An Approach to Array Shape Determination in MATLAB

Pramod G. Joisha *U. Nagaraj Shenoy* *Prithviraj Banerjee*

Technical Report No. CPDC–TR–2000–10–010

October 2000

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering,
Technological Institute,
2145 Sheridan Road,
Northwestern University,
IL 60208–3118.

An Approach to Array Shape Determination in MATLAB*

Pramod G. Joisha U. Nagaraj Shenoy Prithviraj Banerjee

Center for Parallel and Distributed Computing, Electrical and Computer Engineering Department, Technological Institute, 2145 Sheridan Road, Northwestern University, IL 60208–3118.
Phone: (847) 467-4610, Fax: (847) 491-4455
Email: `[pjoisha, nagaraj, banerjee]@ece.nwu.edu`

Abstract

One of the main hurdles that array-based languages such as MATLAB and APL pose to compilation is the lack of an explicit declaration for an array's shape. In programs written using these languages, the attributes of intrinsic type and shape for a variable are implicitly characterized by the defining expression for that variable. In addition, these attributes are allowed to change on the fly. On account of these complications, and others such as the dynamic binding of storage to names, interpreters are typically used to cope with their translation.

In this report, we address the problem of statically inferring the shape of an array in languages such as MATLAB. Inferred shapes are desirable from the standpoint of both program compilation and efficient interpretation because static knowledge of an array's shape could permit reductions in the number of run-time array conformability checks, enable memory preallocation optimizations, and facilitate efficient translations to "scalar" target languages such as C.

This report presents a framework for statically describing the shape of a MATLAB expression, using a methodology based on systematic matrix formulations. The representation exposes the algebraic properties that underlie MATLAB's shape semantics and exactly captures the shape that the expression assumes at run time. Some of the highlights of this framework are its applicability to a large class of MATLAB functions and the uniformity of its approach. We compare our methods with the traditional shadow variable scheme and demonstrate how the algebraic view permits powerful shape-related assertions and optimizations not possible in the conventional approach.

Keywords array shape determination, statically unknown array shapes, dimensionality, shape tuples, shape-tuple functions, array operations, matrices

*This research was supported by DARPA under Contract F30602–98–2–0144.

1 Introduction

In languages such as MATLAB[b] that do not carry explicit type declarations, inferring an array's intrinsic type and shape become vital steps when the goal is to produce an efficient translation of the source in a target language that is statically typed and in which support exists only for elemental (i.e., scalar) operations.[c] Static knowledge of an array's intrinsic type and shape could also improve interpretive efficiency by enabling the system to avoid conformability checks on a function's operands at run time, if certain guarantees can be made on the nature of those operands at compile time. Besides, knowing how an array's shape will evolve during the course of a loop's execution may permit the system to arrive at some estimate of its size, thereby allowing the preallocation of the array outside the loop. This greatly enhances performance, since the overhead of incremental array growth is avoided.

In this work, we examine the problem of statically inferring an array's shape in the MATLAB programming language. The language is representative of numerous other interactive array languages such as APL and SETL, and was primarily chosen on account of the immense popularity that it enjoys in the programming community. In fact, the language's extensive array support, coupled with its simplicity and interactive nature, is the chief reason behind its emergence as the tool of choice for fast prototyping and analysis.

1.1 Motivation

Consider the synthetic MATLAB code fragment shown in Figure 1.[d]

```
m ← round(4*rand+1);
n ← round(5*rand+1);
x ← round(5*rand+1);
y ← round(6*rand+1);

a ← rand(m, n);
b ← rand(x, y);

c ← a*b;
d ← c+a;
e ← d-a;
```

FIGURE 1: A Motivating Code Fragment

Here, the invocations `rand(m, n)` and `rand(x, y)` return a pair of two-dimensional arrays (i.e., matrices) having the extents `m`, `x` and `n`, `y` along the first and second dimensions respectively. Thus, even though there is no way of establishing the values of `m`, `n`, `x` and `y` at compile time, we can still safely conclude at compile time that `a` and `b` have $\langle \mathtt{m}, \mathtt{n} \rangle$ and $\langle \mathtt{x}, \mathtt{y} \rangle$ as their respective shape tuples. However, what should the shape tuple corresponding to `c` be? Note that `c` is the outcome of `a*b` where `*` is the MATLAB matrix multiply operation [Mat97]. Therefore, the answer "$\langle \mathtt{m}, \mathtt{y} \rangle$, if $\mathtt{n} = \mathtt{x}$" is only partly correct. This is because, if either `a` or `b` evaluate to scalars at run time (i.e., $\mathtt{m} = 1 \wedge \mathtt{n} = 1$ or $\mathtt{x} = 1 \wedge \mathtt{y} = 1$), the shape of `c` will be $\langle \mathtt{x}, \mathtt{y} \rangle$ or $\langle \mathtt{m}, \mathtt{n} \rangle$ respectively. In fact, can we even determine the dimensionality of `c` at compile time? This is because, if either `a` or `b` are scalars at run time, `c` will have as many dimensions as the other operand. Since there is no "unique" shape tuple that can be ascribed to `c`, should we maintain (at compile time) a list of candidate

[b]MATLAB is a registered trademark of The MathWorks, Inc.

[c]The term type, in general, stands for a collection of data that have been grouped together for reasons of logical similarity. Logical similarity could occur at the structural level (e.g., a 2×3 matrix) as well as at an arithmetic level (e.g., integers, reals, complexes and so on). In MATLAB, these two aspects are independent of each other and can be considered in isolation; we call the former the attribute of *shape* and the latter the attribute of *intrinsic type*.

[d]The symbol ← will be used to denote an assignment in MATLAB.

shapes against c, each of which could potentially be the final shape of c? How then do we infer the shape of d in the given code excerpt so as to take into consideration all possible "reaching" shapes of c?

When the shape attribute of a variable is not statically determinable, researchers have usually approached the problem by generating code that performs the inference at execution time [DR96]. Though such an approach is robust, it does not offer an opportunity for propagating an expression's shape across statements. That is, after every assignment in which the shape of the right-hand side is not statically determinable, the defined variable's shape is expressed in terms of newly created temporaries called *shadow variables*. From then on, this is the only shape information pertaining to that variable which will be available to other expressions that use the variable. This shape information cannot be easily propagated because the shadow variables are conditionally assigned [DR96]. However, the ability to propagate an expression's shape could potentially lead to useful inferences; for example, in the preceding code fragment, it is possible to statically infer that if the assignment to d succeeds, the subsequent assignment to e will also succeed and that both e and d will then have exactly the same shape. The capacity to perform such inferences does not exist in the conventional shadow variable method.

1.2 Related Work

In the recent past, the compiler community has witnessed much activity in the area of compilation for the MATLAB language [FAL, QMSZ98, DJK93, CB98, MAJ, MCC, MAT]. The work due to Kaplan and Ullman [KU80], based on the theory of lattices, was among the first that dealt with the problem of automatically determining the type attributes in a programming language requiring no declarations. In the work due to Budd [Bud88], a partial ordering of intrinsic type and shape was used in the type determination process. Data-flow techniques were then applied to propagate type information across expressions, statements and procedures. However, the notion of shape as used in [Bud88] corresponded to the broad attributes of scalar, vector, "fixed-size" array and "arbitrary" array and it is not clear how the actual array extents were automatically computed. The FALCON project [DRP96, DR96] was among the early works to examine the problem of type determination in MATLAB. The FALCON system relies on a static shape inferring mechanism that essentially propagates an array's "rank" and shape when possible, and resorts to a dynamic strategy based on shadow variables otherwise. A similar technique is adopted in the "Otter" MATLAB compiler [QMSZ98] that aims at parallelizing a MATLAB source to an SPMD form. In other projects such as Menhir [CB98], a type propagation algorithm similar to the one proposed in [DR96] is used. In cases where the procedure lacks sufficient information to perform an inference, the Menhir system relies on user-provided annotations called directives for the necessary information. Other mathematical frameworks have been proposed [MP99b] that aid source-level optimizations of MATLAB programs by taking into advantage the semantic properties of the matrix operations involved. Investigations into shape, using alternate approaches such as category theory, have also been done [Jay95]; [Jay96] is a position paper that mentions the benefits of treating shape separately from data. These efforts have attempted to consider shape in a broad context—that is, as structures not just limited to matrices and arrays, but encompassing lists, trees and unlabelled graphs as well; the type of operations considered were confined to those that permitted a complete static analysis of shape—that is, operations in which the shape of the output was completely determined by the shapes of the inputs. (These were referred to as *shapely* operations.) An array-based language called FISh [JS98] was designed bottom-up as a culmination of these efforts.

1.2.1 Lattices for Type Estimation

In the Kaplan and Ullman approach, the existence of a lattice of types is postulated, chosen by the compiler designer. The objective is to determine statically a type that subsumes the actual run-time type as tightly as possible. The idea of a lattice is motivated by the fact that a type lower down in the lattice hierarchy can be safely replaced by a type higher up without compromising program correctness. By making the inference as close as possible to the run-time type, this approach attempts to optimize on the program's operational and storage requirements. The type estimation process in [KU78] begins with the most conservative estimates for the program variable types, which then get successively refined with every iteration of a forward and backward type inference pass. It has been shown that for lattices in which the *finite chain condition* holds, a finite number of such iterations produces a safe solution that cannot be improved upon further, and that is in some sense an optimal solution to the type determination problem [KU78]. Though the Kaplan and Ullman procedure can be carried over to MATLAB in a straightforward manner to solve the problem of intrinsic type estimation, the same cannot be said as far as shape determination is concerned. For the Kaplan and Ullman approach to work, the type functions that model the type semantics of the language's operators must be *monotonic* with respect to the defined lattice. For some of MATLAB's built-in functions such as matrix multiply, it can be shown that the shape-tuple function that models the operation's shape semantics will not be monotonic with respect to any lattice that can be defined on the set of shape tuples. Thus, existing lattice-based techniques have only limited scope for precise array shape determination in MATLAB.

1.2.2 FALCON Project

The main goals of the FALCON project were the efficient extraction of information from MATLAB's high-level semantics, the generation of high-performance code for serial and parallel architectures and the use of semantic information to facilitate the work of a parallelizing compiler [DR96]. Inference mechanisms based on static and dynamic methods were developed for the MATLAB language, along with value and symbolic propagation analyses to enhance the type estimation process. The goal of the FALCON inference system was to identify the intrinsic type, rank, shape and structure associated with every variable. The attribute of rank was meant to discriminate scalars, vectors and matrices, while the attribute of structure provided more specialized information such as whether a matrix was square, upper triangular and so on.

The outcome of a shape inference in [DR96] is an exact rank, an exclusive rank or an unknown rank. An exact rank refers to the case in which all of the array extents are known. An exclusive rank is assigned to a variable when the extent along only a particular dimension is known. Finally, a tag signifying an unknown rank is associated with those variables for which none of the array extents are statically determinable. The propagation of shape information across operators is then done by using tables that essentially encapsulate the operators' shape semantics. For variables having the unknown rank tag, shadow variables are generated at compile time to resolve the shape information at run time.

An important point that needs to be mentioned in the context of [DR96] is that all arrays were assumed to be two-dimensional. The current version of MATLAB however provides support for multidimensional arrays. Moreover, as discussed in § 1.1, even though the extents of a MATLAB expression may be unknown at compile time, useful inferences relating to shape that could be overlooked by the traditional shadow variable scheme may still be possible.

1.3 Contributions

This report presents a framework that makes it possible to statically describe the shape of a MATLAB expression. The main contribution is that, unlike previous approaches, the framework empowers useful inferences even in situations wherein the actual array extents may not be statically determinable. This difference is important because current techniques do not attempt further inferences from a statically unknown shape. The following are the specific contributions of this work.

- A framework that, in addition to enabling a compact and exact static representation of shape for a large class of MATLAB functions, reveals useful properties borne by the language's shape semantics.
- Two general taxonomies of MATLAB's functions are proposed to address the applicability of the framework. The taxonomies are useful because they identify classes of "shape-inference friendly" functions, and could therefore serve as guides when incorporating new functions into the language.
- We show how a compiler or interpreter could use the framework to reduce two overheads: that due to array conformability checks, and that due to the incremental growth of arrays in loops.

2 Preliminaries

All data in MATLAB is ultimately an array. For example, a scalar is an array of size 1×1. We use the shape-tuple notation $\langle p_1, p_2, \ldots, p_m \rangle$ to represent the shape of an m-dimensional array whose respective extents from the first to the mth dimension are p_1, p_2 and so on until p_m.

In MATLAB, any m-dimensional array can be considered to be an n-dimensional array, where $n > m$, simply by regarding the higher dimensions to have unit extents. Since higher dimensions are indicated to the right of lower dimensions in the shape-tuple notation, *trailing extents* of unity are effectively of no significance to an array's shape in MATLAB. In other words, the shape tuples $\langle 2, 3, 4 \rangle$, $\langle 2, 3, 4, 1 \rangle$, $\langle 2, 3, 4, 1, 1 \rangle$ and so on represent the same shape. We therefore say that these shape tuples are *MATLAB-equivalent.*

For the sake of convenience, we impose the restriction that the shape tuple of an array must have at least two components in its representation. With this proviso, a column vector with three elements could have any of the shape tuples $\langle 3, 1 \rangle$, $\langle 3, 1, 1 \rangle$ and so on, but not $\langle 3 \rangle$.

The notion of equivalent shape tuples leads to the idea of an array's *canonical shape tuple.* An array's canonical shape tuple is obtained from any of its equivalent shape tuples by discarding all trailing extents of unity from the third component onwards. For the column vector discussed above, the canonical shape tuple would be $\langle 3, 1 \rangle$ while the canonical shape tuple for a scalar would be $\langle 1, 1 \rangle$.

We next define an array's *rank* as the number of its dimensions.[e] Because an array will have an infinite set of shape tuples, it will also have an infinite set of ranks. For example, an array having $\langle 5, 1, 2 \rangle$ as its canonical shape tuple will have a rank of 3 or more. We therefore call the smallest rank that can be ascribed to an array its *canonical rank*; this equals the number of components in its canonical shape tuple.

