned} \boxminus(j) &= \tau_k(j)(\ell : n) \\ &= (\ell_1, \ldots \ell_{k-1}, j, \ell_{k+1}, \ldots, \ell_n) \end{aligned}$$

is a valid location for a by inspection, completing the proof. ■

CLAIM The function block is well-defined.

PROOF Let $S \in \mathbb{A}_{\mathbb{Z}}^*$ and $U \in \mathbb{A}_{\mathbb{N}}^*$ and let A be any array. If $A = \square$ then

$$\text{block } S\ U\ A = \text{block } S\ U\ \square = \square$$

is constant. Assume $A \neq \square$. Write $A = [a \mid (t : n)]$. By Definition 3.23,

$$\text{block } S\ U\ A = C$$

If $S \neq \square$, then write $S = [s : m_s]$, which is possible since $S \in \mathbb{A}_{\mathbb{Z}}$. Set

$$\sigma_i = s_i \text{ for } 1 \leq i \leq m_s.$$

If $S = \square$, then put $m_s = 0$. For $m_s < k \leq n$, let $\sigma_k = 0$. Calculate

$$s'_k = \begin{cases} \sigma_k & \text{if } \sigma_k \geq 0 \\ t_k + \sigma_k + 1 & \text{if } -(t_k + 1) \leq \sigma_k < 0 \\ t_k & \text{if } \sigma_k < -(t_k + 1) \end{cases}.$$

By inspection s'_k is defined for all possible values of σ_k. Furthermore, $s'_k \geq 0$ (in the last case note $t_k \in \mathbb{N}$). For $k \in \iota\, n$,

$$0 \leq \hat{s}_k = \min\{s'_k, t_k\} \leq t_k.$$

If $\hat{s}_k = t_k$ for any k then the result is

$$C = \square$$

is constant. Assume $\hat{s}_k < t_k$ for all k since $\hat{s}_k \leq t_k$. If $U \neq \square$, then write

$U = (m_u, u)$ and set

$$\mu_i = u_i \text{ for } 1 \leq i \leq m_u;$$

otherwise, set $m_u = 0$. For $m_u < k \leq n$, set $\mu_k = t_k$. Then compute

$$\hat{u}_k = \min\{\mu_k, t_k - \hat{s}_k\},\ k \in \iota\, n.$$

If $\hat{u}_k \leq 0$ for any k then $C = \square$; otherwise,

$$C = [c\,|\,(\hat{u} : n)],$$

where

$$c(i : n) = a\left(\gamma^{i,\hat{s}} : n\right)$$

and

$$\gamma^{i,\hat{s}} = +\circ(i, \hat{s}).$$

By construction,

$$1 \leq \hat{u}_q = \min\left\{\mu_q, t_q - \hat{s}_q\right\} \leq t_q - \hat{s}_q.$$

By the previous work,

$$0 \leq \hat{s}_q < t_q$$

and this implies

$$t_q - \hat{s}_q \leq t_q.$$

Since

$$1 \leq \hat{u}_q$$

$(\hat{u} : n)$ is a valid n-tuple Let $(i : n) \in \mathbb{P}(\hat{u} : n)$. By inspection,

$$1 + 0 \leq i_q + \hat{s}_q \leq \hat{u}_q + \hat{s}_q \leq (t_q - \hat{s}_q) + \hat{s}_q = t_q$$

which proves $1 \leq i_q + \hat{s}_q \leq t_q$; consequently,

$$\left(\gamma^{i,\hat{s}} : n\right) = (i_1 + \hat{s}_1, \ldots, i_n + \hat{s}_n)$$

is valid and $a\left(\gamma^{i,\hat{s}} : n\right)$ is well-defined, which completes the proof. ■

CLAIM The function slice is well-defined.

PROOF Choose $k \in \mathbb{N}$ and $p \in \mathbb{Z}$. Let $A \in \mathcal{A}$. If $A = \square$ then

$$\text{slice}_k\ p\ \square = \square$$

is constant. Assume $A \neq \square$. Write $A = [a\,|\,(t : n)]$. If $k \notin \iota\, n$ or $|p| \notin \iota\, t_k$, then by definition

$$\text{slice}_k\ p\ A = \square$$

is constant; otherwise,

$$\mathrm{slice}_k\ p\ A = [\hat{a}\,|\,(t : n; t_k = 1)]$$

where

$$q = \begin{cases} p & \text{if } 1 \le p \le t_k \\ t_k + p + 1 & \text{if } -t_k \le p \le -1 \end{cases}$$

and

$$\hat{a}(i : n; i_k = 1) = a(i : n; i_k = q).$$

Assume $k \in \iota\, n$ or $|p| \in \iota\, t_k$. By inspection $1 \le q \le t_k$. Consequently, q is a valid index. It follows at once that

$$\hat{a}(i : n; i_k = 1) = a(i : n; i_k = q)$$

is well-defined, which completes the proof. ■

CLAIM The function take is well-defined.

PROOF Let $A \in \mathcal{A}^*$, $B \in \mathbb{A}_{\mathbb{Z}}^*$ and $k \in \mathbb{N}$. Consider

$$C = \mathrm{take}_k\ B\ A.$$

If $A = \square$ or $B = \square$ then by Definition 3.25, $C = \square$ is constant. Assume $A \neq \square$ and $B \neq \square$. Write $B = [b : m]$ and $A = [a\,|\,(t : n)]$. If $k > n$ then by definition $C = \square$. Assume $k \le n$. By definition

$$S = \{i \in \mathbb{N}\,|\,b_j < b_{j+1} \text{ for } 1 \le j \le i < m\},$$

and

$$q = 1 + \max(S \cup \{0\}).$$

Evidently q is well-defined as $S \cup \{0\} \neq \varnothing$. Every nonempty finite set of totally ordered elements has a maximum element. In addition, by construction

$$b_1 < \cdots < b_q.$$

For $1 \le i \le q$ assign

$$\beta_i = \begin{cases} t_k + b_i + 1 & \text{if } -t_k \le b_1 \le -1 \\ b_i & \text{otherwise} \end{cases}.$$

If $-t_k \le b_1 \le -1$ then add $t_k + 1$ to each term in the chain of inequalities

$$b_1 < \cdots < b_q$$

to obtain

$$\beta_1 = t_k + b_1 + 1 < \cdots < \beta_q = t_k + b_q + 1.$$

If the inequalities $-t_k \le b_1 \le -1$ fail to hold then still

$$\beta_1 = b_1 < \cdots < \beta_q = b_q.$$

By definition

$$S' = \{i \mid 1 \le \beta_i \le t_k\}.$$

If $S' = \varnothing$ then $C = \square$. Assume $S' \neq \varnothing$. Put

$$s = \min(S'),$$
$$u = 1 - s + \max(S').$$

Now s and u are defined since every nonempty set of totally ordered elements has a minimum and a maximum. By construction of S', the necessary condition

$$1 \le \beta_i \le t_k \text{ for } \min(S') \le i \le \max(S')$$

is guaranteed, which verifies that the β_i's are valid indices.

By inspection,

$$\begin{aligned} u &= 1 - \min(S') + \max(S') \\ &= 1 + \max(S') - \min(S') \text{ by commutativity} \\ &\ge 1 + \min(S') - \min(S') \text{ since } \max(X) \ge \min(X) \\ &= 1 + 0 = 1, \\ u &= 1 + \max(S') - \min(S') \le 1 + t_k - 1 = t_k, \end{aligned}$$

which yields $1 \le u \le t_k$. Consequently, u is a valid index for the kth component. The fact that the β_i's are distinct

$$1 \le \beta_1 < \cdots < \beta_{\max(S')} \le t_k$$

implies $\max(S') \le t_k$. By definition,

$$C = [\hat{a} \mid (t : n; t_k = u)]$$

where

$$\hat{a}(i : n) = a(i : n; i_k = \beta_{i_k + s - 1}).$$

If

$$1 \le i_k \le u$$

then adding $s - 1$ to each term in this chain of inequalities, it is evident that

$$s \le i_k + s - 1 \le u + s - 1.$$

As

$$s = \min(S') \ge 1$$

by construction of S' and

$$u+s-1 \leq \max(S') \leq t_k$$

it follows that

$$1 \leq i_k+s-1 \leq t_k,$$

which shows that i_k+s-1 is a valid argument for β. In review,

$$1 \leq \beta_{i_k+s-1} \leq t_k,$$

which shows that

$$(i:n; i_k = \beta_{i_k+s-1})$$

is a valid index for a; consequently, $\hat{a}$ is well-defined, completing the proof. ■

CLAIM The function drop is well-defined.

PROOF Pick $k \in \mathbb{N}, A \in \mathcal{A}^*$ and $B \in \mathbb{A}_{\mathbb{Z}}^*$. Consider

$$C = \text{drop}_k\ B\,A.$$

If $A = \square$ or $B = \square$, then by Definition 3.26,

$$C = A \in \mathcal{A}^*;$$

whence, the result is uniquely determined if $A = \square$ or $B = \square$. Assume $A \neq \square$ and $B \neq \square$. Write $B = [b:m]$ and $A = [a\,|\,(t:n)]$. If $k > n$ then $C = A$. Assume $k \leq n$ and set

$$S = \{i \in \mathbb{N} \,|\, b_j < b_{j+1} \text{ for } 1 \leq j \leq i < m\},$$

$$q = 1 + \max(S \cup \{0\}).$$

Evidently q is well-defined as $S \cup \{0\} \neq \varnothing$. Every nonempty finite set of totally ordered elements has a maximum element. By construction,

$$b_1 < \cdots < b_q.$$

Using the definition, for $1 \leq i \leq q$ put

$$\beta_i = \begin{cases} t_k + b_i + 1 & \text{if } -t_k \leq b_1 \leq -1 \\ b_i & \text{otherwise} \end{cases}.$$

If

$$-t_k \leq b_1 \leq -1$$

then add t_k+1 to each term in this chain of inequalities

$$b_1 < \cdots < b_q$$

to obtain

$$\beta_1 = t_k + b_1 + 1 < \cdots < \beta_q = t_k + b_q + 1.$$

If the inequalities $-t_k \leq b_1 \leq -1$ fail to hold then again

$$\beta_1 = b_1 < \cdots < \beta_q = b_q.$$

Let

$$S' = \{i \mid 1 \leq \beta_i \leq t_k\}.$$

If $S' = \varnothing$ then $C = A$ and the result is uniquely determined in this case. Assume $S' \neq \square$. Let

$$u = \max(S') - \min(S') + 1.$$

Now u is defined since every nonempty set of totally ordered elements has a minimum and a maximum element. By construction

$$1 \leq \beta_i \leq t_k \text{ for } i \in \{\min(S'), \ldots, \max(S')\},$$

which verifies that the β_i's are valid indices. By inspection,

$$u = \max(S') - \min(S') + 1 \geq \min(S') - \min(S') + 1 = 1 + 0 = 1,$$
$$u = \max(S') - \min(S') + 1 \leq t_k - 1 + 1 = t_k,$$

which implies $1 \leq u \leq t_k$. The fact that the β_i's are distinct

$$1 \leq \beta_1 < \cdots < \beta_{\max(S')} \leq t_k$$

implies $\max(S') \leq t_k$. In addition, $\min(S') \geq 1$ by construction of S'. If $u = t_k$ then the result $C = \square$ is uniquely determined also in this case. Assume $u < t_k$. By definition,

$$C = [\hat{a} \mid (t : n; t_k = t_k - u)]$$

where

$$\hat{a}(i : n) = a(i : n; i_k = s_{i_k}),$$
$$s_0 = 0,$$

and

$$s_q = \min\{j \in \mathbb{N} \mid j > s_{q-1},\ j \notin \{\beta_i \mid \min(S') \leq i \leq \max(S')\}\}.$$

By definition of s,

$$0 \leq s_0 < s_{q-1} < s_q$$

which implies

$$s_q \text{ is strictly increasing, and } s_q \geq 1 \text{ for } q > 0.$$

Since $\beta_1 < \cdots < \beta_{\max(S')}$, the set

$$\{\beta_i \mid \min(S') \leq i \leq \max(S')\}$$

has precisely

$$\max(S') - \min(S') + 1 = u$$

elements. As (see earlier work)

$$1 \leq \beta_1 < \cdots < \beta_{\max(S')} \leq t_k$$

it follows there are $t_k - u$ distinct elements in

$$\iota\, t_k - \{\beta_i \mid \min(S') \leq i \leq \max(S')\}$$

which can be placed in strictly increasing order; whence, by construction the s_q's are exactly these elements. Therefore,

$$1 \leq s_q \leq t_k$$

and so s_q is valid index, which completes the proof. ■

CLAIM The abstractor combine is well-defined.

PROOF Any array $[a \mid T]$ has an array function a that is well-defined on the set $\mathbb{P}(T)$. Thus,

$$\bigoplus_{t \in \mathbb{P}(T)} a(t)$$

is well-defined for any suitable operator $\oplus$. ■

CLAIM The abstractor zip is well-defined.

PROOF Let $\oplus$ be a binary operator and $A, B \in \mathcal{A}^*$. If $A = \square$ then $\underset{\leftrightarrow}{\oplus}$ is defined only if $B = \square$ and then the result

$$A \underset{\leftrightarrow}{\oplus} B = \square$$

is uniquely determined in this case. Assume $A \neq \square$. Write $A = [a \mid T]$. Now $\underset{\leftrightarrow}{\oplus}$ is defined only if B admits the form $[b \mid T]$ and

$$[a \mid T] \underset{\leftrightarrow}{\oplus} [b \mid T] = [\oplus \circ (a, b) \mid T] \in \mathcal{A},$$

since evidently T is a valid n-tuple and for $t \in \mathbb{P}(T)$

$$\oplus \circ (a, b)(t) = a(t) \oplus b(t)$$

is well-defined since a, b and $\oplus$ are well-defined, which completes the proof. ■

CLAIM The abstractor `map` is well-defined.

PROOF Let $\mathtt{A} \in \mathcal{A}^*$. If $\mathtt{A} = \square$ then the result

$$*\mathtt{f}\,\mathtt{A} = \square$$

is constant. Assume $\mathtt{A} \neq \square$. Write $\mathtt{A} = [\mathtt{a} \mid \mathtt{T}]$. Invoke the definition to obtain

$$*\mathtt{f}[\mathtt{a} \mid \mathtt{T}] = [\mathtt{f} \circ \mathtt{a} \mid \mathtt{T}].$$

Evidently $\mathtt{T}$ is a valid tuple. Let $\mathtt{t} \in \mathbb{P}(\mathtt{T})$. Then

$$(\mathtt{f} \circ \mathtt{a})(\mathtt{t}) = \mathtt{f}(\mathtt{a}(\mathtt{t}))$$

is well-defined since $\mathtt{f}$ and $\mathtt{a}$ are well-defined, which completes the proof. ■

CLAIM The abstractor `bicomp` is well-defined.

PROOF Let $\mathtt{F}, \mathtt{A} \in \mathcal{A}^*$. If $\mathtt{F} = \square$ or $\mathtt{A} = \square$ then in order for $\multimap$ to be defined, $\mathtt{A} = \mathtt{F} = \square$. The result

$$\multimap\ \mathtt{F}\,\mathtt{A} = \square$$

is uniquely determined in this case. Assume $\mathtt{F} \neq \square$ and $\mathtt{A} \neq \square$. Write $\mathtt{F} = [\mathtt{f} \mid \mathtt{T}]$. For $\multimap$ to be defined, $\mathtt{A} = [\mathtt{a} \mid \mathtt{T}]$. Now

$$\multimap [\mathtt{f} \mid \mathtt{T}]\ [\mathtt{a} \mid \mathtt{T}] = [@\circ(\mathtt{f}, \mathtt{a}) \mid \mathtt{T}].$$

where

$$@\circ(\mathtt{f}, \mathtt{a})(\mathtt{t}) = @\circ(\mathtt{f}(\mathtt{t}), \mathtt{a}(\mathtt{t})) = \mathtt{f}(\mathtt{t})(\mathtt{a}(\mathtt{t})),$$

which is well-defined since $\mathtt{f}$ and $\mathtt{a}$ are well-defined. This completes the proof. ■

CLAIM The abstractor `triad` is well-defined.