2.1 Terminology

In the context of MATLAB expressions, we shall use the terms illegal arrays, scalars, row vectors, column vectors and matrices to mean the following:

illegal array: An array that is the "outcome" of an ill-formed MATLAB expression.

scalar: A legal array whose canonical rank is 2, and whose extents along the first and second dimensions are 1 each.

row vector: A legal array whose canonical rank is 2, and whose extent along the first dimension is 1.

column vector: A legal array whose canonical rank is 2, and whose extent along the second dimension is 1.

matrix: A legal array whose canonical rank is 2.

Illegal arrays are an artificial construct introduced only for completeness. They are meant to represent the result of an illegal MATLAB expression. For example, when a 2×3 matrix is multiplied with a 4×5 matrix in MATLAB, the run-time system will complain of an error. The concept of an illegal array is meant to abstract such error situations.

Notice the overlap in the above definitions. For instance, that which is a scalar could also be regarded as a row vector, a column vector or a matrix. And a row vector or a column vector is also a matrix. The "containment" in the above definitions is graphically shown in Figure 2. We shall use the phrase "higher dimensional array" to describe legal arrays whose canonical ranks are at least 3. These arrays lie in the

[e]The concept of an array's rank as used in this report is similar to that used in [DR96] and in the APL language, but is not the same. In [DR96], the inference mechanism refines an array's exact rank into either SCALAR, VECTOR or MATRIX; the proposed inference scheme does not explicitly address arrays having higher dimensions. De Rose's ranks of SCALAR, VECTOR and MATRIX correspond to a rank of 2 in our definition. In APL, the term rank is defined to be the number of components in the result of the monadic ρ primitive [PP75]. This definition is used to distinguish between scalars, vectors and arrays in the language. That is, datums having a rank of 0 are called scalars in APL. Vectors have a rank of 1, while arrays have higher ranks in the language.

unshaded region of Figure 2. The term “array” by itself (without any qualification) could mean an illegal array, a scalar, a row vector, a column vector, a matrix or a higher dimensional array.

2.2 Empty Arrays

MATLAB also supports *empty arrays* [Mat97]. These are legal arrays that contain no data but yet have a shape. For example,

```
>> c = rand(1, 0, 4)
c =
   Empty array: 1-by-0-by-4
```

assigns an empty array to the variable `c`. To quote [Mat97]:

> The basic model for empty matrices is that any operation that is defined for m-by-n matrices, and that produces a result whose dimension is some function of m and n, should still be allowed when m and n is zero. The size of the result should be that same function, evaluated at zero.

To encompass the empty array construct, we allow the shape-tuple components to also be zero. Thus, `c` in the last example is an empty array whose shape tuple is $\langle 1, 0, 4\rangle$.[f] In Figure 2, the crescent-shaped region represents these arrays.

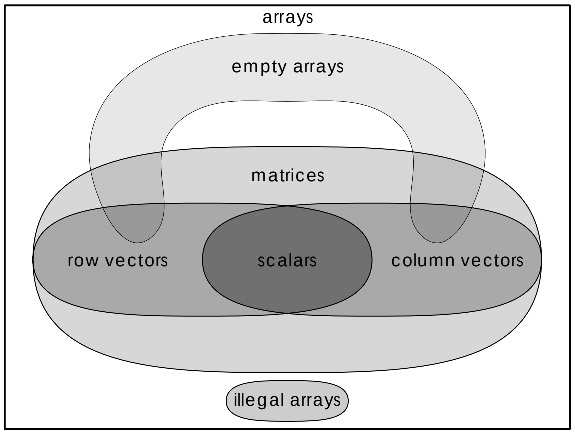

FIGURE 2: The Set of MATLAB Arrays $\mathbb{A}$

2.3 Shape Algebra Basics

Consider the set $\mathbb{L}_{\mathbb{S}}$ of all square diagonal matrices of order 2 or more, in which the principal diagonal elements belong to the set of nonnegative integers $\mathbb{W}$. We shall follow the convention of denoting an $n \times n$ square diagonal matrix having p_1, p_2 and so on until p_n as its principal diagonal elements by $\langle p_1, p_2, \ldots, p_n\rangle$. Thus,

$$\langle p_1, p_2, \ldots, p_n\rangle = \begin{pmatrix} p_1 & 0 & \ldots & 0 \\ 0 & p_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & p_n \end{pmatrix}.$$

By using $\langle p_1, p_2, \ldots, p_n\rangle$ to also represent the shape tuple of a MATLAB array, we in effect infuse the notation the power of matrix arithmetic. The choice of square diagonal matrices to capture the essence of an array’s

[f]An interesting consequence of supporting empty arrays is the ability to conjure up arrays out of nothing. For instance, if we multiply a $\langle 2, 0\rangle$ matrix with a $\langle 0, 3\rangle$ matrix, the product is a $\langle 2, 3\rangle$ matrix with all of its elements set to 0.

shape was motivated by the fact that under the usual matrix arithmetic operations of addition, subtraction, multiplication, division (i.e., inverse), and multiplication by a scalar, the result is also square diagonal.[g]

We additionally include the concept of "illegal shape tuples" so as to represent the shape of an illegal MATLAB array. We do this by considering a set $\mathbb{I}_{\mathbb{S}}$ of integer square diagonal matrices whose members do not belong to $\mathbb{L}_{\mathbb{S}}$. A suitable choice for $\mathbb{I}_{\mathbb{S}}$ would be:

$$\mathbb{I}_{\mathbb{S}} = \{\langle \pi_1, \pi_2 \rangle, \langle \pi_1, \pi_2, 1 \rangle, \langle \pi_1, \pi_2, 1, 1 \rangle, \ldots\} \tag{1}$$

where π_1 and π_2 are integers such that either $\pi_1 < 0$ or $\pi_2 < 0$ or both. Consider the augmented set $\mathbb{S} = \mathbb{L}_{\mathbb{S}} \cup \mathbb{I}_{\mathbb{S}}$. We can easily define an equivalence relation $\wp$ on $\mathbb{S}$ such that two elements in this set are related by $\wp$ if they are MATLAB-equivalent. That is, for any $\boldsymbol{s}, \boldsymbol{t} \in \mathbb{S}$, $\boldsymbol{s} \wp \boldsymbol{t}$ if and only if either $\boldsymbol{s}$ and $\boldsymbol{t}$ are identical or differ by trailing extents of unity from the third component on. Hence, if $\boldsymbol{s} = \langle p_1, p_2, \ldots, p_k \rangle$ and $\boldsymbol{t} = \langle q_1, q_2, \ldots, q_l \rangle$ where $k, l \geq 2$, then

$$\boldsymbol{s} \wp \boldsymbol{t} \Longrightarrow \begin{cases} \boldsymbol{s} = \boldsymbol{t} & \text{if } k = l, \\ \boldsymbol{s} = \langle q_1, q_2, \ldots, q_l, 1, \ldots, 1 \rangle & \text{if } k > l, \\ \boldsymbol{t} = \langle p_1, p_2, \ldots, p_k, 1, \ldots, 1 \rangle & \text{if } k < l. \end{cases} \tag{2}$$

Notice that the set of illegal shape tuples $\mathbb{I}_{\mathbb{S}}$ forms an equivalence class by this relation. Furthermore, observe that the shape tuple of a MATLAB expression can be any element in some equivalence class under $\wp$. Each equivalence class in the set of equivalence classes under $\wp$—called the *quotient set* of $\mathbb{S}$ by $\wp$ (see [TM75])—corresponds to a canonical shape tuple and vice versa.

3 Shape Inferring Framework

The shape inferring framework determines the shape tuple of a MATLAB expression, given the shape tuples of its operands. Every MATLAB function can have its shape semantics modelled algebraically by a *shape-tuple operator*. When applied to the shape tuples of the function's operands, the shape-tuple operator yields the shape tuple of the function's result.

To illustrate the actual mechanics of the shape inferring process, we shall consider the problem of determining the shape of a MATLAB matrix multiply expression. That is, given the MATLAB statement `c` $\leftarrow$ `a*b` where the shape tuples of `a` and `b` are $\boldsymbol{s} = \langle p_1, p_2, \ldots, p_k \rangle$ and $\boldsymbol{t} = \langle q_1, q_2, \ldots, q_l \rangle$ respectively and where $k, l \geq 2$, we shall see how the shape tuple $\boldsymbol{u} = \langle r_1, r_2, \ldots, r_m \rangle$ of the result `c` can be computed. We begin by reprising the shape semantics of MATLAB's matrix multiply operation [Mat97]:

> The function `*` is defined when one of the operands is a legal array and the other is a scalar. If both operands are nonscalars, then they must be matrices such that the extents along the second dimension of `a` and the first dimension of `b` match. Any other combination of shapes produces a run-time error.

The first question that needs to be addressed is what should the rank of the result `c` be. By answering this question, we would know the number of array extent components m in the shape tuple $\boldsymbol{u}$ of `c`. However, we cannot "accurately" answer this question at compile time in the sense that the canonical rank will, in the most general setting, be determinable only at run time. For instance, in the case of `c` $\leftarrow$ `a*b`, the canonical rank of `c` could be anywhere between 2 to $\max(k, l)$ depending on the run-time values of $p_1, p_2, \ldots, p_k$ and $q_1, q_2, \ldots, q_l$. Whatever may be the canonical shape tuple of the result, by virtue of the equivalence relation $\wp$ introduced in § 2.3, it will be equivalent to a shape tuple having $\max(k, l)$ (or more) components. Therefore, we can conservatively determine the rank of `c` at compile time as being

$$\mathcal{R}(\mathtt{c}) = \max(k, l). \tag{3}$$

[g]A row vector could have also been used to represent a shape tuple; however, square diagonal matrices were found to be a more convenient representation because certain quantities appear to be more easily expressible using them—for instance, the size of an array is simply the determinant of its associated shape tuple.

3.1 Shape Predicates

The next issue that needs to be addressed is detecting when a MATLAB matrix multiply operation is well defined. For this purpose, we enlist the services of three "shape-predicate" functions—θ, β and α—that map a shape tuple $\boldsymbol{s}$ to the set $\mathbb{B}$ consisting of the integers 0 and 1. These functions predicate three conditions that could be associated with a given shape tuple. The function $\theta : \mathbb{S} \mapsto \mathbb{B}$ is called the *correctness shape predicate* and maps all legal shape tuples to 1 and all illegal shape tuples to 0. If the shape tuple $\boldsymbol{s}$ indicates a MATLAB matrix, the *matrix shape predicate* $\beta : \mathbb{S} \mapsto \mathbb{B}$ is defined to be 1; otherwise it is 0. If $\boldsymbol{s}$ indicates a MATLAB scalar, the *scalar shape predicate* $\alpha : \mathbb{S} \mapsto \mathbb{B}$ is defined to be 1, and 0 otherwise. Note that the terminology of § 2.1 is used here.

From their definitions, each of the shape-predicate functions can be expressed mathematically in terms of the shape-tuple components. If $\boldsymbol{u} = \langle r_1, r_2, \ldots, r_m \rangle$, we have the following:

$$\beta(\boldsymbol{u}) = \theta(\boldsymbol{u})\delta(r_3 - 1)\delta(r_4 - 1)\ldots\delta(r_m - 1), \tag{4}$$

$$\alpha(\boldsymbol{u}) = \delta(r_1 - 1)\delta(r_2 - 1)\delta(r_3 - 1)\ldots\delta(r_m - 1). \tag{5}$$

In Equations (4) and (5), δ denotes the *discrete Delta function* defined on the integer domain [OS89]:

$$\delta(i) = \begin{cases} 0 & \text{if } i \neq 0, \\ 1 & \text{if } i = 0. \end{cases} \tag{6}$$

The way the θ function is connected to the shape-tuple components is dependent on the actual choice for the two-component illegal shape tuple $\boldsymbol{\pi} = \langle \pi_1, \pi_2 \rangle$ in Equation (1), and does not affect the formulation of our framework. Observe that by Equations (4) and (5), whenever $\beta(\boldsymbol{u})$ or $\alpha(\boldsymbol{u})$ is 1, $\theta(\boldsymbol{u})$ must also be 1.

Getting back to the MATLAB statement `c ← a*b`, the correctness shape predicate $\theta(\boldsymbol{u})$ should be 1 if the MATLAB expression `a*b` is well formed, and 0 otherwise. When is a MATLAB matrix multiply well defined? According to the earlier stated semantics, the outcome of `a*b` is a legal array so long as `a` and `b` are both legal, and either `a` is a scalar, or `b` is a scalar, or `a` and `b` are matrices such that the extent of `a` along its second dimension equals the extent of `b` along its first dimension. Couching these semantics in mathematical language, we get

$$\theta(\boldsymbol{u}) = \theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1 - (1 - \alpha(\boldsymbol{s}))(1 - \alpha(\boldsymbol{t}))(1 - \beta(\boldsymbol{s})\beta(\boldsymbol{t})\delta(p_2 - q_1))\big). \tag{7}$$

It is easy to verify that Equation (7) evaluates to 1 for a well-defined MATLAB matrix multiply operation, and to 0 otherwise. For instance, if `a` were a scalar and `b` a legal array, $\theta(\boldsymbol{s})$, $\theta(\boldsymbol{t})$ and $\alpha(\boldsymbol{s})$ would all become 1, so that $\theta(\boldsymbol{u})$ would simplify to 1, irrespective of what $\beta(\boldsymbol{t})$, p_2 and q_1 actually are. We therefore say that a scalar shape tuple and any legal shape tuple *always* form a "legal combo" for the `*` built-in function.