PROOF Let $\ominus \in \mathcal{A}$. Write $\ominus = [\theta \mid \mathtt{T}]$. Take suitable $\mathtt{A}, \mathtt{B} \in \mathcal{A}^*$. If $\mathtt{A} = \square$ then for the operation to be defined $\mathtt{B} = \square$ and vice versa. In both cases, the result

$$\overleftrightarrow{\ominus}\,\mathtt{A}\,\mathtt{B} = \square$$

is uniquely determined. Assume $\mathtt{A} \neq \square$. For $\overleftrightarrow{\ominus}\,\mathtt{A}\,\mathtt{B}$ to be defined,

$$\mathtt{A} = [\mathtt{a} \mid \mathtt{T}] \quad \text{and} \quad \mathtt{B} = [\mathtt{b} \mid \mathtt{T}]$$

By definition

$$\overleftrightarrow{[\theta \mid \mathtt{T}]}\ [\mathtt{a} \mid \mathtt{T}]\ [\mathtt{b} \mid \mathtt{T}] = [\mathtt{tribi} \circ (\theta, \mathtt{a}, \mathtt{b}) \mid \mathtt{T}].$$

where

$$\mathtt{tribi}\circ(\theta,\mathtt{a},\mathtt{b})(\mathtt{t}) = \mathtt{tribi}\circ(\theta(\mathtt{t}),\mathtt{a}(\mathtt{t}),\mathtt{b}(\mathtt{t})) = \mathtt{a}(\mathtt{t})\,\theta_{\mathtt{t}}\,\mathtt{b}(\mathtt{t}),$$

which is well-defined on $\mathbb{P}(\mathtt{T})$ since $\theta_{\mathtt{t}}$, $\mathtt{a}(\mathtt{t})$ and $\mathtt{b}(\mathtt{t})$ are well-defined. ■

CLAIM The abstractor `biaxis` is well-defined.

PROOF Let $\mathtt{F}\in\mathcal{A}$, $\mathtt{k}\in\mathbb{N}$. Take suitable $\mathtt{A}\in\mathcal{A}^2$. Write $\mathtt{F}=[\mathtt{f}|\,(\mathtt{t}:\mathtt{n})]$. For $\circledast_{\mathtt{k}}\mathtt{F}\,\mathtt{A}$ to be defined, $\mathtt{A}$ must admit the form

$$\mathtt{A} = [\mathtt{x}|\;\sqsupset_{\mathtt{k}}(\mathtt{s})(\mathtt{t}:\mathtt{n})].$$

By definition

$$\circledast_{\mathtt{k}}[\mathtt{f}|\,(\mathtt{t}:\mathtt{n})]\,[\mathtt{x}|\;\sqsupset_{\mathtt{k}}(\mathtt{s})(\mathtt{t}:\mathtt{n})] = [\mathtt{u}|\,(\mathtt{t}:\mathtt{n})]$$

where

$$\mathtt{u}(\mathtt{i}:\mathtt{n}) = \mathtt{f}(\mathtt{i}:\mathtt{n})\left[\mathtt{b}^{(\mathtt{i}:\mathtt{n})}:\mathtt{s}\right]$$

and the 1-array $\left[\mathtt{b}^{(\mathtt{i}:\mathtt{n})}:\mathtt{s}\right]$ is an axis of $\mathtt{A}$ whose components are given by

$$\mathtt{b}^{(\mathtt{i}:\mathtt{n})}_{\mathtt{j}} = \mathtt{x}\circ\sqsupset_{\mathtt{k}}(\mathtt{j})(\mathtt{i}:\mathtt{n}). \tag{7}$$

By inspection it suffices to check that $\mathtt{u}$ is well-defined as the $\mathtt{n}$-tuple in the resulting array is a $\mathtt{n}$-tuple of a given array. By inspection, $\mathtt{u}$ is well-defined provided $\left[\mathtt{b}^{(\mathtt{i}:\mathtt{n})}:\mathtt{s}\right]$ is well-defined since $\mathtt{f}$ is assumed to be well-defined on $\mathbb{P}(\mathtt{t}:\mathtt{n})$. Now $\mathtt{s}\in\mathbb{N}$ since

$$\sqsupset_{\mathtt{k}}(\mathtt{s})(\mathtt{t}:\mathtt{n})$$

is a $\mathtt{n}$-tuple of a given array and by inspection of equation (7), $\mathtt{b}^{(\mathtt{i}:\mathtt{n})}_{\mathtt{j}}$ is well-defined since $\mathtt{x}$ and $\sqsupset_{\mathtt{k}}$ are well-defined, which completes the proof. ■

CLAIM The abstractors $\smile_{\mathtt{k}}$, $\frown_{\mathtt{k}}$ and $\asymp_{\mathtt{k}}$ are well-defined.

PROOF These functions are well-defined since if the folding operation is not directed, then the operation is required to be associative. ■

CLAIM The abstractor permute is well-defined.

PROOF Pick $\mathtt{m}\in\mathbb{N}$ and let $\mathtt{p}$ be a $\mathtt{m}$-permutation. Let $\mathtt{A}\in\mathcal{A}$. The operation

$$\circlearrowleft_{\mathtt{k}}\,\mathtt{p}\,\mathtt{A}$$

is defined only if A has the form

$$A = [a \mid (t : n; t_k = m)]$$

and $k \in \iota\, n$. By definition

$$\circlearrowleft_k\, p\, [a \mid (t : n; t_k = m)] = [b \mid (t : n; t_k = m)]$$

where

$$b(i : n) = a\left(i : n; i_k = p^{-1}(i_k)\right).$$

The tuple $(t : n; t_k = m)$ is evidently well-defined since it is a n-tuple for an array. The element $b(i : n)$ is well-defined since a is an array function and p is a m-permutation, which implies

$$p^{-1}(1), \ldots, p^{-1}(m)$$

is a m-permutation (as $p : \iota\, m \to \iota\, m$ is bijective). ■

CLAIM The abstractor border is well-defined.

PROOF Let $A \in \mathcal{A}^*$ and $B \in \mathbb{A}^*_{\mathbb{N}_0}$. If $B = \square$ or $A = \square$ then

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

is uniquely determined in this case. Assume $B \neq \square$ and $A \neq \square$. Write $A = [a \mid (t : n)]$ and $B = \left[\widehat{b} : \widehat{n}\right]$. By definition

$$\text{border}\ B\, A = \left[c \mid \left(\gamma^{t,b} : n\right)\right],$$

$$\gamma^{t,b} = +\circ(t, \times 2 \circ b),$$

$$b_k = \widehat{b}_k \text{ for } 1 \leq k \leq \widehat{n},$$

$$b_k = 0 \text{ for } \widehat{n} < k \leq n,$$

$$c(i : n) = a\left(\widehat{i} : n\right),$$

and

$$\widehat{i_k} = \min\{t_k, \max\{1, i_k - b_k\}\}.$$

By inspection

$$t_k + 2b_k \geq t_k \geq 1$$

since $b_k \in \mathbb{N}_0$ implies $b_k \geq 0$. Hence, $t_k + 2b_k \in \mathbb{N}$ is a valid index. Now

$$\max\{1, i_k - b_k\} \geq 1;$$

whence,

$$\widehat{i_k} = \min\{t_k, \max\{1, i_k - b_k\}\} \geq \min\{t_k, 1\} \geq 1$$

since $t_k \geq 1$. In addition,

$$\widehat{i_k} = \min\{t_k, \max\{1, i_k - b_k\}\} \leq t_k$$

since the minimum of two numbers cannot be smaller than either of them. Hence,

$$1 \leq \widehat{i_k} \leq t_k;$$

consequently, $\left(\widehat{i} : n\right)$ is a valid location and $c(i:n) = a\left(\widehat{i} : n\right)$ is well-defined, which completes the proof. ■

CLAIM The abstractor unborder is well-defined.

PROOF Let $A \in \mathcal{A}^*$ and $B \in \mathbb{A}^*_{\mathbb{N}_0}$. If $A = \square$ or $B = \square$ then

$$\text{unborder } B\, C = A$$

is uniquely determined in this case. Assume $B \neq \square$ and $A \neq \square$. Write $A = [a|(t:n)]$ and $B = \left[\widehat{b} : \widehat{n}\right]$. By definition,

$$\text{unborder}\, B\, A = \begin{cases} \square & \text{if } t_k - 2b_k < 1 \text{ for some } k \\ \left[c|\left(\gamma^{t,b} : n\right)\right] & \text{otherwise} \end{cases},$$

$$\gamma^{t,b} = -\circ(t, \times 2 \circ b),$$

$$b_k = \widehat{b}_k \text{ for } 1 \leq k \leq \widehat{n},$$

$$b_k = 0 \text{ for } \widehat{n} < k \leq n,$$

$$c(i:n) = a(+\circ(i,b):n).$$

If for some k

$$t_k - 2b_k < 1$$

then

$$\text{unborder } [b:n]\, A = \square.$$

Assume

$$1 \leq t_k - 2b_k$$

which confirms that $t_k - 2b_k$ is a valid index. By inspection

$$1 \leq i_k \leq t_k - 2b_k.$$

Add b_k to both sides of $i_k \leq t_k - 2b_k$ to obtain

$$1 \leq i_k + b_k \leq t_k - 2b_k + b_k = t_k - b_k \leq t_k$$

since $b_k \geq 0$ as $B \in \mathbb{A}^*_{\mathbb{N}_0}$. Hence, $(+\circ(i,b):n)$ is a valid location and so

$$c(i:n) = a(+\circ(i,b):n)$$

is well-defined, which completes the proof. ■

CLAIM The abstractor split is well-defined.

PROOF Let $A \in \mathcal{A}^*$, $p \in \mathbb{Z}$ and $k \in \mathbb{N}$. By definition

$$\text{split}_k\ p\,A = \square \qquad \text{if } A = \square \text{ or } A = [a\,|\,(t:n)] \text{ and } k \notin \iota\, n.$$

The result is uniquely determined in these cases. Assume $A = [a\,|\,(t:n)]$ and $k \in \iota\, n$. By definition,

$$\text{split}_k\ p\,A = \begin{cases} (\square, A) & \text{if } p \leq 0 \\ (A, \square) & \text{if } p \geq t_k \\ (B, C) & \text{otherwise} \end{cases}$$

where

$$A = [a\,|\,(t:n)],$$
$$B = [a\,|\,(t:n; t_k = p)],$$
$$C = [a \circ \tau_k(p+1)\,|\,(t:n; t_k = t_k - p)].$$

Evidently the result is uniquely determined if $p \leq 0$ or $p \geq t_k$. Assume

$$1 \leq p < t_k.$$

If

$$A = [a\,|\,(t:n)] = \left[\hat{a}\,|\,(\hat{t}:n)\right] = \hat{A}$$

then

$$\text{split}_k\ p\,\hat{A} = (\hat{B}, \hat{C}).$$

Hence, $B = \hat{B}$ and $C = \hat{C}$. It suffices to check that B and C are well-defined. By inspection, B is a well-defined subarray of A. The n-tuple for C is valid. If $1 \leq i_k \leq t_k - p$ then

$$a \circ \tau_k(p+1)(i:n) = a(i:n; i_k = i_k + p).$$

Evidently, $1 \leq i_k + p \leq t_k - p + p = t_k$ which shows $(i:n; i_k = i_k + p)$ is a valid location for a, which completes the proof. ■

CLAIM The abstractor cover is well-defined.

PROOF Let $A \in \mathcal{A}^*$. If $X = \square$, $P = \square$, or $A = \square$ then

$$\text{cover } B\ X\ P\ A = (\cdot)\,(\text{border } B\ A)$$

is well-defined since $(\cdot)$ and border are well-defined. Assume $P \neq \square$, $A \neq \square$ and $X \neq \square$, write $A = [a \mid (t : n)]$, $X = [x : m]$, put $x_0 = 0$ and

$$\delta = \max(\{0\} \cup \{w \mid x_{k-1} < x_k \leq n \text{ for all } k \text{ with } 1 \leq k \leq w \leq m\}).$$

Then δ is well-defined since every nonempty finite set has a maximum element. By construction,

$$1 \leq x_1 < x_2 < \cdots < x_\delta \leq n; \tag{8}$$

consequently, the x_i's are valid axes in order. If $\delta \neq 0$, then write $P = [p : \hat{m}]$, set $\delta = \min\{\hat{m}, \delta\}$. If $p_k = \square$ for some k with $1 \leq k \leq \delta$, then put $\delta = 0$.

The minimum of two numbers always exists in the case $p_k \neq \square$ and so δ is well-defined. Let

$$p_k = [h^k : l^k] \quad \text{and} \quad h_0^k = 0 \quad \text{for } 1 \leq k \leq \delta,$$

For $k \in \iota\,\delta$, set

$$S^k = \{i \in \mathbb{N} \mid h_{j-1}^k < h_j^k \leq t_{x_k} \text{ for all } j \text{ with } 1 \leq j \leq i \leq l^k\},$$

and

$$r_k = \max(S^k \cup \{0\}).$$

For $k \in \iota\,\delta$ with $h_{r_k}^k < t_{x_k}$ set

$$r_k = r_k + 1 \quad \text{and then } h_{r_k}^k = t_{x_k}.$$

Evidently r_k is well-defined since $S \cup \{0\} \neq \varnothing$ and every nonempty finite set of totally ordered elements has a maximum element. If

$$0 = \min_{1 \leq k \leq \delta} \{r_k\} \tag{9}$$

then set $\delta = 0$. By construction of S^k,

$$0 = h_0^k < h_1^k < h_2^k < \cdots < h_{r_k}^k \leq t_{x_k}, \tag{10}$$

which confirms the partition is valid for the x_k-axis. If $\delta = 0$ then

$$\text{cover } B\ X\ P\ A = (\cdot)\,(\text{border } B\ A)$$

which is well-defined. Now assume $\delta > 0$. Reviewing, as $\delta > 0$ in view of equation

(9),

(11) $$r_k \in \mathbb{N} \text{ for } 1 \leq k \leq \delta.$$

Set

$$q_k = 0 \text{ for all } k \in \iota\, n,$$

and then update

$$q_{x_k} = k \text{ for all } k \in \iota\, \delta.$$

This is possible by equation (8). As the x_i's consist of all the axes along which partitioning takes place, if $q_k = 0$ then the k-axis is not partitioned. If $q_k \neq 0$ then partition along the k-axis. Next show that the k-axis is actually used as this fact is not obvious. If

$$q_k = p \neq 0$$

then by construction of the q_i's,

$$q_{x_p} = p = q_k$$

and noticing the subscripts cannot be different since the q_i's are distinct,

$$x_p = k.$$

Substituting $p = q_k$,

$$x_{q_k} = k.$$

Thus, $q_k \neq 0$ corresponds to $x_{q_k} = k$, p_{q_k}, h^{q_k}, and r_{q_k} by construction. By definition,

$$\text{cover } B\, X\, P\, A = [c : (z : n)]$$

where

$$z_k = \begin{cases} 1 & \text{if } q_k = 0 \\ r_{q_k} & \text{if } q_k \neq 0 \end{cases}.$$

Now $n \in \mathbb{N}$ and $z_k \in \mathbb{N}$ since $r_{q_k} \in \mathbb{N}$ by equation (11).