3.2 Shape Tuple

To formulate the shape tuple of the result, we take advantage of the fact that the shape tuple representation synonymously denotes a square diagonal matrix. This allows us to algebraically calculate the shape tuple of the result using elementary matrix arithmetic on the shape tuples of the operands. In the case of `c ← a*b`, we get

$$\begin{aligned}\boldsymbol{u} = (1 - \theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\big(&\boldsymbol{s}^*\alpha(\boldsymbol{t}) + \boldsymbol{t}^*\alpha(\boldsymbol{s})(1 - \alpha(\boldsymbol{t})) \\ &+ (\boldsymbol{s}^*\boldsymbol{\Gamma_1} + \boldsymbol{t}^*\boldsymbol{\Gamma_2} + \boldsymbol{\iota}^* - \boldsymbol{\Gamma_1} - \boldsymbol{\Gamma_2})(1 - \alpha(\boldsymbol{s}))(1 - \alpha(\boldsymbol{t}))\big),\end{aligned} \tag{8}$$

which, from Equation (7), expands to

$$\begin{aligned}\boldsymbol{u} = &\Big(1 - \theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1 - (1 - \alpha(\boldsymbol{s}))(1 - \alpha(\boldsymbol{t}))(1 - \beta(\boldsymbol{s})\beta(\boldsymbol{t})\delta(p_2 - q_1))\big)\Big)\boldsymbol{\pi}^* \\ &+ \theta(\boldsymbol{s})\theta(\boldsymbol{t})\Big(1 - (1 - \alpha(\boldsymbol{s}))(1 - \alpha(\boldsymbol{t}))(1 - \beta(\boldsymbol{s})\beta(\boldsymbol{t})\delta(p_2 - q_1))\Big)\Big(\boldsymbol{s}^*\alpha(\boldsymbol{t}) + \boldsymbol{t}^*\alpha(\boldsymbol{s})(1 - \alpha(\boldsymbol{t})) \\ &+ (\boldsymbol{s}^*\boldsymbol{\Gamma_1} + \boldsymbol{t}^*\boldsymbol{\Gamma_2} + \boldsymbol{\iota}^* - \boldsymbol{\Gamma_1} - \boldsymbol{\Gamma_2})(1 - \alpha(\boldsymbol{s}))(1 - \alpha(\boldsymbol{t}))\Big).\end{aligned}$$

In the above equation, each of the quantities $\boldsymbol{\pi}^*$, $\boldsymbol{s}^*$, $\boldsymbol{t}^*$, $\boldsymbol{\iota}^*$, $\boldsymbol{\Gamma_1}$ and $\boldsymbol{\Gamma_2}$ designate $\mathcal{R}(\mathtt{c}) \times \mathcal{R}(\mathtt{c}) = \max(k,l) \times \max(k,l)$ integer square diagonal matrices. In $\boldsymbol{\Gamma_1}$, only the first principal diagonal element is 1 and the rest are 0. In $\boldsymbol{\Gamma_2}$, only the second principal diagonal element is 1 and the remaining are 0. The symbols $\boldsymbol{\pi}^*$ and $\boldsymbol{\iota}^*$ respectively represent the two-component illegal shape tuple $\boldsymbol{\pi} = \langle \pi_1, \pi_2 \rangle$ and the two-component scalar shape tuple $\boldsymbol{\iota} = \langle 1, 1 \rangle$, appropriately "promoted" to $\mathcal{R}(\mathtt{c})$ components by appending unit extents. The quantities $\boldsymbol{s}^*$ and $\boldsymbol{t}^*$ are also obtained by promoting $\boldsymbol{s}$ and $\boldsymbol{t}$ to $\mathcal{R}(\mathtt{c})$ components. By having all the matrices in Equation (8) to be of the same size (i.e., $\mathcal{R}(\mathtt{c}) \times \mathcal{R}(\mathtt{c})$), the computation in the equation is well defined. Using the shape tuple notation, each of the matrices in Equation (8) can be represented as shown below

$$
\begin{aligned}
\boldsymbol{\pi}^* &= \langle \pi_1, \pi_2, \overbrace{1, \dots, 1}^{\mathcal{R}(\mathtt{c})-2 \text{ components}} \rangle, \\
\boldsymbol{s}^* &= \langle \boldsymbol{s}, \overbrace{1, \dots, 1}^{\mathcal{R}(\mathtt{c})-k \text{ components}} \rangle, \\
\boldsymbol{t}^* &= \langle \boldsymbol{t}, \overbrace{1, \dots, 1}^{\mathcal{R}(\mathtt{c})-l \text{ components}} \rangle, \\
\boldsymbol{\iota}^* &= \langle 1, 1, \overbrace{1, \dots, 1}^{\mathcal{R}(\mathtt{c})-2 \text{ components}} \rangle, \\
\boldsymbol{\Gamma_1} &= \langle 1, 0, \overbrace{0, \dots, 0}^{\mathcal{R}(\mathtt{c})-2 \text{ components}} \rangle, \\
\boldsymbol{\Gamma_2} &= \langle 0, 1, \overbrace{0, \dots, 0}^{\mathcal{R}(\mathtt{c})-2 \text{ components}} \rangle.
\end{aligned}
$$

where, $\langle \boldsymbol{s}, 1, \dots, 1 \rangle$ and $\langle \boldsymbol{t}, 1, \dots, 1 \rangle$ are shorthands for $\langle p_1, p_2, \dots, p_k, 1, \dots, 1 \rangle$ and $\langle q_1, q_2, \dots, q_l, 1, \dots, 1 \rangle$ respectively.

In general, the promotion of a shape tuple $\boldsymbol{s} = \langle p_1, p_2, \dots, p_k \rangle$ to m components may correspond to either an expansion or a truncation to m components. The former occurs if $m \geq k$ and results in an expanded version $\boldsymbol{s}^*$ obtained by appending zero or more unit extents to $\boldsymbol{s}$. The latter occurs if $m < k$ and is produced by retaining only the first m components in $\boldsymbol{s}$. Examples where m may be less than k are discussed in § 7.1 and § 7.2.

Example 1: *Matrix Multiplication*

Let us reconsider the previous MATLAB statement `c` $\leftarrow$ `a*b`, and suppose that the shape tuples associated with `a` and `b` are $\langle p_1, p_2 \rangle$ and $\langle q_1, q_2, q_3 \rangle$ respectively. Therefore,

$$
k = 2, \qquad l = 3,
$$
$$
\boldsymbol{s} = \begin{pmatrix} p_1 & 0 \\ 0 & p_2 \end{pmatrix}, \qquad \boldsymbol{t} = \begin{pmatrix} q_1 & 0 & 0 \\ 0 & q_2 & 0 \\ 0 & 0 & q_3 \end{pmatrix}.
$$

From Equation (3) and after promoting the shape tuples $\boldsymbol{s}$ and $\boldsymbol{t}$ to $\mathcal{R}(\mathtt{c})$ components, we get

$$
\begin{aligned}
\mathcal{R}(\mathtt{c}) &= 3, \\
\boldsymbol{s}^* &= \begin{pmatrix} p_1 & 0 & 0 \\ 0 & p_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \\
\boldsymbol{t}^* &= \begin{pmatrix} q_1 & 0 & 0 \\ 0 & q_2 & 0 \\ 0 & 0 & q_3 \end{pmatrix}.
\end{aligned}
$$

From Equation (4), we have $\beta(\boldsymbol{s}) = \theta(\boldsymbol{s})$ and $\beta(\boldsymbol{t}) = \theta(\boldsymbol{t})\delta(q_3 - 1)$. From Equation (5), we also have $\alpha(\boldsymbol{s}) = \delta(p_1 - 1)\delta(p_2 - 1)$ and $\alpha(\boldsymbol{t}) = \delta(q_1 - 1)\delta(q_2 - 1)\delta(q_3 - 1)$. These values can be plugged into Equation (7) to obtain the correctness shape predicate $\theta(\boldsymbol{u})$:

$$\theta(\boldsymbol{u}) = \theta(\boldsymbol{s})\theta(\boldsymbol{t})\Big(1-\big(1-\delta(p_1-1)\delta(p_2-1)\big)\big(1-\delta(q_1-1)\delta(q_2-1)\delta(q_3-1)\big)\big(1-\theta(\boldsymbol{s})\theta(\boldsymbol{t})\delta(q_3-1)\delta(p_2-q_1)\big)\Big).$$

Hence, from Equation (8), we get the shape tuple $\boldsymbol{u}$ of `c` to be

$$\boldsymbol{u} = \begin{pmatrix} \begin{array}{l}\theta(\boldsymbol{u})\big(p_1B+q_1A(1-B)\\ +p_1(1-A)(1-B)\big)+(1-\theta(\boldsymbol{u}))\pi_1\end{array} & 0 & 0 \\ 0 & \begin{array}{l}\theta(\boldsymbol{u})\big(p_2B+q_2A(1-B)\\ +q_2(1-A)(1-B)\big)+(1-\theta(\boldsymbol{u}))\pi_2\end{array} & 0 \\ 0 & 0 & \begin{array}{l}\theta(\boldsymbol{u})\big(B+q_3A(1-B)\\ +(1-A)(1-B)\big)+(1-\theta(\boldsymbol{u}))\end{array} \end{pmatrix},$$

where $A = \alpha(\boldsymbol{s})$, $B = \alpha(\boldsymbol{t})$ and $\boldsymbol{\pi}^* = \langle \pi_1, \pi_2, 1\rangle$. Thus, if the respective values for $\langle p_1, p_2\rangle$ and $\langle q_1, q_2, q_3\rangle$ are, say $\langle 3, 2\rangle$ and $\langle 4, 4, 1\rangle$ at run time, $\theta(\boldsymbol{u})$ will become 0, giving $\boldsymbol{\pi}^*$ for $\boldsymbol{u}$. The key point is that we now have a compact static representation for the shape tuple of `c` that takes into account all possibilities. ■

4 MATLAB Matrix Multiply Revisited

The right-hand side of Equation (8) is essentially a linear sum of four terms:

$$\begin{aligned}&(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^*,\\ &\theta(\boldsymbol{u})\boldsymbol{s}^*\alpha(\boldsymbol{t}),\\ &\theta(\boldsymbol{u})\boldsymbol{t}^*\alpha(\boldsymbol{s})(1-\alpha(\boldsymbol{t})),\\ &\theta(\boldsymbol{u})(\boldsymbol{s}^*\boldsymbol{\Gamma_1}+\boldsymbol{t}^*\boldsymbol{\Gamma_2}+\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1}-\boldsymbol{\Gamma_2})(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t})).\end{aligned}$$

It is easy to see that at any one time, only one of these four terms contributes to the sum. For example, for an illegal shape-tuple combo, $\theta(\boldsymbol{u})$ will be 0 so that only the first term contributes to the sum. When $\theta(\boldsymbol{u})$ is 1—which indicates a legal shape-tuple combo—only three possibilities are allowed by virtue of Equation (7): either `b` is scalar, or `a` is a scalar, or `a` and `b` are `*`-conforming nonscalars (i.e., `a` and `b` are nonscalar matrices such that the extent of `a` along the second dimension equals the extent of `b` along the first dimension). If `b` is a scalar, then whatever `a` may be, only the second term contributes to the sum. However, if `b` is not a scalar and `a` is, only the third term contributes to the sum. The remaining case of `a` and `b` being `*`-conforming nonscalars results in the fourth term contributing to the sum. Since the individual contributions (i.e., $\boldsymbol{\pi}^*$, $\boldsymbol{s}^*$, $\boldsymbol{t}^*$ and $\boldsymbol{s}^*\boldsymbol{\Gamma_1}+\boldsymbol{t}^*\boldsymbol{\Gamma_2}+\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1}-\boldsymbol{\Gamma_2}$) are all clearly members of $\mathbb{S}$, Equation (8) defines a mapping $\ddot{\circledast}$ from $\mathbb{S}\times\mathbb{S}$ to $\mathbb{S}$. Thus, $[\mathbb{S}, \ddot{\circledast}]$ forms an *algebraic system* [TM75]. We therefore call $\ddot{\circledast}$ as the shape-tuple operator or the shape-tuple function for the MATLAB matrix multiply built-in function.

An important question that crops up at this juncture is the following: If one of the shape-tuple operands to $\ddot{\circledast}$ were replaced by an equivalent shape tuple, how would the result change? Lemma 4.1 answers this question by showing that the new and old results would still be equivalent to each other.

Lemma 4.1: For any $\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u} \in \mathbb{S}$, if $\boldsymbol{s} \wp \boldsymbol{t}$, then $(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s}) \wp (\boldsymbol{u}\ddot{\circledast}\boldsymbol{t})$ and $(\boldsymbol{s}\ddot{\circledast}\boldsymbol{u}) \wp (\boldsymbol{t}\ddot{\circledast}\boldsymbol{u})$.

Proof.
From Equation (2), $\boldsymbol{s} \wp \boldsymbol{t}$ could imply one of three things: $\boldsymbol{s} = \boldsymbol{t}$, $\boldsymbol{s} = \langle \boldsymbol{t}, 1, \ldots, 1\rangle$ or $\boldsymbol{t} = \langle \boldsymbol{s}, 1, \ldots, 1\rangle$. If $\boldsymbol{s} = \boldsymbol{t}$, the claim of the lemma trivially holds. Suppose $\boldsymbol{s} \neq \boldsymbol{t}$. Since $\wp$ is symmetric, we can assume without loss of generality that $\boldsymbol{t} = \langle \boldsymbol{s}, 1, \ldots, 1\rangle$. Because $\boldsymbol{t}$ would then differ from $\boldsymbol{s}$ by only trailing unit extents from the third component on, $\boldsymbol{t}$ would be legal or illegal accordingly as $\boldsymbol{s}$ is legal or illegal. Therefore, $\theta(\boldsymbol{t}) = \theta(\boldsymbol{s})$. In addition, we have the following from Equations (4) and (5):

$$\beta(\boldsymbol{t}) = \beta(\boldsymbol{s}), \tag{4.1.1}$$

$$\alpha(\boldsymbol{t}) = \alpha(\boldsymbol{s}). \tag{4.1.2}$$

From Equation (7), we also have

$$\begin{aligned}\theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s}) &= \theta(\boldsymbol{u})\theta(\boldsymbol{s})\big(1-(1-\alpha(\boldsymbol{u}))(1-\alpha(\boldsymbol{s}))(1-\beta(\boldsymbol{u})\beta(\boldsymbol{s})\delta(u_2-s_1))\big),\\ \theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{t}) &= \theta(\boldsymbol{u})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{u}))(1-\alpha(\boldsymbol{t}))(1-\beta(\boldsymbol{u})\beta(\boldsymbol{t})\delta(u_2-t_1))\big),\end{aligned}$$

where $\boldsymbol{s}=\langle s_1,s_2,\ldots,s_k\rangle$, $\boldsymbol{t}=\langle t_1,t_2,\ldots,t_l\rangle$ and $\boldsymbol{u}=\langle u_1,u_2,\ldots,u_m\rangle$. By substituting $\beta(\boldsymbol{t})$ and $\alpha(\boldsymbol{t})$ from Equations (4.1.1) and (4.1.2) into the last equation and because $\theta(\boldsymbol{t})=\theta(\boldsymbol{s})$ and $t_1=s_1$, we get

$$\theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{t})=\theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s}). \tag{4.1.3}$$