If $B = \square$ then set $\hat{n} = 0$; otherwise, write $B = \left[\hat{b} : \hat{n}\right]$. Set

$$b_k = \begin{cases} \hat{b}_k & \text{if } 1 \leq k \leq \hat{n} \\ 0 & \text{if } \hat{n} < k \leq n \end{cases},$$

$$c(i : n) = \left[\alpha^{(i:n)} \mid \left(u^{(i:n)} : n\right)\right],$$

$$u_k^{(i:n)} = \begin{cases} t_k + 2b_k & \text{if } q_k = 0 \\ h_{i_k}^{q_k} - h_{i_k - 1}^{q_k} + 2b_k & \text{if } q_k \neq 0 \end{cases},$$

$$\alpha^{(i:n)}(j : n) = a\left(\gamma^{i,j} : n\right),$$

$$\gamma_k^{i,j} = \phi(i_k, j_k),$$

$$\phi(i_k, j_k) = \begin{cases} \min\{\max\{1, j_k - b_k\}, t_k\} & \text{if } q_k = 0 \\ \min\left\{\max\left\{1, j_k + h_{i_k-1}^{q_k} - b_k\right\}, t_k\right\} & \text{if } q_k \neq 0 \end{cases}.$$

To show that c is well-defined, establish that

$$u_k^{(i:n)}$$

and $\alpha^{(i:n)}(j:n)$ are well-defined. The valency n is correct as the blocks are subarrays of A. If $q_k = 0$ then the k-axis is not partitioned, and

$$u_k^{(i:n)} = t_k + 2b_k \geq t_k.$$

If $q_k \neq 0$ then the k-axis is partitioned, which corresponds to partition p_{q_k} and by definition the width of this cell is

$$\begin{aligned} u_k^{(i:n)} &= h_{i_k}^{q_k} - h_{i_k-1}^{q_k} + 2b_k \\ &\geq h_{i_k}^{q_k} - h_{i_k-1}^{q_k} \text{ since } b_k \geq 0 \\ &\geq 1 \text{ by construction of } s^{q_k}. \end{aligned}$$

Note h^{q_k} is defined by construction since by inspection $0 < q_k \leq \delta$ and h^k is defined for $1 \leq k \leq \delta$. In addition, $h_{i_k}^{q_k}$ and $h_{i_k-1}^{q_k}$ are defined in view of equation (10) since by inspection

$$1 \leq i_k \leq r_{q_k}$$

and $i_k - 1 \geq 0$. It follows that

$$u_k^{(i:n)}$$

is well-defined.

Now $\alpha^{(i:n)}(j:n)$ is well-defined provided

$$\left(\gamma^{i,j} : n\right) = (\phi(i_1, j_1), \ldots, \phi(i_n, j_n))$$

is a valid location for A. To prove $\left(\gamma^{i,j} : n\right)$ is a valid location it suffices to show that

$$\gamma_k^{i,j} = \phi(i_k, j_k)$$

is a valid index. (Although it is not strictly necessary to prove the function is well-defined, observe that $j_k + h_{i_k-1}^{q_k} - b_k$ is the correct translation as $+h_{i_k-1}^{q_k}$ shifts over to the correct cell and $-b_k$ compensates correctly for the added border.) By inspection for suitable ψ,

$$\begin{aligned} \phi(i_k, j_k) &= \min\{\max\{1, \psi\}, t_k\} \\ &\leq t_k \text{ since } \min\{\zeta, \gamma\} \leq \gamma \text{ always,} \end{aligned}$$

and

$$\begin{aligned}\phi(i_k, j_k) &= \min\{\max\{1, \psi\}, t_k\} \\ &\geq \min\{1, t_k\} \text{ since } \max\{\zeta, \gamma\} \geq \zeta \text{ always} \\ &\geq 1.\end{aligned}$$

Hence,

$$1 \leq \phi(i_k, j_k) \leq t_k$$

and this shows that $\phi(i_k, j_k)$ is a valid location which completes the proof. ■

CLAIM The abstractor untile is well-defined.

PROOF Let $A \in \mathcal{A}$. If $A = (\cdot)\,\square$ then

$$\text{untile } A = \square$$

is constant. Assume $A \neq (\cdot)\,\square$. Write

$$A = [a\,|\,(t:n)] \qquad \text{and} \qquad a(i:n) = \left[\alpha^{(i:n)}\,|\,\left(s^{(i:n)} : n\right)\right].$$

For $k = 1, \ldots, n$, $p = 1, \ldots, t_k$, put

$$h_0^k = 0 \qquad \text{and} \qquad h_p^k = \sum_{q=1}^{p} s_k^{\tau_k(q)\mathbb{C}^n}.$$

Now $s_k^{\tau_k(q)\mathbb{C}^n}$ is well-defined by inspection since

$$1 \leq q \leq p \leq t_k.$$

It follows at once that h_p^k is well-defined since $\sum_{q=1}^{p} f(q)$ is well-defined whenever f is well-defined. Now

$$\begin{aligned}h_p^k &= \sum_{q=1}^{p} s_k^{\tau_k(q)\mathbb{C}^n} = s_k^{\tau_k(p)\mathbb{C}^n} + \sum_{q=1}^{p-1} s_k^{\tau_k(q)\mathbb{C}^n} \\ &= s_k^{\tau_k(p)\mathbb{C}^n} + h_{p-1}^k. \qquad (12)\end{aligned}$$

For $k = 1, \ldots, n$, $p = 1, \ldots, t_k$,

$$\psi^k\left(j + h_{p-1}^k\right) = p \text{ for all } j \text{ with } 1 \leq j \leq s_k^{\tau_k(p)\mathbb{C}^n}. \qquad (13)$$

By inspection j is well-defined and

$$\begin{aligned} j + h_{p-1}^{k} &\le s_{k}^{\tau_k(p)\mathfrak{C}^n} + h_{p-1}^{k} \\ &= h_{p}^{k} \text{ by equation (12)} \end{aligned}$$

which implies ψ_{j}^{k} is well-defined for

$$j = 1, \ldots, h_{t_k}^{k} \tag{14}$$

as $1 \le p \le t_k$. For $k = 1, \ldots, n$, set

$$\sigma_k = h_{t_k}^{k}.$$

Evidently, σ_k is well-defined. Letting $\gamma_{k}^{i} = \psi_{i_k}^{k}$ and $\omega_{k}^{i,h,\gamma} = i_k - h_{\gamma_k^i}^{k}$ put

$$u(i:n) = \alpha^{(\gamma^{i}:n)}\left(\omega^{i,h,\gamma}:n\right).$$

For $1 \le k \le n$, if i_k is a valid index, i.e.,

$$1 \le i_k \le \sigma_k = h_{t_k}^{k},$$

then $\gamma_{k}^{i} = \psi_{i_k}^{k}$ is well-defined by equation (14). By equation (13)

$$\psi_{i_k}^{k} = p \tag{15}$$

provided

$$1 + h_{p-1}^{k} \le i_k \le h_{p}^{k}.$$

Subtracting h_{p-1}^{k} from each term in this chain of inequalities to obtain

$$1 \le i_k - h_{p-1}^{k} \le h_{p}^{k} - h_{p-1}^{k} = s_{k}^{\tau_k(p)\mathfrak{C}^n}$$

whence

$$1 \le \omega_{k}^{i,h,\gamma} \le s_{k}^{\tau_k(p)\mathfrak{C}^n}.$$

Therefore $\omega_{k}^{i,h,\gamma}$ is well-defined. As

$$1 \le \gamma_{k}^{i} = p \le t_k$$

by equation (15), it follows that

$$u(i:n) = \alpha^{(\gamma^{i}:n)}\left(\omega^{i,h,\gamma}:n\right)$$

is well-defined, which completes the proof. ■

THEOREM (LIST DECOMPOSITION) Any 1-array can always be expressed

$$[a:n] = [a \circ \tau(1):1] \# \cdots \# [a \circ \tau(n):1].$$

PROOF By TI,

$$[a:\tau(1)1] = [a \circ \tau(1):1].$$

Moreover,

$$\begin{aligned}[a:\tau(2)1] &= [a:\tau(1)1] \# [a \circ \tau(2):\tau(1)1] \text{ by CI} \\ &= [a \circ \tau(1):1] \# [a \circ \tau(2):1] \text{ by TI}\end{aligned}$$

Suppose

$$[a:\tau(n)1] = [a \circ \tau(1):1] \# \cdots \# [a \circ \tau(n):1]$$

for some $n \geq 2$. Then

$$\begin{aligned}&[a:\tau(n+1)1] \\ &= [a:\tau(n)1] \# [a \circ \tau(n+1):\tau(1)1] \text{ by CI} \\ &= [a:\tau(n)1] \# [a \circ \tau(n+1):1] \text{ by TI} \\ &= ([a \circ \tau(1):1] \# \cdots \# [a \circ \tau(n):1]) \# [a \circ \tau(n+1):1] \\ &\text{by inductive hypothesis} \\ &= [a \circ \tau(1):1] \# \cdots \# [a \circ \tau(n+1):1] \\ &\text{since } \# \text{ is associative.}\end{aligned}$$

By the principle of mathematical induction, (LC) holds. ■

THEOREM (HYPERPLANE DECOMPOSITION) For every n-array $[a \mid (t:n)]$ with $n > 1$ and every $k \in \iota n$,

$$[a \mid (t:n)] = [a \circ \tau_k(1) \mid T] \#_k \cdots \#_k [a \circ \tau_k(t_k) \mid T]$$

where using TM

$$T = (t:n; t_k = 1).$$

PROOF By TI,

$$[a \mid \tau_k(1)T] = [a \circ \tau_k(1) \mid T].$$

Furthermore,

$$\begin{aligned}[a \mid \tau_k(2)T] &= [a \mid \tau_k(1)T] \#_k [a \circ \tau_k(2) \mid \tau_k(1)T] \text{ by CI} \\ &= [a \circ \tau_k(1) \mid T] \#_k [a \circ \tau_k(2) \mid T] \text{ by TI.}\end{aligned}$$

Suppose

$$[a \mid \tau_k(s)\, T] = [a \circ \tau_k(1) \mid T] \#_k \cdots \#_k [a \circ \tau_k(s) \mid T]$$

whenever $s \leq p$ for some $p \geq 2$. Then $[a \mid \tau_k(s+1)\, T]$
$= [a \mid \tau_k(s)\, T] \#_k [a \circ \tau_k(s+1) \mid \tau_k(1)\, T]$ by CI
$= [a \mid \tau_k(s)\, T] \#_k [a \circ \tau_k(s+1) \mid T]$ by TI
$= ([a \circ \tau_k(1) \mid T] \#_k \cdots \#_k [a \circ \tau_k(s) \mid T]) \#_k [a \circ \tau_k(s+1) \mid T]$
by inductive hypothesis
$= [a \circ \tau_k(1) \mid T] \#_k \cdots \#_k [a \circ \tau_k(s+1) \mid T]$
since $\#_k$ is associative.
By the principle of mathematical induction, the statement of the theorem is true. ■

THEOREM A left hom, a right hom and a catamorphism are uniquely determined by the binary operator $\oplus$ and the unary function u.

PROOF Assume h and h' are left homs for an array A with binary operator $\oplus$ and unary function u. Let x be an element of A. Then

$$h(\cdot)(x) = u(x) = h'(\cdot)(x),$$

which establishes the basis for induction. Assume $h(B) = h'(B)$ whenever B, which is a subarray of A, is a singleton or a concatenation of a sufficiently small number of them. Let B be any subarray of A. By Theorem 4.9, B is a singleton or a concatenation of finitely many of them. Using Corollary 4.13, write

$$B = C \# (\cdot)(x).$$

As h and h' are left homs,

$$\begin{aligned} h(B) &= h(C \# (\cdot)(x)) \\ &= h(C) \oplus h(\cdot)(x) \\ &= h'(C) \oplus h'(\cdot)(x) \text{ by induction} \\ &= h'(C \# (\cdot)(x)) \\ &= h'(B). \end{aligned}$$

Since h and h' agree for concatenations of two singletons (C is a singleton), they must agree for concatenations of three singletons, and so forth. If $h(\square)$ and $h'(\square)$ are defined, then

$$h(\square) = h'(\square)$$

since both are required to be equal to the unique identity element for $\oplus$. This establishes uniqueness for left homs. A symmetrical argument holds for right homs, i.e.,

replace

$$C + (\cdot)(x)$$

by

$$(\cdot)(x) + C$$

in the preceding discussion and use equation (4.2) instead of equation (4.1).

Now assume h, h' are catamorphisms for an array A with binary operator $\oplus$ and unary function u. The previous work shows that h and h' agree on singletons. Let $B + C$ be a subarray of A with $B, C \neq \square$. Then

$$\begin{aligned} h(B + C) &= h(B) \oplus h(C) \\ &= h'(B) \oplus h'(C) \text{ by induction} \\ &= h'(B + C). \end{aligned}$$

By Theorem 4.9, B and C are singletons or concatenations of finitely many of them. Since h and h' agree on singletons, they must agree on a concatenation of any two singletons, and then on any concatenation involving up to four singletons and so forth. If catamorphisms h and h' are defined on an empty array then again $h(\square) = h'(\square)$, by uniqueness of identity elements. This establishes uniqueness for catamorphisms and completes the proof of the unique extension property for list homomorphisms. ■

THEOREM (PROMOTION LEMMA) Let h be any morphism such that

$$h(a \oplus b) = h(a) \otimes h(b).$$

Then

$$\begin{aligned} h \circ lom(\oplus, u) &= lom(\otimes, h \circ u), \\ h \circ rom(\oplus, u) &= rom(\otimes, h \circ u), \\ h \circ (\oplus, u) &= (\otimes, h \circ u). \end{aligned}$$

PROOF Assume $lom(\oplus, u)$ is defined for some array A, that

$$\begin{aligned} &h(lom(\oplus, u)(B + (\cdot)(x))), \\ &h(lom(\oplus, u)B), \\ &h(lom(\oplus, u)(\cdot)(x)), \end{aligned}$$

are defined whenever $B + (\cdot)(x)$ is a valid subarray of A, and that

$$h(lom(\oplus, u)(\cdot)(x))$$

is defined for all elements x of A. Let x be an element of A. Calculate

$$\begin{aligned}(h \circ \mathrm{lom}(\oplus, u))(\cdot)(x) &= h(\mathrm{lom}(\oplus, u)(\cdot)(x)) \\ &= h(u(x)) \\ &= (h \circ u)(x).\end{aligned}$$

If $B + (\cdot)(x)$ is a valid subarray of A then

$$\begin{aligned}&(h \circ \mathrm{lom}(\oplus, u))(B + (\cdot)(x)) \\ &= h(\mathrm{lom}(\oplus, u)(B + (\cdot)(x))) \\ &= h((\mathrm{lom}(\oplus, u)\,B) \oplus (\mathrm{lom}(\oplus, u)(\cdot)(x))) \\ &= h(\mathrm{lom}(\oplus, u)\,B) \otimes h(\mathrm{lom}(\oplus, u)(\cdot)(x)) \\ &= (h \circ \mathrm{lom}(\oplus, u))\,B \otimes (h \circ \mathrm{lom}(\oplus, u))(\cdot)(x).\end{aligned}$$

This shows that $h \circ \mathrm{lom}(\oplus, u)$ is a left hom. By uniqueness,

$$h \circ \mathrm{lom}(\oplus, u) = \mathrm{lom}(\otimes, h \circ u).$$

A symmetrical argument shows that

$$h \circ \mathrm{rom}(\oplus, u) = \mathrm{rom}(\otimes, h \circ u).$$

Next assume $(\!|\oplus, u|\!)$ is defined for some array A, that

$$h\,((\!|\oplus, u|\!)(B + C)), \quad h\,((\!|\oplus, u|\!)B) \quad \text{and} \quad h\,((\!|\oplus, u|\!)C),$$

are defined whenever $B + C$ is a valid subarray of A, and that

$$h\,((\!|\oplus, u|\!)(\cdot)(x))$$

is defined for all elements x of A. Let x be an element of A. Compute

$$\begin{aligned}(h \circ (\!|\oplus, u|\!))(\cdot)(x) &= h\,((\!|\oplus, u|\!)(\cdot)(x)) \\ &= h(u(x)) \\ &= (h \circ u)(x).\end{aligned}$$

If $B + C$ is a valid subarray of A then

$$\begin{aligned}(h \circ (\!|\oplus, u|\!))\,(B + C) &= h\,((\!|\oplus, u|\!)(B + C)) \\ &= h\,(((\!|\oplus, u|\!)B) \oplus ((\!|\oplus, u|\!)C)) \\ &= h\,((\!|\oplus, u|\!)B) \otimes h\,((\!|\oplus, u|\!)C) \\ &= (h \circ (\!|\oplus, u|\!))\,B \otimes (h \circ (\!|\oplus, u|\!))\,C.\end{aligned}$$

This shows that $h \circ (\oplus, u)$ is a catamorphism. Use uniqueness to obtain

$$h \circ (\oplus, u) = (\otimes, h \circ u),$$

which completes the proof. ■

THEOREM The following identities hold:

$$\begin{aligned}
lom(\oplus, u)[a : n] &= (\cdots((u(a_1) \oplus u(a_2)) \oplus u(a_3))\cdots) \oplus u(a_n), \\
rom(\oplus, u)[a : n] &= u(a_1) \oplus (u(a_2) \oplus (\cdots(u(a_{n-1}) \oplus u(a_n))\cdots)), \\
(\oplus, u)[a : n] &= u(a_1) \oplus u(a_2) \oplus \cdots \oplus u(a_n).
\end{aligned}$$

PROOF Let $lom(\oplus, u)$, $rom(\oplus, u)$, and $(\oplus, u)$ be homomorphisms for some array A. Consider any function h which is a left hom, right hom or catamorphism for A with unary function u. If $[a : \tau_k(1)\,1]$ is any subarray of A, then

$$\begin{aligned}
h[a : \tau_k(1)\,1] &= h[a : 1] \text{ by TI} \\
&= h(\cdot)(a_1) \text{ by SI} \\
&= u(a_1) \text{ by definition of } h.
\end{aligned}$$

If $[a : \tau_k(2)\,1]$ is any subarray of A, then

$$\begin{aligned}
& h[a : \tau_k(2)\,1] \\
&= h([a : \tau_k(1)\,1] +\!\!+_k [a \circ \tau_k(2) : \tau_k(1)\,1]) \text{ by CI} \\
&= h([a \circ \tau_k(1) : 1] +\!\!+_k [a \circ \tau_k(2) : 1]) \text{ by TI} \\
&= h((\cdot)(a_1) +\!\!+_k (\cdot)(a_2)) \text{ by SI} \\
&= u(a_1) \oplus u(a_2) \text{ by definition of } h.
\end{aligned}$$

The basis for induction is established (in all cases).

First, consider $h = lom(\oplus, u)$. Suppose

$$h[a : \tau(n)\,1] = (\cdots((u(a_1) \oplus u(a_2)) \oplus u(a_3))\cdots) \oplus u(a_n)$$

holds for some $n \geq 2$, whenever $[a : \tau(n)\,1]$ is a subarray of A. Let $[a : \tau(n+1)\,1]$ be any subarray of A. Then

$$\begin{aligned}
& h[a : \tau(n+1)\,1] \\
&= h([a : \tau(n)\,1] +\!\!+ [a \circ \tau(n+1) : \tau(1)\,1]) \text{ by CI} \\
&= h([a : \tau(n)\,1] +\!\!+ [a \circ \tau(n+1) : 1]) \text{ by TI} \\
&= h([a : \tau(n)\,1] +\!\!+ (\cdot)(a_{n+1})) \text{ by SI} \\
&= (h[a : \tau(n)\,1]) \oplus u(a_{n+1}) \text{ since } h = lom(\oplus, u) \\
&= ((\cdots((u(a_1) \oplus u(a_2)) \oplus u(a_3))\cdots) \oplus u(a_n)) \oplus u(a_{n+1}) \text{ c by inductive hypoth-}
\end{aligned}$$

esis.

This establishes the stated identity for $\mathtt{lom}(\oplus, \mathtt{u})$.

Second, consider $\mathtt{h} = \mathtt{rom}(\oplus, \mathtt{u})$. Suppose

$$\mathtt{h}\,[\mathtt{a} : \tau(\mathtt{n})\,1] = \mathtt{u}(\mathtt{a}_1) \oplus (\mathtt{u}(\mathtt{a}_2) \oplus (\cdots (\mathtt{u}(\mathtt{a}_{n-1}) \oplus \mathtt{u}(\mathtt{a}_n)) \cdots))$$

holds for some $\mathtt{n} \geq 2$ whenever $[\mathtt{a} : \tau(\mathtt{n})\,1]$ is a subarray of $\mathtt{A}$. If $[\mathtt{a} : \mathtt{n}+1]$ is any subarray of $\mathtt{A}$, then

$$\begin{aligned}
&\mathtt{h}\,[\mathtt{a} : \tau(\mathtt{n}+1)\,1] \\
&= \mathtt{h}([\mathtt{a} : \tau(1)\,1] +\!\!+\ [\mathtt{a} \circ \tau(2) : \tau(\mathtt{n})\,1]) \text{ by CI} \\
&= \mathtt{h}([\mathtt{a} : 1] +\!\!+\ [\mathtt{a} \circ \tau(2) : \tau(\mathtt{n})\,1]) \text{ by TI} \\
&= \mathtt{h}((\cdot)(\mathtt{a}_1) +\!\!+\ [\mathtt{a} \circ \tau(2) : \tau(\mathtt{n})\,1]) \text{ by SI} \\
&= \mathtt{u}(\mathtt{a}_1) \oplus \mathtt{h}\,[\mathtt{a} \circ \tau(2) : \mathtt{n}] \text{ since } \mathtt{h} = \mathtt{rom}(\oplus, \mathtt{u}) \\
&= \mathtt{u}(\mathtt{a}_1) \oplus \\
&\qquad (\mathtt{u}((\mathtt{a} \circ \tau(2))(1)) \oplus (\cdots (\mathtt{u}((\mathtt{a} \circ \tau(2))(\mathtt{n}-1)) \oplus \mathtt{u}((\mathtt{a} \circ \tau(2))(\mathtt{n}))) \cdots)) \\
&\qquad \text{by the inductive hypothesis} \\
&= \mathtt{u}(\mathtt{a}_1) \oplus (\mathtt{u}(\mathtt{a}_2) \oplus (\cdots (\mathtt{u}(\mathtt{a}_n) \oplus \mathtt{u}(\mathtt{a}_{n+1})) \cdots)) \text{ by TIP}
\end{aligned}$$

By the principle of mathematical induction, the stated identity for $\mathtt{rom}(\oplus, \mathtt{u})$ holds.

Third, consider $\mathtt{h} = (\!|\oplus, \mathtt{u}|\!)$. Suppose

$$(\!|\oplus, \mathtt{u}|\!)\,[\mathtt{a} : \tau(\mathtt{n})\,1] = \mathtt{u}(\mathtt{a}_1) \oplus \mathtt{u}(\mathtt{a}_2) \oplus \cdots \oplus \mathtt{u}(\mathtt{a}_n)$$

holds for some $\mathtt{n} \geq 2$ whenever $[\mathtt{a} : \tau(\mathtt{n})\,1]$ is a subarray of $\mathtt{A}$. If $[\mathtt{a} : \mathtt{n}+1]$ is any subarray of $\mathtt{A}$, then

$$\begin{aligned}
&\mathtt{h}\,[\mathtt{a} : \tau(\mathtt{n}+1)\,1] \\
&= \mathtt{h}([\mathtt{a} : \tau(\mathtt{m})\,1] +\!\!+\ [\mathtt{a} \circ \tau(\mathtt{m}+1) : \tau(\mathtt{n}+1-\mathtt{m})\,1]) \text{ by CI} \\
&= (\mathtt{h}\,[\mathtt{a} : \tau(\mathtt{m})\,1]) \oplus (\mathtt{h}\,[\mathtt{a} \circ \tau(\mathtt{m}+1) : \tau(\mathtt{n}+1-\mathtt{m})\,1]) \text{ since } \mathtt{h} = (\!|\oplus, \mathtt{u}|\!) \\
&= (\mathtt{u}(\mathtt{a}_1) \oplus \cdots \oplus \mathtt{u}(\mathtt{a}_\mathtt{m})) \oplus \\
&\qquad (\mathtt{u}((\mathtt{a} \circ \tau(\mathtt{m}+1))(1)) \oplus \cdots \oplus \mathtt{u}((\mathtt{a} \circ \tau(\mathtt{m}+1))(\mathtt{n}+1-\mathtt{m}))) \\
&\qquad\qquad \text{by the inductive hypothesis} \\
&= (\mathtt{u}(\mathtt{a}_1) \oplus \cdots \oplus \mathtt{u}(\mathtt{a}_\mathtt{m})) \oplus (\mathtt{u}(\mathtt{a}_{\mathtt{m}+1}) \oplus \cdots \oplus \mathtt{u}(\mathtt{a}_{n+1})) \text{ by TIP} \\
&= \mathtt{u}(\mathtt{a}_1) \oplus \cdots \oplus \mathtt{u}(\mathtt{a}_{n+1}) \text{ since } \mathtt{m} \text{ is arbitrary.}
\end{aligned}$$

By the principle of mathematical induction, the stated identity holds for $(\!|\oplus, \mathtt{u}|\!)$. ■

THEOREM The following identities hold:

$$\mathtt{lom}_k(\oplus,u)[a\,|\,\tau_k(t_k)\,T] = (\cdots(u[a\circ\tau_k(1)\,|\,T]\oplus u[a\circ\tau_k(2)\,|\,T])\cdots)\oplus u[a\circ\tau_k(t_k)\,|\,T],$$

$$\mathtt{rom}_k(\oplus,u)[a\,|\,\tau_k(t_k)\,T] = u[a\circ\tau_k(1)\,|\,T]\oplus(\cdots(u[a\circ\tau_k(t_k-1)\,|\,T]\oplus u[a\circ\tau_k(t_k)\,|\,T])\cdots),$$

and

$$(\oplus,u)_k[a\,|\,\tau_k(t_k)\,T] = u[a\circ\tau_k(1)\,|\,T]\oplus\cdots\oplus u[a\circ\tau_k(t_k)\,|\,T]$$

where using TM

$$T = (t:n;t_k=1).$$

PROOF Let $\mathtt{lom}_k(\oplus,u)$, $\mathtt{rom}_k(\oplus,u)$, and $(\oplus,u)_k$ be homomorphisms for some array A. Consider any function h which is a left hom, right hom or catamorphism for A with unary function u. If $[a\,|\,\tau_k(1)\,T]$ is any subarray of A, then

$$\begin{aligned}
h[a\,|\,\tau_k(1)\,T] &= h[a\,|\,T] \text{ by TI}\\
&= h\circ \mathrm{reax}_k(\mathrm{collapse}_k[a\,|\,T]) \text{ by RC}\\
&= u\circ \mathrm{reax}_k(\mathrm{collapse}_k[a\,|\,T]) \text{ by definition of } h\\
&= u[a\,|\,T] \text{ by RC}\\
&= u[a\circ\tau_k(1)\,|\,T] \text{ by TI.}
\end{aligned}$$

If $[a|\tau_k(2)\,T]$ is any subarray of A, then

$$\begin{aligned}
&h[a\,|\,\tau_k(2)\,T]\\
&= h([a\,|\,\tau_k(1)\,T] +\!\!+_k [a\circ\tau_k(2)\,|\,\tau_k(1)\,T]) \text{ by CI}\\
&= h([a\circ\tau_k(1)\,|\,T] +\!\!+_k [a\circ\tau_k(2)\,|\,T]) \text{ by TI}\\
&= h((\mathrm{reax}_k(\mathrm{collapse}_k[a\circ\tau_k(1)|T])) +\!\!+_k (\mathrm{reax}_k(\mathrm{collapse}_k[a\circ\tau_k(2)|T])))\\
&\quad \text{by RC}\\
&= u(\mathrm{reax}_k(\mathrm{collapse}_k[a\circ\tau_k(1)|T]))\oplus u(\mathrm{reax}_k(\mathrm{collapse}_k[a\circ\tau_k(2)\,|\,T]))\\
&\quad \text{by definition of } h\\
&= u[a\circ\tau_k(1)\,|\,T]\oplus u([a\circ\tau_k(2)\,|\,T]) \text{ by RC.}
\end{aligned}$$

This establishes the basis for induction (in all cases). First, consider $h = \mathtt{lom}_k(\oplus,u)$. Suppose

$$h[a\,|\,\tau_k(s)\,T] = (\cdots(u[a\circ\tau_k(1)\,|\,T]\oplus u[a\circ\tau_k(2)\,|\,T])\cdots)\oplus u[a\circ\tau_k(s)\,|\,T]$$

holds for some $s \geq 2$ whenever $[a|\tau_k(s)\,T]$ is a subarray of A. If $[a|\tau_k(s+1)\,T]$ is

any subarray of $\mathtt{A}$, then

$$
\begin{aligned}
&\mathtt{h}\,[\mathtt{a}\,|\,\tau_{\mathtt{k}}(\mathtt{s}+1)\,\mathtt{T}] \\
&= \mathtt{h}\,([\mathtt{a}\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}] +\!\!+_{\mathtt{k}}\, [\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s}+1)\,|\,\tau_{\mathtt{k}}(1)\,\mathtt{T}]) \text{ by CI} \\
&= \mathtt{h}\,([\mathtt{a}\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}] +\!\!+_{\mathtt{k}}\, [\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s}+1)\,|\,\mathtt{T}]) \text{ by TI} \\
&= \mathtt{h}\,([\mathtt{a}|\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}] +\!\!+_{\mathtt{k}} \mathtt{reax}_{\mathtt{k}}(\mathtt{collapse}_{\mathtt{k}}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s}+1)\,|\mathtt{T}])) \text{ by RC} \\
&= \mathtt{h}\,[\mathtt{a}\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}] \oplus \mathtt{u}\,(\mathtt{reax}_{\mathtt{k}}\,(\mathtt{collapse}_{\mathtt{k}}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s}+1)\,|\mathtt{T}])) \\
&\qquad \text{since } \mathtt{h} = \mathtt{lom}_{\mathtt{k}}(\oplus,\mathtt{u}) \\
&= \mathtt{h}\,[\mathtt{a}\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}] \oplus \mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s}+1)\,|\mathtt{T}] \text{ by RC} \\
&= (\cdots(\mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(1)\,|\mathtt{T}] \oplus \mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(2)\,|\mathtt{T}])\cdots) \oplus \mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s}+1)\,|\mathtt{T}] \\
&\qquad \text{by the inductive hypothesis.}
\end{aligned}
$$

By the principle of mathematical induction, the stated result holds for $\mathtt{lom}_{\mathtt{k}}(\oplus,\mathtt{u})$.