Now from Equation (8), we have

$$\begin{aligned}\boldsymbol{u}\ddot{\circledast}\boldsymbol{s} = (1-\theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s})\big(&\boldsymbol{u}^*\alpha(\boldsymbol{s})+\boldsymbol{s}^*\alpha(\boldsymbol{u})(1-\alpha(\boldsymbol{s}))\\ &+(\boldsymbol{u}^*\boldsymbol{\Gamma_1}+\boldsymbol{s}^*\boldsymbol{\Gamma_2}+\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1}-\boldsymbol{\Gamma_2})(1-\alpha(\boldsymbol{u}))(1-\alpha(\boldsymbol{s}))\big),\end{aligned} \tag{4.1.4}$$

$$\begin{aligned}\boldsymbol{u}\ddot{\circledast}\boldsymbol{t} = (1-\theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{t}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{t})\big(&\boldsymbol{u}^*\alpha(\boldsymbol{t})+\boldsymbol{t}^*\alpha(\boldsymbol{u})(1-\alpha(\boldsymbol{t}))\\ &+(\boldsymbol{u}^*\boldsymbol{\Gamma_1}+\boldsymbol{t}^*\boldsymbol{\Gamma_2}+\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1}-\boldsymbol{\Gamma_2})(1-\alpha(\boldsymbol{u}))(1-\alpha(\boldsymbol{t}))\big).\end{aligned} \tag{4.1.5}$$

Using Equations (4.1.2) and (4.1.3), the last equation simplifies to

$$\begin{aligned}\boldsymbol{u}\ddot{\circledast}\boldsymbol{t} = (1-\theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s})\big(&\boldsymbol{u}^*\alpha(\boldsymbol{s})+\boldsymbol{t}^*\alpha(\boldsymbol{u})(1-\alpha(\boldsymbol{s}))\\ &+(\boldsymbol{u}^*\boldsymbol{\Gamma_1}+\boldsymbol{t}^*\boldsymbol{\Gamma_2}+\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1}-\boldsymbol{\Gamma_2})(1-\alpha(\boldsymbol{u}))(1-\alpha(\boldsymbol{s}))\big).\end{aligned} \tag{4.1.6}$$

Since $\boldsymbol{t}=\langle\boldsymbol{s},1,\ldots,1\rangle$, Equation (4.1.6) can be rewritten as

$$\begin{aligned}\boldsymbol{u}\ddot{\circledast}\boldsymbol{t} = (1-\theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s})\big(&\boldsymbol{u}^*\alpha(\boldsymbol{s})+\boldsymbol{s}^*\alpha(\boldsymbol{u})(1-\alpha(\boldsymbol{s}))\\ &+(\boldsymbol{u}^*\boldsymbol{\Gamma_1}+\boldsymbol{s}^*\boldsymbol{\Gamma_2}+\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1}-\boldsymbol{\Gamma_2})(1-\alpha(\boldsymbol{u}))(1-\alpha(\boldsymbol{s}))\big).\end{aligned} \tag{4.1.7}$$

Note that the terms $\boldsymbol{\pi}^*$, $\boldsymbol{u}^*$, $\boldsymbol{s}^*$ and $\boldsymbol{\iota}^*$ in Equations (4.1.4) and (4.1.7) are not the same: While they refer to promoted versions of $\boldsymbol{\pi}$, $\boldsymbol{u}$, $\boldsymbol{s}$ and $\boldsymbol{\iota}$ respectively in both cases, the promotion is to $\max(m,k)$ components in the former case and $\max(m,l)$ components in the latter case. Therefore, even though the terms $\boldsymbol{u}^*\boldsymbol{\Gamma_1}+\boldsymbol{s}^*\boldsymbol{\Gamma_2}+\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1}-\boldsymbol{\Gamma_2}$ may not be the same in the two equations, they will still be equivalent under the relation $\wp$. Hence, by comparing Equations (4.1.4) and (4.1.7), we can conclude that $(\boldsymbol{u}\ddot{\circledast}\boldsymbol{s})\,\wp(\boldsymbol{u}\ddot{\circledast}\boldsymbol{t})$. We can similarly show that if $\boldsymbol{s}\wp\boldsymbol{t}$, then $(\boldsymbol{s}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{u})$ is also true.

4.1 The Substitution Property

Let $[X,\bullet]$ be an algebraic system in which $\bullet$ is a binary operation. An equivalence relation E on X is said to have the *substitution property* with respect to the operation $\bullet$ if for any $x_1,x_2,x_1',x_2'\in X$, $(x_1Ex_1')\wedge(x_2Ex_2')$ implies that $(x_1\bullet x_2)E(x_1'\bullet x_2')$ [TM75]. The property shown in Lemma 4.1 can be regarded as a "one-sided" version of the substitution property. Using this as a starting point, it can be shown that $\wp$ also has the substitution property with respect to $\ddot{\circledast}$.

Lemma 4.2: For any $\boldsymbol{s},\boldsymbol{t},\boldsymbol{u},\boldsymbol{v}\in\mathbb{S}$, if $(\boldsymbol{s}\wp\boldsymbol{t})\wedge(\boldsymbol{u}\wp\boldsymbol{v})$, then $(\boldsymbol{s}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{v})$.

Proof.
From Lemma 4.1, we have

$$\boldsymbol{s}\wp\boldsymbol{t}\implies(\boldsymbol{s}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{u}), \tag{4.2.1}$$

$$\boldsymbol{u}\wp\boldsymbol{v}\implies(\boldsymbol{t}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{v}). \tag{4.2.2}$$

But since $\wp$ is transitive,

$$\big((\boldsymbol{s}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{u})\big)\wedge\big((\boldsymbol{t}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{v})\big)\implies(\boldsymbol{s}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{v}).$$

Therefore,

$$(\boldsymbol{s}\wp\boldsymbol{t})\wedge(\boldsymbol{u}\wp\boldsymbol{v})\implies\big((\boldsymbol{s}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{u})\big)\wedge\big((\boldsymbol{t}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{v})\big)\implies(\boldsymbol{s}\ddot{\circledast}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\circledast}\boldsymbol{v}).$$

An important consequence of the substitution property is that it does not matter which among the equivalent shape tuples is chosen while computing Equation (8). We are guaranteed to always arrive at results that will at worst differ only by trailing extents of unity.

The fact that $\wp$ satisfies the substitution property with respect to $\ddot{\circledast}$ should not be surprising. This is because, as far as MATLAB is concerned, only the canonical form of a shape tuple matters in its shape semantics. For example, the shape semantics for the `*` built-in function stated that at least one of the operands must be a scalar (i.e., has the canonical shape tuple $\langle 1, 1\rangle$) or both must be nonscalar matrices having the canonical shape tuples $\langle p, q\rangle$ and $\langle q, r\rangle$. The shape-predicate functions basically flag each of these "canonical-form situations," and are incorporated in the shape-tuple expression so as to single out a shape tuple that represents the shape of the result. Consequently, the shape-predicate functions will not change in value when the operand shape tuples are replaced by equivalent shape tuples—they will continue to single out the same term (or perhaps one in an equivalent form) in the result's shape-tuple expression.

Equivalence relations such as $\wp$ that satisfy the substitution property with respect to some algebraic system are usually called *congruence relations* [TM75]. Such relations enable the construction of new and simpler algebraic systems from a given algebraic system. For example, in the case of $[\mathbb{S}, \ddot{\circledast}]$, the $\ddot{\circledast}$ operation suggests the simpler operation $\dot{\circledast} : \mathbb{S}_\wp \times \mathbb{S}_\wp \mapsto \mathbb{S}_\wp$ that works directly on the quotient set $\mathbb{S}_\wp$ of $\mathbb{S}$ by $\wp$.

4.2 A Simpler Algebra

Given a shape tuple $\boldsymbol{s}$, let $\overline{\boldsymbol{s}}$ denote its equivalence class in $\mathbb{S}_\wp$. Consider the binary operation $\dot{\circledast}$ defined on the set of equivalence classes $\mathbb{S}_\wp$ such that

$$\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{t}} = \overline{\boldsymbol{s}\ddot{\circledast}\boldsymbol{t}}, \tag{9}$$

where $\boldsymbol{s}, \boldsymbol{t} \in \mathbb{S}$. It is easy to see that $\dot{\circledast}$ is well defined on $\mathbb{S}_\wp$ because $\overline{\boldsymbol{s}\ddot{\circledast}\boldsymbol{t}}$ is independent of the elements $\boldsymbol{s}$ and $\boldsymbol{t}$ chosen to represent the equivalence classes $\overline{\boldsymbol{s}}$ and $\overline{\boldsymbol{t}}$. We shall refer to operators such as $\dot{\circledast}$ as shape-tuple class functions or shape-tuple class operators.

An alternate way of looking at the shape-tuple class operator $\dot{\circledast}$ is to view the equivalence classes $\overline{\boldsymbol{s}}$ and $\overline{\boldsymbol{t}}$ as representations of the canonical shape tuples associated with $\boldsymbol{s}$ and $\boldsymbol{t}$ respectively, and to then regard $\dot{\circledast}$ as the canonical shape-tuple operator for the MATLAB matrix multiply built-in function.

Algebraic systems such as $[\mathbb{S}_\wp, \dot{\circledast}]$ are called *quotient algebras* [TM75]. They preserve many of the properties of the parent algebras from which they are derived. Because these algebras operate on equivalence classes, "relationship properties" seen in the parent algebra become "equality properties" in the quotient algebra. For example, as Lemma 4.3 will show, $(\boldsymbol{s}\ddot{\circledast}\boldsymbol{\pi})\,\wp\,\boldsymbol{\pi}$ for all $\boldsymbol{s} \in \mathbb{S}$. In the quotient algebra, this property becomes $\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{\pi}} = \overline{\boldsymbol{\pi}}$.

Lemma 4.3: For all $\boldsymbol{s} \in \mathbb{S}$, $\overline{\boldsymbol{\pi}}\dot{\circledast}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{\pi}} = \overline{\boldsymbol{\pi}}$.

Proof.
By definition of the θ function, $\theta(\boldsymbol{\pi}) = 0$. Hence, for all $\boldsymbol{s} \in \mathbb{S}$,

$$\theta(\boldsymbol{\pi}\ddot{\circledast}\boldsymbol{s}) = \theta(\boldsymbol{\pi})\theta(\boldsymbol{s})\big(1 - (1 - \alpha(\boldsymbol{\pi}))(1 - \alpha(\boldsymbol{s}))(1 - \beta(\boldsymbol{\pi})\beta(\boldsymbol{s})\delta(\pi_2 - s_1))\big) = 0,$$
$$\theta(\boldsymbol{s}\ddot{\circledast}\boldsymbol{\pi}) = \theta(\boldsymbol{s})\theta(\boldsymbol{\pi})\big(1 - (1 - \alpha(\boldsymbol{s}))(1 - \alpha(\boldsymbol{\pi}))(1 - \beta(\boldsymbol{s})\beta(\boldsymbol{\pi})\delta(s_2 - \pi_1))\big) = 0,$$

where $\boldsymbol{s} = \langle s_1, s_2, \ldots, s_k\rangle$ and where $k \geq 2$. Thus, from Equation (8),

$$(\boldsymbol{\pi}\ddot{\circledast}\boldsymbol{s})\,\wp\,\boldsymbol{\pi},$$
$$(\boldsymbol{s}\ddot{\circledast}\boldsymbol{\pi})\,\wp\,\boldsymbol{\pi}.$$

Hence,

$$\overline{\boldsymbol{\pi}\ddot{\circledast}\boldsymbol{s}} = \overline{\boldsymbol{s}\ddot{\circledast}\boldsymbol{\pi}} = \overline{\boldsymbol{\pi}}.$$

But from Equation (9), $\overline{\boldsymbol{\pi}\ddot{\circledast}\boldsymbol{s}} = \overline{\boldsymbol{\pi}}\dot{\circledast}\overline{\boldsymbol{s}}$ and $\overline{\boldsymbol{s}\ddot{\circledast}\boldsymbol{\pi}} = \overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{\pi}}$. Hence, for all $\boldsymbol{s} \in \mathbb{S}$,

$$\overline{\boldsymbol{\pi}}\dot{\circledast}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{\pi}} = \overline{\boldsymbol{\pi}}.$$

Lemma 4.3 is important because it tells us that the equivalence class of illegal shape tuples $\overline{\boldsymbol{\pi}}$ forms the *zero element* or the *annihilator* of $[\mathbb{S}_\wp, \dot{\circledast}]$. An element $\omega \in X$ is called the annihilator of the algebraic system $[X, \bullet]$ if $\omega \bullet x = x \bullet \omega = \omega$ for all $x \in X$ [CLR95].

Additionally, the equivalence class of scalar shape tuples $\overline{\boldsymbol{\iota}}$ forms the *identity element* of $[\mathbb{S}_\wp, \dot{\circledast}]$. An element $\omega \in X$ is called the identity element of the algebraic system $[X, \bullet]$ if $\omega \bullet x = x \bullet \omega = x$ for all $x \in X$ [TM75]. If an annihilator or an identity exist for an algebra, it can be easily shown that they will be unique.

Lemma 4.4: For all $\boldsymbol{s} \in \mathbb{S}$, $\overline{\boldsymbol{s}} \dot{\circledast} \overline{\boldsymbol{\iota}} = \overline{\boldsymbol{\iota}} \dot{\circledast} \overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}$.

Proof.
If $\overline{\boldsymbol{s}}$ is the illegal shape-tuple class $\overline{\boldsymbol{\pi}}$, the claim trivially holds from Lemma 4.3. Therefore, assume that $\overline{\boldsymbol{s}} \neq \overline{\boldsymbol{\pi}}$. Hence,

$$\theta(\boldsymbol{s}) = 1.$$

We also have $\theta(\boldsymbol{\iota}) = 1$ and $\alpha(\boldsymbol{\iota}) = 1$. This implies that

$$\theta(\boldsymbol{s} \ddot{\circledast} \boldsymbol{\iota}) = \theta(\boldsymbol{\iota} \ddot{\circledast} \boldsymbol{s}) = 1.$$

From Equation (8), we therefore get $\boldsymbol{s} \ddot{\circledast} \boldsymbol{\iota} = \boldsymbol{s}^*$ and $\boldsymbol{\iota} \ddot{\circledast} \boldsymbol{s} = \boldsymbol{\iota}^* \alpha(\boldsymbol{s}) + \boldsymbol{s}^*(1 - \alpha(\boldsymbol{s})) = \boldsymbol{s}^*$. Hence,

$$(\boldsymbol{\iota} \ddot{\circledast} \boldsymbol{s}) \wp \boldsymbol{s},$$
$$(\boldsymbol{s} \ddot{\circledast} \boldsymbol{\iota}) \wp \boldsymbol{s}.$$

Therefore,

$$\overline{\boldsymbol{s}} \dot{\circledast} \overline{\boldsymbol{\iota}} = \overline{\boldsymbol{s} \ddot{\circledast} \boldsymbol{\iota}} = \overline{\boldsymbol{s}},$$
$$\overline{\boldsymbol{\iota}} \dot{\circledast} \overline{\boldsymbol{s}} = \overline{\boldsymbol{\iota} \ddot{\circledast} \boldsymbol{s}} = \overline{\boldsymbol{s}}.$$

5 MATLAB Array Addition

To demonstrate how the inferring formalism can be applied to another MATLAB built-in function, we consider the array addition operation [Mat97]:

```
c ← a+b;
```

The result `c` is defined only when either operand is a scalar and the other a legal array, or when both operands are legal arrays having identical shapes. This enables us to once again conservatively determine the rank of `c` at compile time as being

$$\mathcal{R}(\texttt{c}) = \max(k, l), \tag{10}$$

where k and l are the ranks of `a` and `b` respectively. As before, let $\boldsymbol{s} = \langle p_1, p_2, \ldots, p_k \rangle$ be the shape tuple of `a` and $\boldsymbol{t} = \langle q_1, q_2, \ldots, q_l \rangle$ be that of `b`. We can translate MATLAB's requirement for array addition conformance to the following correctness shape predicate:

$$\theta(\boldsymbol{u}) = \begin{cases} \theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1 - (1 - \alpha(\boldsymbol{s}))(1 - \alpha(\boldsymbol{t}))(1 - \delta(p_1 - q_1) \ldots \delta(p_k - q_k))\big) & \text{if } k = l, \\ \theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1 - (1 - \alpha(\boldsymbol{s}))(1 - \alpha(\boldsymbol{t}))(1 - \delta(p_1 - q_1) \ldots \delta(p_k - q_k)\delta(1 - q_{k+1}) \ldots \delta(1 - q_l))\big) & \text{if } k < l, \\ \theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1 - (1 - \alpha(\boldsymbol{s}))(1 - \alpha(\boldsymbol{t}))(1 - \delta(p_1 - q_1) \ldots \delta(p_l - q_l)\delta(p_{l+1} - 1) \ldots \delta(p_k - 1))\big) & \text{if } k > l, \end{cases} \tag{11}$$

where $\boldsymbol{u}$ is the shape tuple of `c`. To compactly express the previous equation, we extend the definition of the discrete Delta function given in Equation (6) to integer square diagonal matrices:

$$\delta(\langle r_1, r_2, \ldots, r_m \rangle) = \delta(r_1)\delta(r_2) \ldots \delta(r_m), \tag{12}$$

where r_1, r_2 and so on until r_m are integers. Thus, Equation (12) simplifies Equation (11) to

$$\theta(\boldsymbol{u}) = \theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t}))(1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*))\big), \tag{13}$$

where $\boldsymbol{s}^*$ and $\boldsymbol{t}^*$ are obtained by promoting $\boldsymbol{s}$ and $\boldsymbol{t}$ respectively to $\max(\mathcal{R}(\mathtt{a}),\mathcal{R}(\mathtt{b})) = \max(k,l)$ components. This promotion occurs as before—that is, by appending the necessary number of unit extents to the shape tuples so as to make all of the matrices involved of size $\mathcal{R}(\mathtt{c})\times\mathcal{R}(\mathtt{c})$. By using $\boldsymbol{\pi}^*$, $\boldsymbol{s}^*$, $\boldsymbol{t}^*$, and the correctness and scalar shape predicates, we can also translate MATLAB's array addition shape semantics to formulate the shape tuple $\boldsymbol{u}$ of the result:

$$\boldsymbol{u} = (1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\big(\boldsymbol{s}^*\alpha(\boldsymbol{t}) + \boldsymbol{t}^*(1-\alpha(\boldsymbol{t}))\big). \tag{14}$$

5.1 An Algebra

The right-hand side of Equation (14) is a linear sum of three terms:

$$\begin{aligned}&(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^*,\\&\theta(\boldsymbol{u})\boldsymbol{s}^*\alpha(\boldsymbol{t}),\\&\theta(\boldsymbol{u})\boldsymbol{t}^*(1-\alpha(\boldsymbol{t})).\end{aligned}$$

At any one time, only one of these three terms contributes to the sum. For an illegal shape-tuple combo, $\theta(\boldsymbol{u})$ will be 0 so that only the first term contributes to the sum. For a legal shape-tuple combo, $\theta(\boldsymbol{u})$ will be 1 so that the result is either $\boldsymbol{s}^*$ or $\boldsymbol{t}^*$. Thus clearly, Equation (14) defines a mapping $\ddot{\oplus}$ from $\mathbb{S}\times\mathbb{S}$ to $\mathbb{S}$ implying that $[\mathbb{S},\ddot{\oplus}]$ is an algebraic system.

All of the properties derived in § 4 for the matrix multiply operation also hold for the array addition operation. We begin by showing that the equivalence relation $\wp$ also has the substitution property with respect to $\ddot{\oplus}$. This enables us to consider the simplified algebra $[\mathbb{S}_\wp,\dot{\oplus}]$ in which $\overline{\boldsymbol{\pi}}$ and $\overline{\boldsymbol{\iota}}$ are respectively the annihilator and identity.

Lemma 5.1: For any $\boldsymbol{s},\boldsymbol{t},\boldsymbol{u}\in\mathbb{S}$, if $\boldsymbol{s}\,\wp\,\boldsymbol{t}$, then $(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s})\,\wp(\boldsymbol{u}\ddot{\oplus}\boldsymbol{t})$ and $(\boldsymbol{s}\ddot{\oplus}\boldsymbol{u})\,\wp(\boldsymbol{t}\ddot{\oplus}\boldsymbol{u})$.

Proof.
If $\boldsymbol{s}=\boldsymbol{t}$, the claim of the lemma trivially holds. Suppose $\boldsymbol{s}\neq\boldsymbol{t}$. Because $\wp$ is symmetric, we can assume without loss of generality that $\boldsymbol{t}=\langle\boldsymbol{s},1,\ldots,1\rangle$. Thus,

$$\theta(\boldsymbol{t}) = \theta(\boldsymbol{s}), \tag{5.1.1}$$
$$\alpha(\boldsymbol{t}) = \alpha(\boldsymbol{s}). \tag{5.1.2}$$

From Equation (13), we also have

$$\begin{aligned}\theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s}) &= \theta(\boldsymbol{u})\theta(\boldsymbol{s})\big(1-(1-\alpha(\boldsymbol{u}))(1-\alpha(\boldsymbol{s}))(1-\delta(\boldsymbol{u}^*-\boldsymbol{s}^*))\big),\\ \theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{t}) &= \theta(\boldsymbol{u})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{u}))(1-\alpha(\boldsymbol{t}))(1-\delta(\boldsymbol{u}^*-\boldsymbol{t}^*))\big).\end{aligned}$$

By substituting $\theta(\boldsymbol{t})$ and $\alpha(\boldsymbol{t})$ from Equations (5.1.1) and (5.1.2) into the last equation and because $\delta(\boldsymbol{u}^*-\boldsymbol{t}^*) = \delta(\boldsymbol{u}^*-\langle\boldsymbol{s},1,\ldots,1\rangle) = \delta(\boldsymbol{u}^*-\boldsymbol{s}^*)$, we get

$$\theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{t}) = \theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s}). \tag{5.1.3}$$

We now compute the shape tuples $\boldsymbol{u}\ddot{\oplus}\boldsymbol{s}$ and $\boldsymbol{u}\ddot{\oplus}\boldsymbol{t}$ using Equation (14):

$$\boldsymbol{u}\ddot{\oplus}\boldsymbol{s} = (1-\theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s})\big(\boldsymbol{u}^*\alpha(\boldsymbol{s}) + \boldsymbol{s}^*(1-\alpha(\boldsymbol{s}))\big), \tag{5.1.4}$$
$$\boldsymbol{u}\ddot{\oplus}\boldsymbol{t} = (1-\theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{t}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{t})\big(\boldsymbol{u}^*\alpha(\boldsymbol{t}) + \boldsymbol{t}^*(1-\alpha(\boldsymbol{t}))\big). \tag{5.1.5}$$

From Equations (5.1.2) and (5.1.3), the last equation simplifies to

$$\boldsymbol{u}\ddot{\oplus}\boldsymbol{t} = (1-\theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s})\big(\boldsymbol{u}^*\alpha(\boldsymbol{s}) + \boldsymbol{t}^*(1-\alpha(\boldsymbol{s}))\big). \tag{5.1.6}$$

Since $\boldsymbol{t} = \langle \boldsymbol{s}, 1, \ldots, 1\rangle$, Equation (5.1.6) can be rewritten as

$$\boldsymbol{u}\ddot{\oplus}\boldsymbol{t} = (1-\theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s})\big(\boldsymbol{u}^*\alpha(\boldsymbol{s}) + \boldsymbol{s}^*(1-\alpha(\boldsymbol{s}))\big). \tag{5.1.7}$$

Hence, by comparing Equations (5.1.4) and (5.1.7), we can conclude that $(\boldsymbol{u}\ddot{\oplus}\boldsymbol{s}) \wp (\boldsymbol{u}\ddot{\oplus}\boldsymbol{t})$. Similarly, $(\boldsymbol{s}\ddot{\oplus}\boldsymbol{u}) \wp (\boldsymbol{t}\ddot{\oplus}\boldsymbol{u})$ if $\boldsymbol{s} \wp \boldsymbol{t}$.

The one-sided substitution property shown in Lemma 5.1 can be used to prove that $\wp$ also satisfies the substitution property with respect to $\dot{\oplus}$. The style of proof is similar to that shown in Lemma 4.2. In fact, if the one-sided substitution property holds for an equivalence relation with respect to some binary operation, then the substitution property follows, and vice versa.

Lemma 5.2: For all $\boldsymbol{s} \in \mathbb{S}$, $\overline{\boldsymbol{\pi}}\dot{\oplus}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{\pi}} = \overline{\boldsymbol{\pi}}$.

Proof.
We have $\theta(\boldsymbol{\pi}) = 0$. Hence

$$\theta(\boldsymbol{\pi}\ddot{\oplus}\boldsymbol{s}) = \theta(\boldsymbol{\pi})\theta(\boldsymbol{s})\big(1-(1-\alpha(\boldsymbol{\pi}))(1-\alpha(\boldsymbol{s}))(1-\delta(\boldsymbol{\pi}^*-\boldsymbol{s}^*))\big) = 0,$$
$$\theta(\boldsymbol{s}\ddot{\oplus}\boldsymbol{\pi}) = \theta(\boldsymbol{s})\theta(\boldsymbol{\pi})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{\pi}))(1-\delta(\boldsymbol{s}^*-\boldsymbol{\pi}^*))\big) = 0.$$

Thus, from Equation (14),

$$(\boldsymbol{\pi}\ddot{\oplus}\boldsymbol{s}) \wp \boldsymbol{\pi},$$
$$(\boldsymbol{s}\ddot{\oplus}\boldsymbol{\pi}) \wp \boldsymbol{\pi}.$$

Hence, for all $\boldsymbol{s} \in \mathbb{S}$,

$$\overline{\boldsymbol{\pi}}\dot{\oplus}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{\pi}} = \overline{\boldsymbol{\pi}}.$$

Lemma 5.3: For all $\boldsymbol{s} \in \mathbb{S}$, $\overline{\boldsymbol{\iota}}\dot{\oplus}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{\iota}} = \overline{\boldsymbol{s}}$.

Proof.
If $\overline{\boldsymbol{s}}$ is the illegal shape-tuple class $\overline{\boldsymbol{\pi}}$, the claim trivially follows from Lemma 5.2. Therefore, assume that $\boldsymbol{s}$ is legal. Hence, $\theta(\boldsymbol{s}) = 1$ and from Equation (13), this implies that

$$\theta(\boldsymbol{s}\ddot{\oplus}\boldsymbol{\iota}) = \theta(\boldsymbol{\iota}\ddot{\oplus}\boldsymbol{s}) = 1.$$

From Equation (14), we thus get $\boldsymbol{s}\ddot{\oplus}\boldsymbol{\iota} = \boldsymbol{s}^*$ and $\boldsymbol{\iota}\ddot{\oplus}\boldsymbol{s} = \boldsymbol{\iota}^*\alpha(\boldsymbol{s}) + \boldsymbol{s}^*(1-\alpha(\boldsymbol{s})) = \boldsymbol{s}^*$. Hence,

$$\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{\iota}} = \overline{\boldsymbol{\iota}}\dot{\oplus}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}.$$

5.2 An Idempotent, Commutative Monoid

The algebraic system $[\mathbb{S}_\wp, \dot{\oplus}]$ can be shown to form a special discrete structure called a *semigroup*. An algebra $[X, \bullet]$ is called a semigroup if the binary operation $\bullet$ is associative: For any $x, y, z \in X$, $(x \bullet y) \bullet z = x \bullet (y \bullet z)$.

Lemma 5.4: For all $\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u} \in \mathbb{S}$, $(\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} = \overline{\boldsymbol{s}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}})$.