Second, consider $\mathtt{h} = \mathtt{rom}_{\mathtt{k}}(\oplus,\mathtt{u})$. Suppose

$$
\begin{aligned}
&\mathtt{h}\,[\mathtt{a}\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}] \\
&= \mathtt{u}\,([\mathtt{a}\circ\tau_{\mathtt{k}}(1)\,|\mathtt{T}]) \oplus (\cdots(\mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s}-1)\,|\mathtt{T}] \oplus \mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s})\,|\mathtt{T}])\cdots)
\end{aligned}
$$

holds for some $\mathtt{s} \geq 2$ whenever $[\mathtt{a}|\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}]$ is a subarray of $\mathtt{A}$. If $[\mathtt{a}\,|\,\tau_{\mathtt{k}}(\mathtt{s}+1)\,\mathtt{T}]$ is any subarray of $\mathtt{A}$, then

$$
\begin{aligned}
&\mathtt{h}\,[\mathtt{a}\,|\,\tau_{\mathtt{k}}(\mathtt{s}+1)\,\mathtt{T}] \\
&= \mathtt{h}\,([\mathtt{a}\,|\,\tau_{\mathtt{k}}(1)\,\mathtt{T}] +\!\!+_{\mathtt{k}}\, [\mathtt{a}\circ\tau_{\mathtt{k}}(2)\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}]) \text{ by CI} \\
&= \mathtt{h}\,([\mathtt{a}\circ\tau_{\mathtt{k}}(1)\,|\,\mathtt{T}] +\!\!+_{\mathtt{k}}\, [\mathtt{a}\circ\tau_{\mathtt{k}}(2)\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}]) \text{ by TI} \\
&= \mathtt{h}\,((\mathtt{reax}_{\mathtt{k}}\,(\mathtt{collapse}_{\mathtt{k}}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(1)\,|\mathtt{T}])) +\!\!+_{\mathtt{k}}\, [\mathtt{a}\circ\tau_{\mathtt{k}}(2)\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}]) \text{ by RC} \\
&= \mathtt{u}\,(\mathtt{reax}_{\mathtt{k}}\,(\mathtt{collapse}_{\mathtt{k}}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(1)\,|\mathtt{T}])) \oplus \mathtt{h}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(2)\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}] \\
&\qquad \text{since } \mathtt{h} = \mathtt{rom}_{\mathtt{k}}(\oplus,\mathtt{u}) \\
&= \mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(1)\,|\mathtt{T}] \oplus \mathtt{h}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(2)\,|\,\tau_{\mathtt{k}}(\mathtt{s})\,\mathtt{T}] \text{ by RC} \\
&= \mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(1)\,|\mathtt{T}] \oplus (\mathtt{u}\,([(\mathtt{a}\circ\tau_{\mathtt{k}}(2))\circ\tau_{\mathtt{k}}(1)\,|\mathtt{T}]) \oplus \\
&\quad (\cdots(\mathtt{u}\,[(\mathtt{a}\circ\tau_{\mathtt{k}}(2))\circ\tau_{\mathtt{k}}(\mathtt{s}-1)\,|\mathtt{T}] \oplus \mathtt{u}\,[(\mathtt{a}\circ\tau_{\mathtt{k}}(2))\circ\tau_{\mathtt{k}}(\mathtt{s})\,|\mathtt{T}])\cdots)) \\
&\qquad \text{by the inductive hypothesis} \\
&= \mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(1)\,|\mathtt{T}] \oplus (\mathtt{u}\,([\mathtt{a}\circ\tau_{\mathtt{k}}(2)\,|\mathtt{T}]) \oplus \\
&\quad (\cdots(\mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s})\,|\mathtt{T}] \oplus \mathtt{u}\,[\mathtt{a}\circ\tau_{\mathtt{k}}(\mathtt{s}+1)\,|\mathtt{T}]))\cdots) \\
&\qquad \text{by TIP}
\end{aligned}
$$

By the principle of mathematical induction, the stated identity holds for $\mathrm{rom}_k(\oplus, u)$.

Third, consider $h = (\oplus, u)_k$. Suppose

$$h[a \mid \tau_k(s)\,T] = u[a \circ \tau_k(1) \mid T] \oplus u[a \circ \tau_k(2) \mid T] \oplus \cdots \oplus u[a \circ \tau_k(s) \mid T]$$

holds for some $s \geq 2$ whenever $[a \mid \tau_k(s)\,T]$ is a subarray of A. If $[a \mid \tau_k(s+1)\,T]$ is any subarray of A, then

$h[a \mid \tau_k(s+1)\,T]$
$= h([a \mid \tau_k(m)\,T] \#_k [a \circ \tau_k(m+1) \mid \tau_k(s+1-m)\,T])$ by CI
$= h[a \mid \tau_k(m)\,T] \oplus h[a \circ \tau_k(m+1) \mid \tau_k(s+1-m)\,T]$ as $h = (\oplus, u)_k$
$= (u[a \circ \tau_k(1) \mid T] \oplus u[a \circ \tau_k(2) \mid T] \oplus \cdots \oplus u[a \circ \tau_k(m) \mid T]) \oplus$
$(u[(a \circ \tau_k(m+1)) \circ \tau_k(1) \mid T] \oplus$
$u[(a \circ \tau_k(m+1)) \circ \tau_k(2) \mid T] \oplus \cdots \oplus u[(a \circ \tau_k(m+1)) \circ \tau_k(s+1-m) \mid T])$
by the inductive hypothesis
$= u[a \circ \tau_k(1) \mid T] \oplus u[a \circ \tau_k(2) \mid T] \oplus \cdots \oplus u[a \circ \tau_k(s+1) \mid T]$
since m is arbitrary.

By the principle of mathematical induction, the stated identity holds for $(\oplus, u)_k$. ■

THEOREM The unique extension property extends to left array homs, right array homs and array catamorphisms.

PROOF Assume h and h' are left array homs for a n-array A with $n > 1$ utilizing some binary operator $\oplus$ and unary function u. Let $\mathrm{reax}_k\ C$ be a subarray of A. Then

$$h(\mathrm{reax}_k\ C) = u(\mathrm{reax}_k\, C) = h'(\mathrm{reax}_k\, C),$$

which establishes the basis for induction. Assume $h(B) = h'(B)$ whenever B, which is a subarray of A, is a hyperplane or a concatenation of a sufficiently small number of them. Let B be any subarray of A. By Theorem 4.10, B is a hyperplane or a concatenation of finitely many of them. Using planar forms, write

$$B = C \#_k (\mathrm{reax}_k\, D).$$

As h and h' are left array homs,

$$\begin{aligned} h(B) &= h(C \#_k (\mathrm{reax}_k\, D)) \\ &= h(C) \oplus h(\mathrm{reax}_k\, D) \\ &= h'(C) \oplus h'(\mathrm{reax}_k\, D) \text{ by induction} \\ &= h'(C \#_k (\mathrm{reax}_k\, D)) \\ &= h'(B). \end{aligned}$$

For surely, h and h' agree on a concatenation of two hyperplanes (C is a hyperplane), and then they must agree for concatenations of three hyperplanes, and so forth. If $h(\square)$ and $h'(\square)$ are defined, then

$$h(\square) = h'(\square)$$

since both are required to be equal to the unique identity element for $\oplus$. This establishes uniqueness for left array homs. A completely symmetrical argument holds for right array homs (replace

$$C \#_k (\text{reax}_k D)$$

by

$$(\text{reax}_k C) \#_k D$$

in the preceding discussion and use equation (4.8) instead of equation (4.7)).

Now assume h, h' are array catamorphisms for an array A with binary operator $\oplus$ and unary function u. The prior work shows that h and h' agree on hyperplanes. Let $B \#_k C$ be a subarray of A with $B, C \neq \square$. Then

$$\begin{aligned} h(B \# C) &= h(B) \oplus h(C) \\ &= h'(B) \oplus h'(C) \text{ by induction} \\ &= h'(B \# C). \end{aligned}$$

By Theorem 4.10, B and C are hyperplanes or concatenations of finitely many of them. Since h and h' agree on hyperplanes, they must agree on a concatenation of any two of them, and then on any concatenation involving up to four hyperplanes and so forth. If catamorphisms h and h' are defined on an empty array then again $h(\square) = h'(\square)$, by uniqueness of identity elements. This establishes uniqueness for array catamorphisms and completes the proof of the unique extension property for array homomorphisms. ■

THEOREM If h is any homomorphism such that

$$h(a \oplus b) = h(a) \otimes h(b),$$

then

$$\begin{aligned} h \circ \text{lom}_k(\oplus, u) &= \text{lom}_k(\otimes, h \circ u), \\ h \circ \text{rom}_k(\oplus, u) &= \text{rom}_k(\otimes, h \circ u), \\ h \circ (\oplus, u)_k &= (\otimes, h \circ u)_k. \end{aligned}$$

PROOF Assume $\mathtt{lom}_k(\oplus, u)$ is defined for some n-array A with $n > 1$, that

$$h(\mathtt{lom}_k(\oplus, u)(B +\!\!+_k (\mathrm{reax}_k C))),$$
$$h(\mathtt{lom}_k(\oplus, u)B),$$

are defined whenever $B +\!\!+_k (\mathrm{reax}_k C)$ is a valid subarray of A, and that

$$h(u(\mathrm{reax}_k C))$$

is defined whenever $\mathrm{reax}_k C$ is a subarray of A. Let $\mathrm{reax}_k C$ be a subarray of A.

Calculate

$$\begin{aligned}(h \circ \mathtt{lom}_k(\oplus, u))(\mathrm{reax}_k C) &= h(\mathtt{lom}_k(\oplus, u)(\mathrm{reax}_k C)) \\ &= h(u(\mathrm{reax}_k C)) \\ &= (h \circ u)(\mathrm{reax}_k C).\end{aligned}$$

If $B +\!\!+_k (\mathrm{reax}_k C)$ is a valid subarray of A then

$$\begin{aligned}&(h \circ \mathtt{lom}_k(\oplus, u))(B +\!\!+_k (\mathrm{reax}_k C)) \\ &= h(\mathtt{lom}_k(\oplus, u)(B +\!\!+_k (\mathrm{reax}_k C))) \\ &= h((\mathtt{lom}_k(\oplus, u)B) \oplus (\mathtt{lom}_k(\oplus, u)(\mathrm{reax}_k C))) \\ &= h(\mathtt{lom}_k(\oplus, u)B) \otimes h(\mathtt{lom}_k(\oplus, u)(\mathrm{reax}_k C)) \\ &= (h \circ \mathtt{lom}_k(\oplus, u))B \otimes (h \circ \mathtt{lom}_k(\oplus, u))(\mathrm{reax}_k C).\end{aligned}$$

This shows that $h \circ \mathtt{lom}_k(\oplus, u)$ is a left hom. By uniqueness,

$$h \circ \mathtt{lom}_k(\oplus, u) = \mathtt{lom}_k(\otimes, h \circ u).$$

A symmetrical argument shows that

$$h \circ \mathtt{rom}_k(\oplus, u) = \mathtt{rom}_k(\otimes, h \circ u).$$

Next assume $(\oplus, u)_k$ is defined for some array A, that

$$h((\oplus, u)_k(B +\!\!+_k C)), \quad h((\oplus, u)_k B) \quad \text{and} \quad h((\oplus, u)_k C),$$

are defined whenever $B +\!\!+_k C$ is a valid subarray of A, and that

$$h(u(\mathrm{reax}_k C))$$

is defined whenever $\mathrm{reax}_k C$ is a subarray of A. Let $\mathrm{reax}_k C$ be a subarray of A.

Compute

$$\begin{aligned}(h \circ (\oplus, u)_k)\,(\text{reax}_k\ C) &= h\,((\oplus, u)_k\,(\text{reax}_k\ C)) \\ &= h\,(u\,(\text{reax}_k\,C)) \\ &= (h \circ u)\,(\text{reax}_k\,C).\end{aligned}$$

If $B +\!\!+_k C$ is a valid subarray of A, then

$$\begin{aligned}(h \circ (\oplus, u)_k)\,(B +\!\!+_k C) &= h\,((\oplus, u)_k\,(B +\!\!+_k C)) \\ &= h\,(((\oplus, u)_k B) \oplus ((\oplus, u)_k C)) \\ &= h\,((\oplus, u)_k B) \otimes h\,((\oplus, u)_k C) \\ &= (h \circ (\oplus, u)_k)\,B \otimes (h \circ (\oplus, u)_k)\,C.\end{aligned}$$

This shows that $h \circ (\oplus, u)_k$ is a catamorphism. Hence,

$$h \circ (\oplus, u)_k = (\otimes, h \circ u)_k,$$

by uniqueness. ■

THEOREM Let

$$A = [a\,|\,(t : n)]$$

with $n > 1$ and

$$T = (t : n; t_k = 1).$$

Then

$$\begin{aligned}\nearrow_k \underset{\longleftrightarrow}{\oplus} A &= [\smile_k \oplus A\,|\,T], \\ \nwarrow_k \underset{\longleftrightarrow}{\oplus} A &= [\frown_k \oplus A\,|\,T],\end{aligned}$$

and

$$/_k \underset{\longleftrightarrow}{\oplus} A = [\asymp_k \oplus A\,|\,T].$$

PROOF Trivial if the length of the k-axis is one, i.e., $t_k = 1$. Assume the theorem holds whenever

$$t_k \leq p$$

for some $p \geq 1$. Suppose

$$t_k = p + 1.$$

Invoking CI write

$$[a\,|\,(t : n)] = [a\,|\,\tau_k(p)\,T] +\!\!+_k [a \circ \tau_k(p+1)\,|\,T],$$

since

$$t_k - p = (p+1) - p = 1$$

and

$$\tau_k(1)T = T$$

by TI. Calculate

$$\begin{aligned}
&\nrightarrow_k \underset{\longleftrightarrow}{\oplus} A \\
&= \nrightarrow_k \underset{\longleftrightarrow}{\oplus} ([a \mid \tau_k(p)T] +\!\!+_k [a \circ \tau_k(p+1) \mid T]) \\
&= \left(\nrightarrow_k \underset{\longleftrightarrow}{\oplus} [a \mid \tau_k(p)T]\right) \underset{\longleftrightarrow}{\oplus} \left(\nrightarrow_k \underset{\longleftrightarrow}{\oplus} [a \circ \tau_k(p+1) \mid T]\right) \\
&\qquad \text{by definition of } \nrightarrow_k \\
&= \left(\nrightarrow_k \underset{\longleftrightarrow}{\oplus} [a \mid \tau_k(p)T]\right) \underset{\longleftrightarrow}{\oplus} ([a \circ \tau_k(p+1) \mid T]) \text{ by definition of } \nrightarrow_k \\
&= [\smile_k \oplus [a \mid \tau_k(p)T] \mid T] \underset{\longleftrightarrow}{\oplus} [a \circ \tau_k(p+1) \mid T] \\
&\qquad \text{by inductive hypothesis} \\
&= [\oplus \circ (\smile_k \oplus [a \mid \tau_k(p)T], a \circ \tau_k(p+1)) \mid T] \text{ by Definition 3.27} \\
&= [\smile_k \oplus A \mid T] \text{ by Definition 3.33.}
\end{aligned}$$