Proof.
We prove this lemma by considering three cases:

- At least one shape tuple among $\boldsymbol{s}$, $\boldsymbol{t}$ and $\boldsymbol{u}$ is illegal.
  Suppose $\boldsymbol{s}$ is an illegal shape tuple. Then from Lemma 5.2,
  $$\begin{aligned}(\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} &= (\overline{\boldsymbol{\pi}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} = \overline{\boldsymbol{\pi}}\dot{\oplus}\overline{\boldsymbol{u}} = \overline{\boldsymbol{\pi}},\\ \overline{\boldsymbol{s}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}}) &= \overline{\boldsymbol{\pi}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}}) = \overline{\boldsymbol{\pi}}.\end{aligned}$$
  Therefore,
  $$(\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} = \overline{\boldsymbol{s}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}}).$$
  The same holds if either $\overline{\boldsymbol{t}} = \overline{\boldsymbol{\pi}}$ or $\overline{\boldsymbol{u}} = \overline{\boldsymbol{\pi}}$.
- At least one among the legal shape tuples $\boldsymbol{s}$, $\boldsymbol{t}$ and $\boldsymbol{u}$ represents a scalar.
  Suppose $\overline{\boldsymbol{s}} = \overline{\boldsymbol{\iota}}$. Then from Lemma 5.3,
  $$\begin{aligned}(\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} &= (\overline{\boldsymbol{\iota}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} = \overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}},\\ \overline{\boldsymbol{s}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}}) &= \overline{\boldsymbol{\iota}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}}) = \overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}}.\end{aligned}$$
  Therefore,
  $$(\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} = \overline{\boldsymbol{s}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}}).$$
  The same holds if either $\overline{\boldsymbol{t}} = \overline{\boldsymbol{\iota}}$ or $\overline{\boldsymbol{u}} = \overline{\boldsymbol{\iota}}$.
- None of the legal shape tuples $\boldsymbol{s}$, $\boldsymbol{t}$ and $\boldsymbol{u}$ represent a scalar.
  From Equation (13), $\theta(\boldsymbol{s}\ddot{\oplus}\boldsymbol{t}) = \delta(\boldsymbol{s}^* - \boldsymbol{t}^*)$. Therefore,
  $$\boldsymbol{s}\ddot{\oplus}\boldsymbol{t} = (1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*))\boldsymbol{\pi}^* + \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\boldsymbol{t}^*. \tag{5.4.1}$$
  Hence,
  $$\begin{aligned}\alpha(\boldsymbol{s}\ddot{\oplus}\boldsymbol{t}) &= \alpha\big((1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*))\boldsymbol{\pi}^* + \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\boldsymbol{t}^*\big),\\ &= (1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*))\alpha(\boldsymbol{\pi}^*) + \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\alpha(\boldsymbol{t}^*),\\ &= 0. \end{aligned}\tag{5.4.2}$$
  Now $\theta((\boldsymbol{s}\ddot{\oplus}\boldsymbol{t})\ddot{\oplus}\boldsymbol{u}) = \theta(\boldsymbol{z})\theta(\boldsymbol{u})\big(1-(1-\alpha(\boldsymbol{z}))(1-\alpha(\boldsymbol{u}))(1-\delta(\boldsymbol{z}^*-\boldsymbol{u}^*))\big)$ where $\boldsymbol{z} = \boldsymbol{s}\ddot{\oplus}\boldsymbol{t}$. Simplifying this, we get
  $$\begin{aligned}\theta((\boldsymbol{s}\ddot{\oplus}\boldsymbol{t})\ddot{\oplus}\boldsymbol{u}) &= \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\delta(\boldsymbol{z}^*-\boldsymbol{u}^*)\\ &= \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\delta\big((1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*))\boldsymbol{\pi}^* + \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\boldsymbol{t}^* - \boldsymbol{u}^*\big)\\ &= \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\delta\big((1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*))\boldsymbol{\pi}^* + \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\boldsymbol{t}^* - (1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*) + \delta(\boldsymbol{s}^*-\boldsymbol{t}^*))\boldsymbol{u}^*\big)\\ &= \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\delta\big((1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*))(\boldsymbol{\pi}^*-\boldsymbol{u}^*) + \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)(\boldsymbol{t}^*-\boldsymbol{u}^*)\big)\\ &= \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\big((1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*))\delta(\boldsymbol{\pi}^*-\boldsymbol{u}^*) + \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\delta(\boldsymbol{t}^*-\boldsymbol{u}^*)\big)\\ &= \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\delta(\boldsymbol{t}^*-\boldsymbol{u}^*).\end{aligned}\tag{5.4.3}$$
  Substituting Equation (5.4.3) into Equation (14), we thus obtain
  $$(\boldsymbol{s}\ddot{\oplus}\boldsymbol{t})\ddot{\oplus}\boldsymbol{u} = (1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\delta(\boldsymbol{t}^*-\boldsymbol{u}^*))\boldsymbol{\pi}^* + \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\delta(\boldsymbol{t}^*-\boldsymbol{u}^*)\boldsymbol{u}^*. \tag{5.4.4}$$
  We can similarly show that
  $$\boldsymbol{s}\ddot{\oplus}(\boldsymbol{t}\ddot{\oplus}\boldsymbol{u}) = (1-\delta(\boldsymbol{t}^*-\boldsymbol{u}^*)\delta(\boldsymbol{s}^*-\boldsymbol{u}^*))\boldsymbol{\pi}^* + \delta(\boldsymbol{t}^*-\boldsymbol{u}^*)\delta(\boldsymbol{s}^*-\boldsymbol{u}^*)\boldsymbol{u}^*. \tag{5.4.5}$$
  But $\delta(\boldsymbol{t}^*-\boldsymbol{u}^*)\delta(\boldsymbol{s}^*-\boldsymbol{u}^*) = \delta(\boldsymbol{s}^*-\boldsymbol{t}^*)\delta(\boldsymbol{t}^*-\boldsymbol{u}^*)$. Therefore, from Equations (5.4.4) and (5.4.5), $\overline{(\boldsymbol{s}\ddot{\oplus}\boldsymbol{t})\ddot{\oplus}\boldsymbol{u}} = \overline{\boldsymbol{s}\ddot{\oplus}(\boldsymbol{t}\ddot{\oplus}\boldsymbol{u})}$. Hence,
  $$(\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} = \overline{\boldsymbol{s}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}}).$$

Thus, $(\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} = \overline{\boldsymbol{s}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}})$ is true for all $\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u} \in \mathbb{S}$.

Because of the associativity and identity properties, $[\mathbb{S}_{\wp}, \dot{\oplus}]$ forms an algebraic structure called a *monoid* [TM75]. In addition, $\dot{\oplus}$ also exhibits the property of *commutativity* [TM75]: Lemma 5.5 proves this key result.

Lemma 5.5: For all $\boldsymbol{s}, \boldsymbol{t} \in \mathbb{S}$, $\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}} = \overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{s}}$.

Proof.
If either $\boldsymbol{s}$ or $\boldsymbol{t}$ is the illegal shape tuple or the scalar shape tuple, the result follows trivially from Lemmas 5.2 and 5.3. Hence, suppose that both $\boldsymbol{s}$ and $\boldsymbol{t}$ are legal shape tuples, neither of which represent a scalar. Therefore, from Equation (13),

$$\theta(\boldsymbol{s}\ddot{\oplus}\boldsymbol{t}) = \delta(\boldsymbol{s}^* - \boldsymbol{t}^*), \tag{5.5.1}$$
$$\theta(\boldsymbol{t}\ddot{\oplus}\boldsymbol{s}) = \delta(\boldsymbol{t}^* - \boldsymbol{s}^*). \tag{5.5.2}$$

Thus,

$$\boldsymbol{s}\ddot{\oplus}\boldsymbol{t} = (1 - \delta(\boldsymbol{s}^* - \boldsymbol{t}^*))\boldsymbol{\pi}^* + \delta(\boldsymbol{s}^* - \boldsymbol{t}^*)\boldsymbol{t}^*, \tag{5.5.3}$$
$$\boldsymbol{t}\ddot{\oplus}\boldsymbol{s} = (1 - \delta(\boldsymbol{t}^* - \boldsymbol{s}^*))\boldsymbol{\pi}^* + \delta(\boldsymbol{t}^* - \boldsymbol{s}^*)\boldsymbol{s}^*. \tag{5.5.4}$$

But

$$\delta(\boldsymbol{s}^* - \boldsymbol{t}^*) = \delta(\boldsymbol{t}^* - \boldsymbol{s}^*).$$

Therefore, $\delta(\boldsymbol{s}^* - \boldsymbol{t}^*)\boldsymbol{t}^* = \delta(\boldsymbol{t}^* - \boldsymbol{s}^*)\boldsymbol{s}^*$. Hence, from Equation (5.5.3) and Equation (5.5.4) we get

$$(\boldsymbol{s}\ddot{\oplus}\boldsymbol{t}) \wp (\boldsymbol{t}\ddot{\oplus}\boldsymbol{s}).$$

and the claim of the lemma follows.

The last property that we prove is the *idempotent law* [TM75]. An algebraic system $[X, \bullet]$ is said to satisfy the idempotent law if for all $x \in X$, $x \bullet x = x$.

Lemma 5.6: For all $s \in \mathbb{S}$, $\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}$.

Proof.
From Equation (13),

$$\theta(\boldsymbol{s}\ddot{\oplus}\boldsymbol{s}) = \theta(\boldsymbol{s}).$$

Therefore,

$$\begin{aligned} \boldsymbol{s}\ddot{\oplus}\boldsymbol{s} &= (1 - \theta(\boldsymbol{s}))\boldsymbol{\pi}^* + \theta(\boldsymbol{s})\big(\boldsymbol{s}^*\alpha(\boldsymbol{s}) + \boldsymbol{s}^*(1 - \alpha(\boldsymbol{s}))\big) \\ &= (1 - \theta(\boldsymbol{s}))\boldsymbol{\pi}^* + \theta(\boldsymbol{s})\boldsymbol{s}^*. \end{aligned} \tag{5.6.1}$$

From Equation (5.6.1), if $\boldsymbol{s} \wp \boldsymbol{\pi}$, $(\boldsymbol{s}\ddot{\oplus}\boldsymbol{s}) \wp \boldsymbol{\pi}$, and if $\theta(\boldsymbol{s}) = 0$, $(\boldsymbol{s}\ddot{\oplus}\boldsymbol{s}) \wp \boldsymbol{s}$. Hence, either way,

$$\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}.$$

5.3 A Note on $[\mathbb{S}_\wp, \dot{\circledast}]$

It should be mentioned here that associativity, commutativity and the idempotent law *do not* hold for $[\mathbb{S}_\wp, \dot{\circledast}]$:

- **Associativity**: Consider $(\langle 2,5,3\rangle \ddot{\circledast} \langle 1,2\rangle)\ddot{\circledast}\langle 2,1\rangle$ and $\langle 2,5,3\rangle\ddot{\circledast}(\langle 1,2\rangle\ddot{\circledast}\langle 2,1\rangle)$. From Equation (8),

$$\big((\langle 2,5,3\rangle\ddot{\circledast}\langle 1,2\rangle)\ddot{\circledast}\langle 2,1\rangle\big)\ \wp(\boldsymbol{\pi}\ddot{\circledast}\langle 2,1\rangle),$$
$$\big(\langle 2,5,3\rangle\ddot{\circledast}(\langle 1,2\rangle\ddot{\circledast}\langle 2,1\rangle)\big)\ \wp(\langle 2,5,3\rangle\ddot{\circledast}\langle 1,1\rangle).$$

Therefore,
$$(\overline{\langle 2,5,3\rangle}\dot{\circledast}\overline{\langle 1,2\rangle})\dot{\circledast}\overline{\langle 2,1\rangle} \neq \overline{\langle 2,5,3\rangle}\dot{\circledast}(\overline{\langle 1,2\rangle}\dot{\circledast}\overline{\langle 2,1\rangle}).$$

- **Commutativity**: Consider $\langle 2,5\rangle\ddot{\circledast}\langle 5,2\rangle$. Therefore,

$$(\langle 2,5\rangle\ddot{\circledast}\langle 5,2\rangle)\ \wp\langle 2,2\rangle,$$
$$(\langle 5,2\rangle\ddot{\circledast}\langle 2,5\rangle)\ \wp\langle 5,5\rangle.$$

Hence,
$$\overline{\langle 2,5\rangle}\dot{\circledast}\overline{\langle 5,2\rangle} \neq \overline{\langle 5,2\rangle}\dot{\circledast}\overline{\langle 2,5\rangle}.$$

- **Idempotent Law**: Consider $\langle 2,5,3\rangle\ddot{\circledast}\langle 2,5,3\rangle$. We have

$$(\langle 2,5,3\rangle\ddot{\circledast}\langle 2,5,3\rangle)\ \wp\,\boldsymbol{\pi}.$$

Therefore,
$$\overline{\langle 2,5,3\rangle}\dot{\circledast}\overline{\langle 2,5,3\rangle} \neq \overline{\langle 2,5,3\rangle}.$$

5.4 MATLAB's Element-wise Operators

In MATLAB, there exists an important class of built-in operations called the *element-wise* functions to which the array addition operator `+` belongs. These operators have exactly the same shape semantics as `+`, and therefore form *isomorphic* [TM75] shape-tuple algebras. Hence, the algebraic system $[\mathbb{S}_\wp, \dot{\oplus}]$ can be used to describe the shape-tuple algebras of all of these element-wise functions. Some examples of element-wise built-in functions are the left array divide operator (`.\`), the array power operator (`.^`) and the logical AND operator (`&`) [Mat97]. For a larger list, see Table 2.

6 Built-in Functions

From a shape inferring perspective, it suffices to focus attention on only those language operators that are built directly into the MATLAB system. These operators, known as *built-in functions*, are similar to the primitives in APL, and ultimately comprise all MATLAB programs. Once we know how shape inferring works for each of these functions, the hope is to determine the shapes of arbitrary MATLAB expressions by applying program-wide shape determination techniques.

6.1 Taxonomies

To appreciate when the shape inferring framework is applicable, we consider two taxonomies—one on the basis of shape, and the other on the basis of rank. The shape-based classification groups value-returning built-in functions into three categories:

- **Type I:** $\mathbb{S} \times \mathbb{S} \times \ldots \mathbb{S} \mapsto \mathbb{S}$.

  These built-in functions produce values whose shapes are completely known once the shapes of the arguments, if any, are known. Examples of such built-in functions are the matrix multiply operator discussed in § 3 and § 4, and the element-wise operators mentioned in § 5 and § 5.4.

- **Type II:** $\mathbb{S} \times \mathbb{S} \times \ldots \mathbb{S} \times \mathbb{A} \times \mathbb{A} \ldots \mathbb{A} \mapsto \mathbb{S}$.

  These are built-in functions that do not belong to Type I, and that produce values whose shapes are completely known only when the *value* of at least one argument is known. An example of a built-in function that belongs to this category is the colon operator (`:`) [Mat97]. For instance, given the construction

  ```
  c ← a:b;
  ```

  `c` will be a row vector consisting of $\lfloor b' - a' \rfloor + 1$ elements where a' and b' represent the run-time *real values* of `a` and `b` respectively. If $a' > b'$, `c` is the empty row vector. Thus, the shape of the result `c` is determinable only when the values of `a` and `b` are known. This built-in function is not a Type I operator because the mere knowledge of the shapes of `a` and `b` will not suffice when computing the shape of `c`.

- **Type III**.

  These are built-in functions that neither belong to Type I nor Type II. For these operators, even full knowledge of the arguments does not suffice when computing the shape of the result. For example, there exists a built-in function called `dbstack` that returns the stack trace information as a column vector [Mat97]:

  ```
  c ← dbstack;
  ```

  In the case of such built-in functions, a complete execution history may be necessary to determine the shape of the result. Type III operators appear to be few and will not be discussed further in this report.

Table 1 shows some of MATLAB's built-in functions grouped by the shape-based taxonomy described above. Notice that certain built-in functions such as `rand` can be considered as being either Type I or Type II depending on which overloaded version is invoked. For instance, when invoked without arguments, `rand` always returns a scalar-shaped result and therefore behaves as a Type I built-in function. When invoked with arguments, say `2` and `3`, `rand` produces a matrix of size 2×3 and thus behaves as a Type II built-in function.

A nearly orthogonal way of looking at MATLAB's built-in functions is by considering the rank of their results. Depending on how the rank of its result is connected to its arguments, a built-in function can be regarded as belonging to one of four broad divisions:

- **Simple**: These are built-in functions for which a conservative estimate of the result's rank is known once the ranks of the operands are known. All of the built-functions listed in Table 1 except the shaded entries are Simple by this definition.

- **Rank I**: These are built-in functions that are not Simple and for which a conservative estimate of the result's rank is known only when the shape of at least one operand is known. The `permute` built-in function is an example of a Rank I operator. When applied on a legal array `a`, `permute(a, e)` rearranges the dimensions of `a` so that they are in the order specified by the vector `e`. Thus, just knowing the ranks $\mathcal{R}$(`a`) and $\mathcal{R}$(`e`) will not be enough for determining the rank of the result. However, once we know the number of elements in the vector `e` (i.e., the shape of `e`), we will be able to conservatively estimate the rank of the result: If `e` has the shape tuple $\boldsymbol{s} = \langle 1, p \rangle$, then such an estimate would be $\max(|\boldsymbol{s}|, 2) = \max(p, 2)$.[h]

- **Rank II**: These are built-in functions for which knowledge of the value of at least one argument is necessary in order to arrive at a conservative estimate for the rank of the result. An example of such a built-in function is the `cat` operator. Given two legal arrays `a` and `b`, `cat(`e`, a, b)` produces an array that is the concatenation of `a` and `b` along the eth dimension. Hence, depending on what e is, the canonical rank of the result could be anywhere between 2 to $\max(e, \mathcal{R}(\texttt{a}), \mathcal{R}(\texttt{b}))$.

- **Rank III**: These are built-in functions that are not Simple, Rank I or Rank II. For these operators, even full knowledge of the arguments may not suffice when determining the rank of the result. Examples of built-in functions that belong to this category are the `eval`, `evalin` and `feval` operators.

Table 1: Shape-Based Classification for MATLAB's Built-in Functions

Type I	
Operator	*Rank*
`a*b`	$\max(\mathcal{R}(\texttt{a}), \mathcal{R}(\texttt{b}))$
`a+b`	
`a-b`	
`a.*b`	
`a.^b`	
`a./b`	
`a.\b`	
`a==b`	
`a~=b`	
`a<b`	
`a>b`	
`a<=b`	
`a>=b`	
`a&b`	
`a|b`	
`a/b`	
`a\b`	
`[a, b]`	
`[a; b]`	
`+a`	$\mathcal{R}(\texttt{a})$
`-a`	
`~a`	
`c(:)` $\leftarrow$ `a`	$\mathcal{R}(\texttt{c})$
`a^b`	2
`a(:)`	
`a'`	
`a.'`	
`rand`	

Type II	
Operator	*Rank*
`a:b`	2
`a(`e`)`	$\mathcal{R}(e)$
`permute(a, e)`	$\max(\|\boldsymbol{s}\|, 2)$
`cat(`e`, a, b)`	$\max(e, \mathcal{R}(\texttt{a}), \mathcal{R}(\texttt{b}))$
`a(`$e_1, e_2, \ldots, e_n$`)`	$\max(n, 2)$
`rand(`$e_1, e_2, \ldots, e_n$`)`	
`ones(`$e_1, e_2, \ldots, e_n$`)`	
`c(`e`)` $\leftarrow$ `a`	$\max(\mathcal{R}(\texttt{a}), \mathcal{R}(\texttt{c}))$
`c(`$e_1, e_2, \ldots, e_n$`)` $\leftarrow$ `a`	$\max(n, \mathcal{R}(\texttt{c}))$

Type III	
Operator	*Rank*
`dbstack`	2
`eval(a)`	?
`evalin(a, b)`	
`feval(a, b)`	

[h]The notation $|\boldsymbol{s}|$ denotes the determinant of a square matrix $\boldsymbol{s}$.

The shaded entries in Table 1 denote built-in functions that are not Simple. While `cat` and `permute` are Rank I and Rank II respectively, the remaining shaded entries represent Rank III operators. The `eval`, `evalin` and `feval` built-in functions allow arbitrary strings to be evaluated. However, just knowing what these argument strings are will not always suffice for determining the rank of the result. In the most general setting, a complete execution history may be necessary to determine the rank of the result.

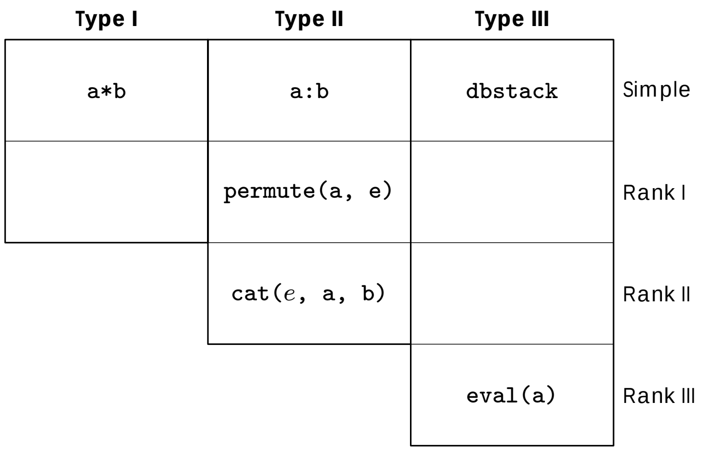

Figure 3: Near-Orthogonal Classifications Based on Shape and Rank

The way the two taxonomies partition all of MATLAB's built-in functions is shown in Figure 3. Observe that by definition, all Type I operators can be either Simple or Rank I. Being Simple does not necessarily mean Type I—Simple built-in functions could be Type I, Type II or Type III. Similarly, Rank I built-in functions could be Type I, Type II or Type III. On the other hand, Rank II built-in functions can only be Type II or Type III, while Rank III operators can only be Type III.

The framework described in this report is meant to primarily address the Simple, Type I function group, and can be extended to handle Simple, Type II operators; it does not address built-in functions in the other groups, which appear to be few in the language. Note that the conventional shadow variable approach addresses the Simple group of built-in functions since the procedure relies on the explicit knowledge of array ranks at compile time.

6.2 The Bounded Property

Most built-in functions in MATLAB appear to be Simple. Members of this group appear to share an important characteristic: Whenever the ranks of their arguments are bounded by a suitable constant, the ranks of their results are also bounded by the same constant. Thus, if we were to consider the MATLAB statement $\mathtt{c} \leftarrow \varphi(\mathtt{a},\ \mathtt{b})$ where φ is a Simple built-in function having the *bounded property*, it is possible to find a constant $\mathfrak{R}_\varphi$ such that for all $\mathcal{R}(\mathtt{a})$ and $\mathcal{R}(\mathtt{b})$,

$$\mathcal{R}(\mathtt{a}) \leq \mathfrak{R}_\varphi \wedge \mathcal{R}(\mathtt{b}) \leq \mathfrak{R}_\varphi \Longrightarrow \mathcal{R}(\mathtt{c}) \leq \mathfrak{R}_\varphi. \tag{15}$$

For example, for the `*` built-in function, since $\mathcal{R}(\mathtt{c}) = \max(\mathcal{R}(\mathtt{a}), \mathcal{R}(\mathtt{b}))$, such a constant $\mathfrak{R}_*$ could be any integer greater than or equal to 2. For the `cat` built-in function, $\mathfrak{R}_{\mathtt{cat}}$ could be any integer greater than or equal to the largest number of arguments in an invocation of the function in the program.

The bounded property is particularly important because it enables us to conservatively estimate at compile time the ranks of *all* expressions in programs that comprise solely of these operators. Such an estimate would be possible even in the presence of general loops since the arrays produced by these built-in functions will not arbitrarily grow in rank. For instance, in the following code fragment,

```
while (...),
      a ← φ(a, b);
end;
```

the canonical rank of the array a will always be at most $\mathfrak{R}_\varphi$, irrespective of how many times the loop is executed.

7 Shape Inferring for Simple, Type I Built-in Functions

The framework described in § 3 can be applied in a straightforward manner to deduce the shapes of all of MATLAB's Simple, Type I operators. In the following subsections, we demonstrate this by showing the workings of the framework for specific built-in functions. In each case, the shape-tuple expressions are formulated so as to form an algebra in $\mathbb{S}$. Because MATLAB considers only the canonical form of a shape tuple in its shape semantics, the equivalence relation $\wp$ exhibits the substitution property with respect to each of the shape-tuple functions. This allows for the consideration of simpler quotient algebras, called *shape-tuple class algebras*, that operate directly on the equivalence classes in $\mathbb{S}_\wp$. Every algebraic system $[\mathbb{S}, \ddot{\star}]$ that corresponds to the shape-tuple operator $\ddot{\star}$ can be replaced by the simpler shape-tuple class algebra $[\mathbb{S}_\wp, \dot{\star}]$ that corresponds to the shape-tuple class operator $\dot{\star}$. By way of design, $\overline{\boldsymbol{\pi}}$ is made the zero element in each of the shape-tuple class algebras. Tables 2 and 3 summarize the discussion and results of this section.

Table 2: Shape Inferring for Some Simple, Type I Built-in Functions

MATLAB Expression	Shape Expression	θ	$\boldsymbol{u}$
`a*b`	$\boldsymbol{s} \ddot{\circledast} \boldsymbol{t}$	$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t}))$ $(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})$ $\delta(\boldsymbol{\Psi s}^*\boldsymbol{\Psi\Gamma_1} - \boldsymbol{t}^*\boldsymbol{\Gamma_1}))\big)$	$(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\big(\boldsymbol{s}^*\alpha(\boldsymbol{t})$ $+\boldsymbol{t}^*\alpha(\boldsymbol{s})(1-\alpha(\boldsymbol{t})) + (\boldsymbol{s}^*\boldsymbol{\Gamma_1}$ $+\boldsymbol{t}^*\boldsymbol{\Gamma_2} + \boldsymbol{\iota}^* - \boldsymbol{\Gamma_1} - \boldsymbol{\Gamma_2})(1-\alpha(\boldsymbol{s}))$ $(1-\alpha(\boldsymbol{t}))\big)$
`a+b` `a-b` `a.*b` `a.^b` `a./b` `a.\b` `a==b` `a~=b` `a<b` `a>b` `a<=b` `a>=b` `a&b` `a|b` `atan2(a, b)` `bitor(a, b)` `bitand(a, b)` `bitxor(a, b)` `max(a, b)` `min(a, b)`	$\boldsymbol{s} \ddot{\oplus} \boldsymbol{t}$	$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t}))$ $(1-\delta(\boldsymbol{s}^*-\boldsymbol{t}^*))\big)$	$(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\big(\boldsymbol{s}^*\alpha(\boldsymbol{t})$ $+\boldsymbol{t}^*(1-\alpha(\boldsymbol{t}))\big)$
`a^b`	$\boldsymbol{s} \ddot{\odot} \boldsymbol{t}$	$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\Big(1-\big(1-\alpha(\boldsymbol{s})\beta(\boldsymbol{t})$ $\delta(\boldsymbol{t\Gamma_1} - \boldsymbol{\Psi t \Psi \Gamma_1})\big)$ $\big(1-\alpha(\boldsymbol{t})\beta(\boldsymbol{s})\delta(\boldsymbol{s\Gamma_1}$ $-\boldsymbol{\Psi s \Psi \Gamma_1})\big)\Big)$	$(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\big(\boldsymbol{s}^*\alpha(\boldsymbol{t})$ $+\boldsymbol{t}^*(1-\alpha(\boldsymbol{t}))\big)$

continued on next page

continued from previous page

MATLAB Expression	Shape Expression	θ	$\boldsymbol{u}$
`a'` `a.'`	$\ddot{\neg}\,\boldsymbol{s}$	$\beta(\boldsymbol{s})$	$(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\boldsymbol{\Psi}\boldsymbol{s}^*\boldsymbol{\Psi}$

`+a`
`-a`
`~a`
`sin(a)`
`cos(a)`
`tan(a)`
`sinh(a)`
`cosh(a)`
`tanh(a)`
`asin(a)`
`acos(a)`
`atan(a)`
`abs(a)`
`exp(a)`
`log(a)`
`conj(a)`
`sqrt(a)`
`floor(a)`
`ceil(a)`
`fix(a)`
`round(a)`
`cumprod(a)`
`cumprod(a, b)`
`cumsum(a)`
`cumsum(a, b)`
`bitcmp(a, b)`
`bitshift(a, b)`
`bitset(a, b)`
`fft(a)`
`fftn(a)`
	$\ddot{\nabla}\,\boldsymbol{s}$	$\theta(\boldsymbol{s})$	$\boldsymbol{s}$