By the principle of mathematical induction,

$$\nrightarrow_k \underset{\longleftrightarrow}{\oplus} A = [\smile_k \oplus A \mid T]$$

holds always. Replacing $\nrightarrow_k$ by $/_k$ and $\smile_k$ by $\asymp_k$ in the preceding equations,

$$/_k \underset{\longleftrightarrow}{\oplus} A = [\asymp_k \oplus A \mid T].$$

Invoke CI and TI again to write

$$[a \mid (t : n)] = [a \mid T] +\!\!+_k [a \circ \tau_k(2) \mid \tau_k(p)T]$$

and apply a symmetrical argument to obtain

$$\nleftarrow_k \underset{\longleftrightarrow}{\oplus} A = [\frown_k \oplus A \mid T],$$

which completes the proof. ■

LEMMA

$$\begin{aligned}
\text{reax}_k(A +\!\!+_p B) &= (\text{reax}_k A) +\!\!+_p (\text{reax}_k B) \quad \text{if } p < k, \\
\text{reax}_k(A +\!\!+_p B) &= (\text{reax}_k A) +\!\!+_{p+1} (\text{reax}_k B) \quad \text{if } p \geq k.
\end{aligned}$$

PROOF Let $A, B \in \mathcal{A}$ and write

$$A = [a \mid (t : n)] \quad \text{and} \quad B = [b \mid (s : m)].$$

In order for $A \#_p B$ to be defined, the shape tuple can differ only in the pth component which means

$$(t:n) = (s:m; s_p = t_p).$$

This implies $m = n$ and $s_i = t_i$ for all $i \neq p$. Hence

$$B = [b|\,(t:n; t_p = s_p)];$$

wherefore,

$$C = A \#_p B = [c|\,(t:n; t_p = t_p + s_p)],$$

where

$$c(i:n) = \begin{cases} a(i:n) & \text{if } 1 \leq i_p \leq t_p \\ b(i:n; i_p = i_p - t_p) & \text{if } t_p < i_p \leq t_p + s_p \end{cases}.$$

Thus,

$$\text{reax}_k\, C = [c \circ \langle\rangle_k \mid \sqsupset_k (1)(t:n; t_p = t_p + s_p)]$$

and

$$c \circ \langle\rangle_k (\sqsupset_k (1)(i:n)) = c(i:n) \text{ by DR.}$$

In addition,

$$\text{reax}_k\, A = \text{reax}_k [a|\,(t:n)] = [a \circ \langle\rangle_k \mid \sqsupset_k (1)(t:n)].$$

Moreover,

$$\text{reax}_k\, B = \text{reax}_k\, [b|\,(s:m)] = [b \circ \langle\rangle_k \mid \sqsupset_k (1)(s:m)].$$

First assume $p < k$. For

$$D = (\text{reax}_k\, A) \#_p (\text{reax}_k\, B)$$

to be defined, the shape tuples must be equal except for the pth component, which means (after setting the pth components equal)

$$\sqsupset_k (1)(t:n) = \sqsupset_k (1)(s:m; s_p = t_p).$$

As $\sqsupset_k$ is well-defined, $m = n$ and $s_i = t_i$ for all $i \neq p$. Hence

$$B = [b|\,(t:n; t_p = s_p)].$$

Apply reax_k to obtain

$$\text{reax}_k\, B = [b \circ \langle\rangle_k \mid \sqsupset_k (1)(t:n; t_p = s_p)].$$

Now

$$D = [d|\,\sqsupset_k (1)(t:n; t_p = t_p + s_p)]$$

where

$$\begin{aligned}
&\mathtt{d}(\sqsupset_{\mathtt{k}}(1)(\mathtt{i}:\mathtt{n}))\\
&= \begin{cases} \mathtt{a} \circ \wr_{\mathtt{k}}(\sqsupset_{\mathtt{k}}(1)(\mathtt{i}:\mathtt{n})) & \text{if } 1 \le \mathtt{i}_{\mathtt{p}} \le \mathtt{t}_{\mathtt{p}} \\ \mathtt{b} \circ \wr_{\mathtt{k}}(\sqsupset_{\mathtt{k}}(1)(\mathtt{i}:\mathtt{n};\mathtt{i}_{\mathtt{p}} = \mathtt{i}_{\mathtt{p}} - \mathtt{t}_{\mathtt{p}})) & \text{if } \mathtt{t}_{\mathtt{p}} < \mathtt{i}_{\mathtt{p}} \le \mathtt{t}_{\mathtt{p}} + \mathtt{s}_{\mathtt{p}} \end{cases}\\
&= \begin{cases} \mathtt{a}(\mathtt{i}:\mathtt{n}) & \text{if } 1 \le \mathtt{i}_{\mathtt{p}} \le \mathtt{t}_{\mathtt{p}} \\ \mathtt{b}(\mathtt{i}:\mathtt{n};\mathtt{i}_{\mathtt{p}} = \mathtt{i}_{\mathtt{p}} - \mathtt{t}_{\mathtt{p}}) & \text{if } \mathtt{t}_{\mathtt{p}} < \mathtt{i}_{\mathtt{p}} \le \mathtt{t}_{\mathtt{p}} + \mathtt{s}_{\mathtt{p}} \end{cases} \text{ by DR}\\
&= \mathtt{c}(\mathtt{i}:\mathtt{n}).
\end{aligned}$$

By inspection,

$$\mathrm{reax}_{\mathtt{k}}\,\mathtt{C} = \mathtt{D},$$

which verifies

$$\mathrm{reax}_{\mathtt{k}}(\mathtt{A} \mathbin{\#}_{\mathtt{p}} \mathtt{B}) = (\mathrm{reax}_{\mathtt{k}}\,\mathtt{A}) \mathbin{\#}_{\mathtt{p}} (\mathrm{reax}_{\mathtt{k}}\,\mathtt{B}) \text{ if } \mathtt{p} < \mathtt{k}.$$

Second assume $\mathtt{p} \ge \mathtt{k}$. For

$$\mathtt{D} = (\mathrm{reax}_{\mathtt{k}}\,\mathtt{A}) \mathbin{\#}_{\mathtt{p}+1} (\mathrm{reax}_{\mathtt{k}}\,\mathtt{B})$$

to be defined, the shape tuples must be equal except for the $(\mathtt{p}+1)$ st component, which means (after setting the $\mathtt{p}$th components equal since $\sqsupset_{\mathtt{k}}$ moves $\mathtt{s}_{\mathtt{p}}$ to $\mathtt{s}_{\mathtt{p}+1}$)

$$\sqsupset_{\mathtt{k}}(1)(\mathtt{t}:\mathtt{n}) = \sqsupset_{\mathtt{k}}(1)(\mathtt{s}:\mathtt{m};\mathtt{s}_{\mathtt{p}} = \mathtt{t}_{\mathtt{p}}).$$

As $\sqsupset_{\mathtt{k}}$ is well-defined, $\mathtt{m} = \mathtt{n}$ and $\mathtt{s}_{\mathtt{i}} = \mathtt{t}_{\mathtt{i}}$ for all $\mathtt{i} \ne \mathtt{p}$. Hence

$$\mathtt{B} = [\mathtt{b} \mid (\mathtt{t}:\mathtt{n};\mathtt{t}_{\mathtt{p}} = \mathtt{s}_{\mathtt{p}})].$$

By inspection,

$$\mathrm{reax}_{\mathtt{k}}\,\mathtt{C} = \mathtt{D},$$

which verifies

$$\mathrm{reax}_{\mathtt{k}}(\mathtt{A} \mathbin{\#}_{\mathtt{p}} \mathtt{B}) = (\mathrm{reax}_{\mathtt{k}}\,\mathtt{A}) \mathbin{\#}_{\mathtt{p}+1} (\mathrm{reax}_{\mathtt{k}}\,\mathtt{B}) \text{ if } \mathtt{p} \ge \mathtt{k},$$

which completes the proof. ■

THEOREM For all $\mathtt{n}$-arrays, $\mathtt{k} \in \iota\,\mathtt{n}$ and $\mathtt{p} \in \iota\,(\mathtt{n}-1)$,

$$\left(\rightsquigarrow_{\mathtt{p}} \underset{\longleftrightarrow}{\oplus}\right) \circ \mathrm{collapse}_{\mathtt{k}} = \mathrm{collapse}_{\mathtt{k}} \circ \left(\rightsquigarrow_{\mathtt{p}+1} \underset{\longleftrightarrow}{\oplus}\right) \text{ if } \mathtt{p} \ge \mathtt{k},$$

$$\left(\rightsquigarrow_{\mathtt{p}} \underset{\longleftrightarrow}{\oplus}\right) \circ \mathrm{collapse}_{\mathtt{k}} = \mathrm{collapse}_{\mathtt{k}} \circ \left(\rightsquigarrow_{\mathtt{p}} \underset{\longleftrightarrow}{\oplus}\right) \text{ if } \mathtt{p} < \mathtt{k},$$

$$\left(\nleftarrow_{p} \underset{\longleftrightarrow}{\oplus}\right) \circ \mathrm{collapse}_{k} = \mathrm{collapse}_{k} \circ \left(\nleftarrow_{p+1} \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if } p \geq k,$$
$$\left(\nleftarrow_{p} \underset{\longleftrightarrow}{\oplus}\right) \circ \mathrm{collapse}_{k} = \mathrm{collapse}_{k} \circ \left(\nleftarrow_{p} \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if } p < k,$$

and

$$\left(\nearrow_{p} \underset{\longleftrightarrow}{\oplus}\right) \circ \mathrm{collapse}_{k} = \mathrm{collapse}_{k} \circ \left(\nearrow_{p+1} \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if } p \geq k,$$
$$\left(\nearrow_{p} \underset{\longleftrightarrow}{\oplus}\right) \circ \mathrm{collapse}_{k} = \mathrm{collapse}_{k} \circ \left(\nearrow_{p} \underset{\longleftrightarrow}{\oplus}\right) \quad \text{if } p < k.$$

PROOF In order to compute

$$\mathrm{collapse}_{k}\, A$$

it must be possible to write

$$A = [a \mid \sqsupset_{k} (1)(t : n)].$$

Put

$$T = (t : n; t_{p} = 1).$$

Then

$$\mathrm{collapse}_{k}\, [a \mid \sqsupset_{k} (1)(t : n)] = [a \circ \sqsupset_{k} (1) \mid (t : n)].$$

By Theorem 4.38

$$\left(\nearrow_{p} \underset{\longleftrightarrow}{\oplus}\right) [a \circ \sqsupset_{k} (1) \mid (t : n)] = [\asymp_{p} \oplus [a \circ \sqsupset_{k} (1) \mid (t : n)] \mid T].$$

By composition,

$$\left(\nearrow_{p} \underset{\longleftrightarrow}{\oplus}\right) \circ \mathrm{collapse}_{k}\, [a \mid \sqsupset_{k} (1)(t : n)] = [\asymp_{p} \oplus [a \circ \sqsupset_{k} (1) \mid (t : n)] \mid T].$$

If $p < k$ then

$$\begin{aligned} &\left(\nearrow_{p} \underset{\longleftrightarrow}{\oplus}\right) [a \mid \sqsupset_{k} (1)(t : n)] \\ &= [\asymp_{p} \oplus [a \mid \sqsupset_{k} (1)(t : n)] \mid \sqsupset_{k} (1)(t : n; t_{p} = 1)] \\ &= [\asymp_{p} \oplus [a \mid \sqsupset_{k} (1)(t : n)] \mid \sqsupset_{k} (1) T]. \end{aligned}$$

Now

$$\begin{aligned} &\mathrm{collapse}_{k}\, [\asymp_{p} \oplus [a \mid \sqsupset_{k} (1)(t : n)] \mid \sqsupset_{k} (1) T] \\ &\qquad = [(\asymp_{p} \oplus [a \mid \sqsupset_{k} (1)(t : n)]) \circ \sqsupset_{k} (1) \mid T] \end{aligned}$$

Calculate

$$\asymp_{\mathtt{p}} \oplus [\mathtt{a} \circ \sqsupset_{\mathtt{k}} (1) \mid (\mathtt{t} : \mathtt{n})] \; (\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{p}} = 1)$$
$$= (\mathtt{a} \circ \sqsupset_{\mathtt{k}} (1) (\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{p}} = 1)) \oplus \cdots \oplus (\mathtt{a} \circ \sqsupset_{\mathtt{k}} (1) (\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{p}} = \mathtt{t}_{\mathtt{p}}))$$

and

$$(\asymp_{\mathtt{p}} \oplus [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})]) \circ \sqsupset_{\mathtt{k}} (1) \; (\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{p}} = 1)$$
$$= (\mathtt{a} \circ \sqsupset_{\mathtt{k}} (1) (\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{p}} = 1)) \oplus \cdots \oplus (\mathtt{a} \circ \sqsupset_{\mathtt{k}} (1) (\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{p}} = \mathtt{t}_{\mathtt{p}})).$$

By inspection, if $\mathtt{p} < \mathtt{k}$ then

$$[\asymp_{\mathtt{p}} \oplus [\mathtt{a} \circ \sqsupset_{\mathtt{k}} (1) \mid (\mathtt{t} : \mathtt{n})] \mid \mathtt{T}]$$
$$= [(\asymp_{\mathtt{p}} \oplus [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})]) \circ \sqsupset_{\mathtt{k}} (1) \mid \mathtt{T}].$$

By transitivity, if $\mathtt{p} < \mathtt{k}$ then

$$\left(\diagup_{\mathtt{p}} \underset{\longleftrightarrow}{\oplus}\right) \circ \mathrm{collapse}_{\mathtt{k}} [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})]$$
$$= \mathrm{collapse}_{\mathtt{k}} \circ \left(\diagup_{\mathtt{p}} \underset{\longleftrightarrow}{\oplus}\right) [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})].$$

If $\mathtt{p} \geq \mathtt{k}$ then

$$\left(\diagup_{\mathtt{p}+1} \underset{\longleftrightarrow}{\oplus}\right) [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})]$$
$$= [\asymp_{\mathtt{p}+1} \oplus [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})] \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n}; \mathtt{t}_{\mathtt{p}} = 1)]$$
$$= [\asymp_{\mathtt{p}+1} \oplus [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})] \mid \sqsupset_{\mathtt{k}} (1) \mathtt{T}].$$

Now

$$\mathrm{collapse}_{\mathtt{k}} [\asymp_{\mathtt{p}+1} \oplus [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})] \mid \sqsupset_{\mathtt{k}} (1) \mathtt{T}]$$
$$= [(\asymp_{\mathtt{p}+1} \oplus [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})]) \circ \sqsupset_{\mathtt{k}} (1) \mid \mathtt{T}]$$

Calculate

$$(\asymp_{\mathtt{p}+1} \oplus [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})]) \circ \sqsupset_{\mathtt{k}} (1) (\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{p}} = 1)$$
$$= (\mathtt{a} \circ \sqsupset_{\mathtt{k}} (1) (\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{p}} = 1)) \oplus \cdots \oplus (\mathtt{a} \circ \sqsupset_{\mathtt{k}} (1) (\mathtt{i} : \mathtt{n}; \mathtt{i}_{\mathtt{p}} = \mathtt{t}_{\mathtt{p}})).$$

By inspection, if $\mathtt{p} \geq \mathtt{k}$ then

$$[\asymp_{\mathtt{p}} \oplus [\mathtt{a} \circ \sqsupset_{\mathtt{k}} (1) \mid (\mathtt{t} : \mathtt{n})] \mid \mathtt{T}] = [(\asymp_{\mathtt{p}+1} \oplus [\mathtt{a} \mid \sqsupset_{\mathtt{k}} (1) (\mathtt{t} : \mathtt{n})]) \circ \sqsupset_{\mathtt{k}} (1) \mid \mathtt{T}].$$

By transitivity, if $\mathtt{p} \geq \mathtt{k}$ then

$$\left(\diagup_{\mathtt{p}} \overset{\oplus}{\longleftrightarrow}\right) \circ \mathrm{collapse}_{\mathtt{k}}[\mathtt{a} \mid \sqsupset_{\mathtt{k}}(1)(\mathtt{t}:\mathtt{n})] = \mathrm{collapse}_{\mathtt{k}} \circ \left(\diagup_{\mathtt{p}+1} \overset{\oplus}{\longleftrightarrow}\right)[\mathtt{a} \mid \sqsupset_{\mathtt{k}}(1)(\mathtt{t}:\mathtt{n})].$$

The identities for left and right folds follow by symmetrical computations. ■

THEOREM The following identity holds always:

$$\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}] = (\multimap[\mathtt{f} \mid \mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}} \mid \mathtt{T}].$$

PROOF Assume $\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}]\ \mathtt{X}$ is defined. Let $\mathtt{A} = [\mathtt{a} \mid \sqsupset_{\mathtt{k}}(\mathtt{s})\mathtt{T}]$ be a subarray of $\mathtt{X}$. Apply Definition 3.31 to write

$$\circledast_{\mathtt{k}}[\mathtt{f} \mid \mathtt{T}]\ [\mathtt{a} \mid \sqsupset_{\mathtt{k}}(\mathtt{s})\mathtt{T}] = [\mathtt{h} \mid \mathtt{T}]$$

where

$$\mathtt{h}(\mathtt{i}:\mathtt{n}) = \mathtt{f}(\mathtt{i}:\mathtt{n})\left[\mathtt{b}^{(\mathtt{i}:\mathtt{n})} : \mathtt{s}\right]$$

and the 1-array $\left[\mathtt{b}^{(\mathtt{i}:\mathtt{n})} : \mathtt{s}\right]$ is an axis of $\mathtt{A}$ whose components are given by

$$\mathtt{b}_{\mathtt{j}}^{(\mathtt{i}:\mathtt{n})} = \mathtt{a} \circ \sqsupset_{\mathtt{k}}(\mathtt{j})(\mathtt{i}:\mathtt{n}).$$

Apply Definition 3.31 again to write

$$\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}} \mid \mathtt{T}]\ [\mathtt{a} \mid \sqsupset_{\mathtt{k}}(\mathtt{s})\mathtt{T}] = [\mathtt{g} \mid \mathtt{T}]$$

where

$$\mathtt{g}(\mathtt{i}:\mathtt{n}) = \left[\mathtt{b}^{(\mathtt{i}:\mathtt{n})} : \mathtt{s}\right].$$

In view of Definition 3.30,

$$(\multimap[\mathtt{f} \mid \mathtt{T}])[\mathtt{g} \mid \mathtt{T}] = [@ \circ (\mathtt{f}, \mathtt{g}) \mid \mathtt{T}].$$

By inspection,

$$@ \circ (\mathtt{f}, \mathtt{g})(\mathtt{i}:\mathtt{n}) = \mathtt{f}(\mathtt{i}:\mathtt{n})\mathtt{g}(\mathtt{i}:\mathtt{n}) = \mathtt{f}(\mathtt{i}:\mathtt{n})\left[\mathtt{b}^{(\mathtt{i}:\mathtt{n})} : \mathtt{s}\right] = \mathtt{h}(\mathtt{i}:\mathtt{n}).$$

It follows that

$$[\mathtt{h} \mid \mathtt{T}] = [@ \circ (\mathtt{f}, \mathtt{g}) \mid \mathtt{T}],$$

which completes the proof by transitivity. ■

THEOREM (AXIAL MORPHISM THEOREM) Assume $\mathtt{F} = [\mathtt{f} \mid \mathtt{T}]$ is a harray. For $\mathtt{t} \in \mathbb{P}(\mathtt{T})$, there exist a binary operator $\theta(\mathtt{t})$ and an unary function $\mu(\mathtt{t})$ which

determine each homomorphism $\mathtt{f}(\mathtt{t})$. If every element of $\mathtt{F}$ is a left hom, then

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[\twoheadrightarrow\circ\theta\,|\,\mathtt{T}])\circ(\multimap[*\circ\mu\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}].$$

If every element of $\mathtt{F}$ is a right hom, then

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[\twoheadleftarrow\circ\theta\,|\,\mathtt{T}])\circ(\multimap[*\circ\mu\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}].$$

If every element of $\mathtt{F}$ is a catamorphism, then

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[\diagup\circ\theta\,|\,\mathtt{T}])\circ(\multimap[*\circ\mu\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}].$$

PROOF Assume $\mathtt{F} = [\mathtt{f}\,|\,\mathtt{T}]$ is a harray and $\circledast_{\mathtt{k}}\mathtt{F}\ \mathtt{X}$ is defined. For $\mathtt{t}\in\mathbb{P}(\mathtt{T})$, $\mathtt{f}(\mathtt{t})$ is a list homomorphism. By the unique extension property there exist a binary operator $\theta_{\mathtt{t}} = \theta(\mathtt{t})$ and a unary function $\mu_{\mathtt{t}} = \mu(\mathtt{t})$ which determine $\mathtt{f}(\mathtt{t})$. Suppose every component homomorphism is a left hom. By the homomorphism lemma,

$$\mathtt{f}(\mathtt{t}) = \mathtt{l}\circ\mathtt{m}(\theta_{\mathtt{t}}, \mathtt{u}_{\mathtt{t}}) = (\twoheadrightarrow\theta_{\mathtt{t}})\circ(*\mu_{\mathtt{t}}).$$

It follows by inspection that

$$[\mathtt{f}\,|\,\mathtt{T}] = [(\circ)\circ(\twoheadrightarrow\circ\theta, *\circ\mu)\,|\,\mathtt{T}]$$

since

$$\begin{aligned}(\circ)\circ(\twoheadrightarrow\circ\theta, *\circ\mu)(\mathtt{t}) &= (\circ)\circ(\twoheadrightarrow\theta(\mathtt{t}), *\mu(\mathtt{t}))\\ &= (\twoheadrightarrow\theta_{\mathtt{t}})\circ(*\mu_{\mathtt{t}}).\end{aligned}$$

Now

$$\begin{aligned}\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] &= (\multimap[\mathtt{f}\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}] \text{ by Biaxis Identity}\\ &= (\multimap[(\circ)\circ(\twoheadrightarrow\circ\theta, *\circ\mu)\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}]\\ &= (\multimap[\twoheadrightarrow\circ\theta\,|\,\mathtt{T}])\circ(\multimap[*\circ\mu\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}]\end{aligned}$$

where the last equation holds by Bicomp Distributivity. If every component homomorphism is a right hom, then by the homomorphism lemma,

$$\mathtt{f}(\mathtt{t}) = \mathtt{r}\circ\mathtt{m}(\theta_{\mathtt{t}}, \mathtt{u}_{\mathtt{t}}) = (\twoheadleftarrow\theta_{\mathtt{t}})\circ(*\mu_{\mathtt{t}}).$$

By inspection,

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[\twoheadleftarrow\circ\,\theta\,|\,\mathtt{T}])\circ(\multimap[*\circ\ \mu\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}].$$

Lastly, if every component homomorphism is a catamorphism, then by the homomorphism lemma,

$$\mathtt{f}(\mathtt{t}) = (\!|\theta_{\mathtt{t}}, \mu_{\mathtt{t}}|\!) = (\diagup\theta_{\mathtt{t}})\circ(*\mu_{\mathtt{t}}).$$

Thus,

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[\not\swarrow \circ \theta\,|\,\mathtt{T}]) \circ (\multimap[* \circ \mu\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}],$$

which completes the proof. ■

THEOREM If every element of a harray $[\mathtt{f}\,|\,\mathtt{T}]$ is a left fold then

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[\nrightarrow \circ \theta\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}].$$

If every element of a harray $\mathtt{R}$ is a right fold then

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[\nleftarrow \circ \theta\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}].$$

If every element of a harray $\mathtt{R}$ is a reduce operation, then

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[\not\swarrow \circ \theta\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}].$$

PROOF If every element of a harray $[\mathtt{f}\,|\,\mathtt{T}]$ is a left fold then

$$\mathtt{f}(\mathtt{t}) = (\circ) \circ (\nrightarrow \circ \theta, \mathtt{K}_{\mathtt{id}})(\mathtt{t}).$$

Compute

$$\begin{aligned}
\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] &= (\multimap[\mathtt{f}\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}] \text{ by Biaxis Identity} \\
&= (\multimap[(\circ) \circ (\nrightarrow \circ \theta, \mathtt{K}_{\mathtt{id}})\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}] \\
&= (\multimap[\nrightarrow \circ \theta\,|\,\mathtt{T}]) \circ (\multimap[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}],
\end{aligned}$$

where the last equation holds by Bicomp Distributivity. Call upon Lemma 4.52 (see page 133) to obtain

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[\nrightarrow \circ \theta\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}].$$

Similarly, if every element of a harray $[\mathtt{f}\,|\,\mathtt{T}]$ is a right fold then

$$\mathtt{f}(\mathtt{t}) = (\circ) \circ (\nleftarrow \circ \theta, \mathtt{K}_{\mathtt{id}})(\mathtt{t}).$$

If every element of a harray $[\mathtt{f}\,|\,\mathtt{T}]$ is a catamorphism, then

$$\mathtt{f}(\mathtt{t}) = (\circ) \circ (\not\swarrow \circ \theta, \mathtt{K}_{\mathtt{id}})(\mathtt{t}).$$

The stated identities hold by inspection. ■

THEOREM If $\mathtt{M} = [\mathtt{f}\,|\,\mathtt{T}]$ is a map array then

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[* \circ \mu\,|\,\mathtt{T}]) \circ \circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}],$$

where $\mu(\mathtt{t})$ is the unary function which determines the map $\mathtt{f}(\mathtt{t})$.

PROOF If every element of a harray $[\mathtt{f}\,|\,\mathtt{T}]$ is a map then

$$\mathtt{f}(\mathtt{t}) = (\circ)\circ(\mathtt{K}_{\mathtt{id}}, *\circ\mu)(\mathtt{t}).$$

Calculate

$$\begin{aligned}\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] &= (\multimap[\mathtt{f}\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}] \text{ by Biaxis Identity}\\ &= (\multimap[(\circ)\circ(\mathtt{K}_{\mathtt{id}}, *\circ\mu)(\mathtt{t})\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}]\\ &= (\multimap[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}])\circ(\multimap[*\circ\ \mu\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}],\end{aligned}$$

where the last equation holds by Bicomp Distributivity. Call upon Lemma 4.52 (see page 133) to obtain

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}] = (\multimap[*\circ\ \mu\,|\,\mathtt{T}])\circ\circledast_{\mathtt{k}}[\mathtt{K}_{\mathtt{id}}\,|\,\mathtt{T}],$$

which is the desired result. ■

THEOREM If $\mathtt{F}$ is a harray then

$$\circledast_{\mathtt{k}}\mathtt{F} = (\multimap\mathtt{R})\circ(\circledast_{\mathtt{k}}\mathtt{M})$$

where $\mathtt{R}$ is a reducing array, $\mathtt{M}$ is a map array and

$$\mathtt{F} = \mathtt{R}\underset{\longleftrightarrow}{\circ}\mathtt{M}.$$

PROOF Write $\mathtt{F} = [\mathtt{f}\,|\,\mathtt{T}]$. As $\mathtt{F}$ is a harray, in view of the homomorphism lemma it is possible to define for $\mathtt{t}\in\mathbb{P}(\mathtt{T})$, $\mathtt{m}(\mathtt{t}) = *\mathtt{u}_{\mathtt{t}}$ and

$$\mathtt{r}(\mathtt{t}) = \begin{cases} \twoheadrightarrow\oplus_{\mathtt{t}} & \text{if } \mathtt{f}(\mathtt{t}) = (\twoheadrightarrow\oplus_{\mathtt{t}})\circ(*\mathtt{u}_{\mathtt{t}})\\ \twoheadleftarrow\oplus_{\mathtt{t}} & \text{if } \mathtt{f}(\mathtt{t}) = (\twoheadleftarrow\oplus_{\mathtt{t}})\circ(*\mathtt{u}_{\mathtt{t}})\\ \diagup\oplus_{\mathtt{t}} & \text{if } \mathtt{f}(\mathtt{t}) = (\diagup\oplus_{\mathtt{t}})\circ(*\mathtt{u}_{\mathtt{t}})\end{cases}.$$

Put $\mathtt{R} = [\mathtt{r}\,|\,\mathtt{T}]$ and $\mathtt{M} = [\mathtt{m}\,|\,\mathtt{T}]$. By construction,

$$\mathtt{R}\underset{\longleftrightarrow}{\circ}\mathtt{M} = [\circ(\mathtt{r},\mathtt{m})\,|\,\mathtt{T}] = [\mathtt{f}\,|\,\mathtt{T}] = \mathtt{F}.$$

Evidently $\mathtt{R}$ is a reducing array and $\mathtt{M}$ is a map array. Using Definition 3.31,

$$\circledast_{\mathtt{k}}[\mathtt{f}\,|\,\mathtt{T}]\ [\mathtt{a}\,|\,\sqsupset_{\mathtt{k}}(\mathtt{s})(\mathtt{t}:\mathtt{n})] = \left[\mathtt{y}^{\mathtt{f}}\,|\,\mathtt{T}\right]$$

where

$$\mathtt{y}^{\mathtt{f}}(\mathtt{t}) = \mathtt{f}(\mathtt{t})\left[\mathtt{b}^{\mathtt{t}}:\mathtt{s}\right].$$

Now

$$\begin{aligned}&(\multimap \mathtt{R}) \circ (\circledast_{\mathtt{k}} \mathtt{M})\ [\mathtt{a}|\ \sqsupset_{\mathtt{k}} (\mathtt{s})(\mathtt{t} : \mathtt{n})]\\ =&\multimap \mathtt{R}\ [\mathtt{y}^{\mathtt{m}} | \mathtt{T}] \text{ by Definition 3.31}\\ =&\ [@ \circ (\mathtt{r}, \mathtt{y}^{\mathtt{m}}) | \mathtt{T}] \text{ by Definition 3.30}\end{aligned}$$

where

$$\mathtt{y}^{\mathtt{m}}(\mathtt{t}) = \mathtt{m}(\mathtt{t})\left[\mathtt{b}^{\mathtt{t}} : \mathtt{s}\right],$$

and

$$\begin{aligned}@ \circ (\mathtt{r}, \mathtt{y}^{\mathtt{m}})(\mathtt{t}) &= \mathtt{r}(\mathtt{t})(\mathtt{y}^{\mathtt{m}}(\mathtt{t}))\\ &= \mathtt{r}(\mathtt{t})\left(\mathtt{m}(\mathtt{t})\left[\mathtt{b}^{\mathtt{t}} : \mathtt{s}\right]\right)\\ &= (\mathtt{r}(\mathtt{t}) \circ \mathtt{m}(\mathtt{t}))\left[\mathtt{b}^{\mathtt{t}} : \mathtt{s}\right]\\ &= \mathtt{f}(\mathtt{t})\left[\mathtt{b}^{\mathtt{t}} : \mathtt{s}\right]\\ &= \mathtt{y}^{\mathtt{f}}(\mathtt{t}).\end{aligned}$$

This shows that

$$\left[\mathtt{y}^{\mathtt{f}} | \mathtt{T}\right] = [@ \circ (\mathtt{r}, \mathtt{y}^{\mathtt{m}}) | \mathtt{T}]$$

whence

$$\circledast_{\mathtt{k}} [\mathtt{f}|\mathtt{T}]\ [\mathtt{a}|\ \sqsupset_{\mathtt{k}} (\mathtt{s})(\mathtt{t} : \mathtt{n})] = (\multimap \mathtt{R}) \circ (\circledast_{\mathtt{k}} \mathtt{M})\ \ [\mathtt{a}|\ \sqsupset_{\mathtt{k}} (\mathtt{s})(\mathtt{t} : \mathtt{n})].$$

Since $[\mathtt{a}|\ \sqsupset_{\mathtt{k}} (\mathtt{s})(\mathtt{t} : \mathtt{n})]$ is arbitrary, it follows that

$$\circledast_{\mathtt{k}} \mathtt{F} = (\multimap \mathtt{R}) \circ (\circledast_{\mathtt{k}} \mathtt{M}),$$

which is the desired result. ■

CLAIM The concatenation operator $\#_{\mathtt{k}}$ is an associative operator.

PROOF Let $\mathtt{A}, \mathtt{B}, \mathtt{C} \in \mathbb{A}^{\mathtt{n}*}$. Fix $\mathtt{k} \in \iota\, \mathtt{n}$. If $\mathtt{A} = \square$, then

$$\begin{aligned}(\mathtt{A} \#_{\mathtt{k}} \mathtt{B}) \#_{\mathtt{k}} \mathtt{C} &= \mathtt{B} \#_{\mathtt{k}} \mathtt{C}\\ &= \mathtt{A} \#_{\mathtt{k}} (\mathtt{B} \#_{\mathtt{k}} \mathtt{C}).\end{aligned}$$

Similarly, by inspection, if $\mathtt{B} = \square$ or $\mathtt{C} = \square$ then

$$(\mathtt{A} \#_{\mathtt{k}} \mathtt{B}) \#_{\mathtt{k}} \mathtt{C} = \mathtt{A} \#_{\mathtt{k}} (\mathtt{B} \#_{\mathtt{k}} \mathtt{C}).$$

It suffices to consider nonempty arrays. Write

$$\mathtt{A} = [\mathtt{a}|\ (\mathtt{s} : \mathtt{n})], \qquad \mathtt{B} = [\mathtt{b}|\ (\mathtt{t} : \mathtt{n})] \quad \text{and} \quad \mathtt{C} = [\mathtt{c}|\ (\mathtt{u} : \mathtt{n})].$$

In order to compute $A \#_k B$, it is required that

$$s_i = t_i \quad \text{for all} \quad i \neq k.$$

Hence,

$$B = [b \mid (s : n; s_k = t_k)].$$

By definition

$$A \#_k B = [ab \mid (s : n; s_k = s_k + t_k)]$$

where

$$ab(i : n) = \begin{cases} a(i : n) & \text{if } 1 \leq i_k \leq s_k \\ b(i : n; i_k = i_k - s_k) & \text{if } s_k < i_k \leq s_k + t_k \end{cases}.$$

In order to compute $(A \#_k B) \#_k C$, a necessary condition is that

$$s_i = u_i \quad \text{for all} \quad i \neq k.$$

Therefore,

$$C = [c \mid (s : n; s_k = u_k)].$$

By definition

$$(A \#_k B) \#_k C = [abc \mid (s : n; s_k = s_k + t_k + u_k)], \tag{16}$$

where

$$abc(i : n) = \begin{cases} ab(i : n) & \text{if } 1 \leq i_k \leq s_k + t_k \\ c(i : n; i_k = i_k - s_k - t_k) & \text{if } s_k + t_k < i_k \leq s_k + t_k + u_k \end{cases}$$

$$= \begin{cases} a(i : n) & \text{if } 1 \leq i_k \leq s_k \\ b(i : n; i_k = i_k - s_k) & \text{if } s_k < i_k \leq s_k + t_k \\ c(i : n; i_k = i_k - s_k - t_k) & \text{if } s_k + t_k < i_k \leq s_k + t_k + u_k \end{cases}. \tag{17}$$

In order to calculate $B \#_k C$,

$$t_i = u_i \quad \text{for all} \quad i \neq k.$$

Hence

$$C = [c \mid (t : n; t_k = u_k)],$$

and

$$B \#_k C = [bc \mid (t : n; t_k = t_k + u_k)],$$

where

$$bc(i : n) = \begin{cases} b(i : n) & \text{if } 1 \leq i_k \leq t_k \\ c(i : n; i_k - t_k) & \text{if } t_k < i_k \leq t_k + u_k \end{cases}.$$

In order to calculate $A \#_k (B \#_k C)$,

$$s_i = t_i \quad \text{for all} \quad i \neq k,$$

yielding

$$A \#_k (B \#_k C) = \left[\widehat{abc} \mid (s : n; s_k = s_k + t_k + u_k)\right], \tag{18}$$

where

$$\begin{aligned}
\widehat{abc}(i:n) &= \begin{cases} a(i:n) & \text{if } 1 \le i_k \le s_k \\ bc(i:n, i_k = i_k - s_k) & \text{if } s_k < i_k \le s_k + t_k + u_k \end{cases} \\
&= \begin{cases} a(i:n) & \text{if } 1 \le i_k \le s_k \\ bc(i:n, i_k = i_k - s_k) & \text{if } 1 \le i_k - s_k \le t_k + u_k \end{cases} \\
&= \begin{cases} a(i:n) & \text{if } 1 \le i_k \le s_k \\ b(i:n, i_k = i_k - s_k) & \text{if } 1 \le i_k - s_k \le t_k \\ c(i:n; i_k = i_k - s_k - t_k) & \text{if } t_k < i_k - s_k \le t_k + u_k \end{cases} \\
&= \begin{cases} a(i:n) & \text{if } 1 \le i_k \le s_k \\ b(i:n, i_k = i_k - s_k) & \text{if } s_k \le i_k \le s_k + t_k \\ c(i:n; i_k = i_k - s_k - t_k) & \text{if } s_k + t_k < i_k \le s_k + t_k + u_k \end{cases} \qquad (19)
\end{aligned}$$

By inspection of equations (16), (18), (17) and (19), conclude associativity holds. ■

THEOREM (GENERALIZED ASSOCIATIVITY) If for any suitable operands A, B and C, the operator $\oplus$ satisfies

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

then for any finite sequence $A_1, A_2, \ldots, A_m$ of such operands,

$$A_1 \oplus A_2 \oplus \cdots \oplus A_m \tag{20}$$

is defined and yields the same result for arbitrary groupings whenever all operations are pairwise defined.

PROOF Consider equation (4.11) and assume all operations are pairwise defined. The case $m = 3$ is true by assumption. Assume the theorem holds whenever $m \le p$ for some $p \ge 3$. Let $m = p + 1$. Calculate $A_i \oplus A_{i+1}$ and then perform all remaining operations in arbitrary order. By the inductive hypothesis, the result must always be the same, say B_i, since there are fewer than p operations after performing the first one. Now there are at most p different solutions. Plainly B_i and B_{i+2} must be equal since it is possible to group A_i with A_{i+1} and A_{i+2} with A_{i+3} when computing B_i or B_{i+2}. Hence there are at most two distinct solutions, namely B_1 and B_2, since all solutions B_i for which i is even must be equal, and symmetrically when i is odd. A similar argument works for groupings of three. Within any grouping of three, it is possible to group at will by the inductive hypothesis. Hence,

$$(A_1 \oplus A_2) \oplus A_3 \qquad \text{and} \qquad A_1 \oplus (A_2 \oplus A_3)$$

yield the same result. After forming such groupings, the result must be unique by the inductive hypothesis as the total number of operations is less than p if $m = p+1$. This forces $B_1 = B_2$, which means the solution is unique. By the principle of mathematical induction, this completes the proof. ■

THEOREM The recursive program

$$*\,f\,\square = \square,$$
$$*\,f\,(\cdot)_n(x) = (\cdot)_n(f(x)),$$
$$*\,f(A \mathbin{\#_k} B) = (*\,f\,A) \mathbin{\#_k} (*\,f\,B).$$

is equivalent to the definition

$$*\,f\,\square = \square,$$
$$*\,f\,[a\,|\,T] = [f \circ a\,|\,T].$$

PROOF By the unique extension property the recursive program must be equivalent to $*_k(*f)$. By Theorem 4.40

$$*_k(*f) = *f,$$

which completes the proof. ■

INDEX

$+\!\!+$, 17
—, 49
$\mathcal{A}^{k}_{Y}$, 55
ι, 34
K^{δ}, 92
$\mathbb{P}$, 36
$\asymp$, 80
$\smile$, 79
$\frown$, 80
$\circledast$, 78
$-\!\circ$, 78
$[\alpha]$, 17
$\bigsqcup$, 76
$\overline{C}^{n}_{k}$, 50
$\sim$, 45
$\langle\!\rangle_{k}$, 38
$\varnothing$, 3
$\square$, 47
$\mathcal{N}$, 35
$\mathcal{Z}$, 35
$\urcorner_{n}$, 50
$\rightarrow$, 5
$\mathbb{H}_{Y}$, 55
$*$, 75
$\sqsupset_{k}$, 37
$\mathcal{A}^{*}_{Y}$, 55
$\mathbb{A}^{n}$, 55
$\mathbb{A}_{Y}(T)$, 55
$\looparrowright$, 7
$\hookleftarrow$, 7
τ_{k}, 36
$\overleftrightarrow{\ominus}$, 79
$\overleftrightarrow{\oplus}$, 74

abstractors, 32
accumulation lemma, 20
adjacent, 44
aggregation, 14
argument, 6
arrangement, 14
array catamorphism, 117
array function, 44
array model, 44
array product, 161
array space, 55
arrows, 23
ax, 65
axes, 63
axes array, 85
axis, 51

base type, 4
biaxis, 78
biaxis identity, 134
bicomp, 78
bicomp distributivity, 134
binary function, 6
block, 67
block array, 68
BMF, 21
border, 83
border array, 82

Cartesian product, 6
catamorphism, 109
categorical data type, 22
cell, 86
CI, 100
closed, 10
codomain, 5
`col`, 91
collapse, 64
collapse rule, 144
column order, 48
combine, 76
composition, 9
concatenate, 59

concatenation, 59
concatenation identity, 100
concatenation operator, 59
conformable, 60
constant tuple, 38
constructor, 20, 32
contiguous subarray, 52
cover, 53, 86
cover array, 86

data structure, 13
data type, 4
delete, 38
destructor, 32, 61
`dice`, 91
direction, 51
disjoint, 53
distl, 149
domain, 5
DR, 98
drop, 73

element, 44
empty array, 47
empty set, 3
equality of `n`-arrays, 44
extended list homomorphism, 117

field direction, 51
`first`, 90
foldl, 19
foldr, 19
function, 5
function selector, 64
functional array, 44

generating relation, 20, 56
generator, 20, 56
group, 9

harray, 126
HC, 107
higher-order function, 7
HOF, 17
homomorphism, 8
homomorphism lemma, 18, 113
H[M]R, 100
hypercube, 45

hyperplane, 54
hyperplane concatenation, 107
hyperplane relabeling, 100

identity, 8
identity matrix, 92
ignoring the `k`-axis , 59
index, 44, 65
index generator, 34
index set, 36
indices, 44
infinite array, 82
infix form, 6
`init`, 90
`initial`, 91
injective, 6
instance, 43
integer array, 48
iota array, 50
isomorphism, 8
istr, 149

`k`-axis, 51
`k`-order, 48
Kronecker delta function, 92
Kronecker product, 160

LC, 106
left accumulate, 20
left array hom, 116
left fold, 110, 119
left list hom, 108
left order, 39
left reduce, 19
lexicographical order, 48
list catamorphism, 109
list concatenation, 106
list forms, 104
list function, 49
list partition, 71
location, 44
lower fold, 80

macro, 90
map, 17, 75, 111
map array, 127
map distributivity, 17
mapping, 5

mixed fold, 80
monoid, 8
morphism, 23
move right, 37
multiarray trick, 45, 215

n-array space, 55
n-dimensional tuple, 35
n-permutation, 81
n-tuple, 35
nested, 60
nesting, 14
neutral, 8
nodes, 44
nondecreasing, 48
nonempty array, 47
nonnegative array, 48
nonoverlapping, 53
nonscalar, 56

objects, 23
one-to-one, 6
operation, 5
ordered reduction, 129
orientation, 14
orthogonal functions, 32
outer product, 159
overlapping, 53

parameter, 6
partial function, 6
partition, 85
partition array, 85
partition pair, 85
pattern matching, 4
permute, 81
permuted reduction, 130
planar form, 105
planar map, 119
points, 44
polymorphic, 17
positive array, 48
power, 164
promotion laws, 19
promotion lemma, 110
proper subarray, 52

range, 5
RC, 101
reax_k, 58
reduce, 110, 119
reducing array, 127
repetition, 14
reshape, 155
reverse, 157
right array hom, 117
right fold, 110, 119
right list hom, 108
right order, 39
right reduce, 19
row order, 48

scalar array, 56
scalar function, 56
scalar identity, 101
scanl, 20
semigroup, 8
sequence, 34
sequential product, 36
shape, 62
shapesize, 155
shortest path problem, 161
SI, 101
singleton, 56
size, 62
slice, 69
smoothness, 14
split, 84
squared 2-array, 45
Squiggol, 21
strict subarray, 52
strictly increasing, 48
subarray, 52
submatrix, 52

tail, 91
take, 71
tau identities, 99
tau shift, 36
TI[I,M,N,P], 99
tile, 91
tiled array, 61, 138
tiling, 61, 138
TM, 99
total homomorphism, 131
total reduction, 131

`transpose`, 158
triad, 79
triad-zip identity, 134
tuple, 6
type, 47
type constructor, 17
type variable, 4

unary function, 6
unborder, 83
unique extension property, 109
universal set, 144
upper fold, 80

valency, 14

well-ordering, 14

zero array, 92
zip, 75

Dissertations in the Department of Computer Science

Rask, Raimo. Automating Estimation of Software Size During the Requirements Specification Phase — Application of Albrecht's Function Point Analysis Within Structured Methods. Joensuun yliopiston luonnontieteellisiä julkaisuja 28 — University of Joensuu. Publications in Sciences, 28. 128 p. + appendix. Joensuu, 1992.

Ahonen, Jarmo. Modeling Physical Domains for Knowledge Based Systems. Joensuun yliopiston luonnontieteellisiä julkaisuja 33 — University of Joensuu. Publications in Sciences, 33. 127 p. Joensuu, 1995.

Kopponen, Marja. CAI in CS. University of Joensuu, Computer Science, Dissertations 1. 97 p. Joensuu, 1997.

Forsell, Martti. Implementation of Instruction-Level and Thread-Level Parallelism in Computers, University of Joensuu, Computer Science, Dissertations 2. 121 p. Joensuu, 1997.

Juvaste, Simo. Modeling Parallel Shared Memory Computations, University of Joensuu, Computer Science, Dissertations 3. 190 p. Joensuu, 1998.

Ageenko, Eugene. Context-based Compression of Binary Images. University of Joensuu, Computer Science, Dissertations 4. 111 p. Joensuu, 2000.

Tukiainen, Markku. Developing a New Model of Spreadsheet Calculation: A Goals and Plans Approach. University of Joensuu, Computer Science, Dissertations 5. 151 p. Joensuu, 2001.