`a/b`	$\boldsymbol{s}\ddot{\oslash}\boldsymbol{t}$	$\theta(\boldsymbol{s})\theta(\boldsymbol{t})(1-\alpha(\boldsymbol{t}))\big(1-\alpha(\boldsymbol{s})$ $(1-\beta(\boldsymbol{t}))\big)\big(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})$ $\delta(\boldsymbol{s}^*\boldsymbol{\Gamma_2}-\boldsymbol{t}^*\boldsymbol{\Gamma_2})\big)$	$(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\Big(\boldsymbol{s}^*\alpha(\boldsymbol{t})$ $+\boldsymbol{t}^*(1-\beta(\boldsymbol{t})) + \big(\boldsymbol{s}^*\boldsymbol{\Gamma_1} + \boldsymbol{\iota}^*$ $-\boldsymbol{\Gamma_1} - \boldsymbol{\Gamma_2} + \boldsymbol{\Psi}\boldsymbol{t}^*\boldsymbol{\Psi}\boldsymbol{\Gamma_2}\big)(1-\alpha(\boldsymbol{t}))\beta(\boldsymbol{t})\Big)$
`a\b`	$\boldsymbol{s}\ddot{\circ}\boldsymbol{t}$	$\theta(\boldsymbol{s})\theta(\boldsymbol{t})(1-\alpha(\boldsymbol{s}))\big(1-\alpha(\boldsymbol{t})$ $(1-\beta(\boldsymbol{s}))\big)\big(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})$ $\delta(\boldsymbol{s}^*\boldsymbol{\Gamma_1}-\boldsymbol{t}^*\boldsymbol{\Gamma_1})\big)$	$(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\Big(\boldsymbol{t}^*\alpha(\boldsymbol{s})$ $+\boldsymbol{s}^*(1-\beta(\boldsymbol{s})) + \big(\boldsymbol{\Psi}\boldsymbol{s}^*\boldsymbol{\Psi}\boldsymbol{\Gamma_1} + \boldsymbol{\iota}^*$ $-\boldsymbol{\Gamma_1} - \boldsymbol{\Gamma_2} + \boldsymbol{t}^*\boldsymbol{\Gamma_2}\big)(1-\alpha(\boldsymbol{s}))\beta(\boldsymbol{s})\Big)$
`[a; b]`	$\boldsymbol{s}\ddot{\circledcirc}\boldsymbol{t}$	$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\delta\big(\boldsymbol{s}^*(\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1})$ $-\boldsymbol{t}^*(\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1})\big)$	$(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})(\boldsymbol{s}^*$ $+\boldsymbol{t}^*\boldsymbol{\Gamma_1})$
`[a, b]`	$\boldsymbol{s}\ddot{\ominus}\boldsymbol{t}$	$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\delta\big(\boldsymbol{s}^*(\boldsymbol{\iota}^*-\boldsymbol{\Gamma_2})$ $-\boldsymbol{t}^*(\boldsymbol{\iota}^*-\boldsymbol{\Gamma_2})\big)$	$(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})(\boldsymbol{s}^*\boldsymbol{\Gamma_2}$ $+\boldsymbol{t}^*)$

7.1 Matrix Power

In MATLAB, it is possible to raise square matrices to a scalar power and to raise scalars to a square matrix power [Mat97]. This is made possible by the matrix power built-in function:

```
c ← a^b;
```

The result `c` in the above statement is defined only if either `a` or `b` are square matrices and the other operand is a scalar. The result will then be a square matrix having the same shape tuple as the nonscalar operand in the matrix power expression. If both operands are scalars, the result will also be scalar. From these semantics, we can conclude that `c` will always have a canonical rank of 2:

$$\mathcal{R}(\mathtt{c}) = 2. \tag{16}$$

These semantics could also be translated into a correctness shape predicate and into the shape tuple $\boldsymbol{u}$ of the result:

$$\theta(\boldsymbol{u}) = \theta(\boldsymbol{s})\theta(\boldsymbol{t})\Big(1 - \big(1 - \alpha(\boldsymbol{s})\beta(\boldsymbol{t})\delta(\boldsymbol{t}\boldsymbol{\Gamma_1} - \boldsymbol{\Psi}\boldsymbol{t}\boldsymbol{\Psi}\boldsymbol{\Gamma_1})\big)\big(1 - \alpha(\boldsymbol{t})\beta(\boldsymbol{s})\delta(\boldsymbol{s}\boldsymbol{\Gamma_1} - \boldsymbol{\Psi}\boldsymbol{s}\boldsymbol{\Psi}\boldsymbol{\Gamma_1})\big)\Big), \tag{17}$$

$$\boldsymbol{u} = (1 - \theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\big(\boldsymbol{s}^*\alpha(\boldsymbol{t}) + \boldsymbol{t}^*(1 - \alpha(\boldsymbol{t}))\big). \tag{18}$$

In Equation (17), $\boldsymbol{\Psi}$ denotes the *elementary square matrix* [Ban93]:

$$\boldsymbol{\Psi} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}. \tag{19}$$

Any square matrix premultiplied and postmultiplied with the elementary square matrix will have its first two principal diagonal elements interchanged. For example, $\boldsymbol{\Psi}\langle 2, 3, 4, 5\rangle\boldsymbol{\Psi} = \langle 3, 2, 4, 5\rangle$.

Note that in this case, $\mathcal{R}(\mathtt{c})$ could be less than either $\mathcal{R}(\mathtt{a})$ or $\mathcal{R}(\mathtt{b})$. Therefore, the promoted shape tuples $\boldsymbol{s}^*$ and $\boldsymbol{t}^*$ in Equation (18) may have to be obtained by truncating $\boldsymbol{s}$ or $\boldsymbol{t}$ to $\mathcal{R}(\mathtt{c})$ components, as mentioned in § 3.2. Since $\mathcal{R}(\mathtt{c})$ is 2, $\boldsymbol{s}^*$ and $\boldsymbol{t}^*$ are obtained by discarding all extents from the third component on in $\boldsymbol{s}$ and $\boldsymbol{t}$ respectively.

It can be shown that Equation (18) defines an algebraic system $[\mathbb{S}, \dot{\odot}]$. It can further be shown that the shape-tuple class algebra $[\mathbb{S}_\wp, \dot{\odot}]$ is a commutative monoid in which the idempotent law does not hold.

7.2 Transpose

In MATLAB, the complex conjugate transpose (`'`) and the nonconjugate transpose (`.'`) operators require the operand to be a matrix. These built-in functions are not defined on higher dimensional arrays. When applied on matrices, they resemble the traditional matrix transpose operation. We thus have,

$$\mathcal{R}(\mathtt{c}) = 2, \tag{20}$$

$$\theta(\boldsymbol{u}) = \beta(\boldsymbol{s}), \tag{21}$$

$$\boldsymbol{u} = (1 - \theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\boldsymbol{\Psi}\boldsymbol{s}^*\boldsymbol{\Psi}. \tag{22}$$

Clearly, Equation (22) defines an algebraic system $[\mathbb{S}, \ddot{\neg}]$ in which $\ddot{\neg}$ is a unary shape-tuple operator. Note that from Equations (21) and (22), we also have

$$\ddot{\neg}\boldsymbol{\pi} = \boldsymbol{\pi}$$

even though the first two components of the illegal shape tuple as defined in Equation (1) may not be the same.

7.3 Matrix Division

In MATLAB, it is possible to *right divide* (`/`) or *left divide* (`\`) a matrix by another matrix. Consider the following right matrix division operation:

```
c ← a/b;
```

If `b` is a scalar, `a` can be any legal array. The outcome `c` will then be a legal array such that `c*b` and `a` are the same. If `b` is instead a higher dimensional array, `a` is allowed to only be a scalar. The result `c` is then a legal array such that `c.*b` and `a*ones(size(b))` are equal. Finally, if `b` is a nonscalar matrix, `a` can only be a matrix such that the extents along the second dimensions of `b` and `a` match. These semantics therefore yield the following equations:

$$\begin{aligned}
\mathcal{R}(\mathtt{c}) &= \max(\mathcal{R}(\mathtt{a}), \mathcal{R}(\mathtt{b})), && (23)\\
\theta(\boldsymbol{u}) &= \theta(\boldsymbol{s})\theta(\boldsymbol{t})(1-\alpha(\boldsymbol{t}))\big(1-\alpha(\boldsymbol{s})(1-\beta(\boldsymbol{t}))\big)\big(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})\delta(\boldsymbol{s}^*\boldsymbol{\Gamma_2}-\boldsymbol{t}^*\boldsymbol{\Gamma_2})\big), && (24)\\
\boldsymbol{u} &= (1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\big(\boldsymbol{s}^*\alpha(\boldsymbol{t}) + \boldsymbol{t}^*(1-\beta(\boldsymbol{t})) && (25)\\
&\quad + (\boldsymbol{s}^*\boldsymbol{\Gamma_1} + \boldsymbol{\iota}^* - \boldsymbol{\Gamma_1} - \boldsymbol{\Gamma_2} + \boldsymbol{\Psi}\boldsymbol{t}^*\boldsymbol{\Psi}\boldsymbol{\Gamma_2})(1-\alpha(\boldsymbol{t}))\beta(\boldsymbol{t})\big). && (26)
\end{aligned}$$

Similarly, a left matrix division operation is represented in MATLAB as `a\b`. In this operation, if `a` is a scalar, `b` can be any legal array. The result `c` is a legal array such that `a*c` and `b` are equal. If `a` is a higher dimensional array, `b` can only be a scalar and the result would be a legal array such that `a.*c` and `b*ones(size(a))` are the same. Lastly, if `a` is a nonscalar matrix, `b` must be a matrix such that the extents along the first dimensions of `a` and `b` match. These semantics therefore give rise to the following equations:

$$\begin{aligned}
\mathcal{R}(\mathtt{c}) &= \max(\mathcal{R}(\mathtt{a}), \mathcal{R}(\mathtt{b})), && (27)\\
\theta(\boldsymbol{u}) &= \theta(\boldsymbol{s})\theta(\boldsymbol{t})(1-\alpha(\boldsymbol{s}))\big(1-\alpha(\boldsymbol{t})(1-\beta(\boldsymbol{s}))\big)\big(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})\delta(\boldsymbol{s}^*\boldsymbol{\Gamma_1}-\boldsymbol{t}^*\boldsymbol{\Gamma_1})\big), && (28)\\
\boldsymbol{u} &= (1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\Big(\boldsymbol{t}^*\alpha(\boldsymbol{s}) + \boldsymbol{s}^*(1-\beta(\boldsymbol{s})) && (29)\\
&\quad + \big(\boldsymbol{\Psi}\boldsymbol{s}^*\boldsymbol{\Psi}\boldsymbol{\Gamma_1} + \boldsymbol{\iota}^* - \boldsymbol{\Gamma_1} - \boldsymbol{\Gamma_2} + \boldsymbol{t}^*\boldsymbol{\Gamma_2}\big)(1-\alpha(\boldsymbol{s}))\beta(\boldsymbol{s})\Big). && (30)
\end{aligned}$$

We shall represent the algebraic systems formed by Equations (25) and (29) as $[\mathbb{S}, \ddot{\oslash}]$ and $[\mathbb{S}, \ddot{\circ}]$ respectively. It can be shown that the simplified algebras that they correspond to—namely, $[\mathbb{S}_\wp, \dot{\oslash}]$ and $[\mathbb{S}_\wp, \dot{\circ}]$—lack an identity element, are neither associative nor commutative, and do not satisfy the idempotent law.

7.4 Vertical and Horizontal Concatenation

In § 6.1, we discussed the `cat` built-in function as being an example of a Rank II, Type II operator. To review, the invocation `cat(`e`, a, b)` produces an array that is the concatenation of the legal arrays `a` and `b` along the eth dimension. For the relatively common cases when e is 1 or 2, MATLAB offers the vertical concatenation (`[;]`) and horizontal concatenation (`[,]`) operators.

If `a` and `b` are two legal arrays whose extents along all dimensions except the first match, then it is possible to concatenate these two arrays vertically:

```
c ← [a; b];
```

The result `c` would have the sum of the extents of `a` and `b` along the first dimension, and the same extents as either of them along the remaining dimensions. We therefore get

$$\begin{aligned}
\mathcal{R}(\mathtt{c}) &= \max(\mathcal{R}(\mathtt{a}), \mathcal{R}(\mathtt{b})), && (31)\\
\theta(\boldsymbol{u}) &= \theta(\boldsymbol{s})\theta(\boldsymbol{t})\delta\big(\boldsymbol{s}^*(\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1}) - \boldsymbol{t}^*(\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1})\big), && (32)\\
\boldsymbol{u} &= (1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})(\boldsymbol{s}^* + \boldsymbol{t}^*\boldsymbol{\Gamma_1}). && (33)
\end{aligned}$$

Likewise, if `a` and `b` are two legal arrays whose extents along all dimensions except the second match, it is possible to concatenate these two arrays horizontally:

```
c ← [a, b];
```

The result `c` would then be an array having the same extents as either `a` or `b` along all dimensions except the second, while along the second dimension, its extent would equal the sum of the extents of `a` and `b` along that dimension:

$$\mathcal{R}(\mathtt{c}) = \max(\mathcal{R}(\mathtt{a}), \mathcal{R}(\mathtt{b})), \tag{34}$$

$$\theta(\boldsymbol{u}) = \theta(\boldsymbol{s})\theta(\boldsymbol{t})\delta\big(\boldsymbol{s}^*(\boldsymbol{\iota}^* - \boldsymbol{\Gamma_2}) - \boldsymbol{t}^*(\boldsymbol{\iota}^* - \boldsymbol{\Gamma_2})\big), \tag{35}$$

$$\boldsymbol{u}^* = (1 - \theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})(\boldsymbol{s}^*\boldsymbol{\Gamma_2} + \boldsymbol{t}^*). \tag{36}$$

Once again, Equations (33) and (36) form algebras. We shall denote the two algebraic systems given rise to by these equations as $[\mathbb{S}, \ddot{\circledcirc}]$ and $[\mathbb{S}, \ddot{\ominus}]$ respectively. It can be shown that the quotient algebras $[\mathbb{S}_\wp, \dot{\circledcirc}]$ and $[\mathbb{S}_\wp, \dot{\ominus}]$ for these algebras do not have an identity element, are associative and commutative, and do not satisfy the idempotent law.

Example 2: *Horizontal Concatenation*
Consider the MATLAB statement `c ← [a, b]` and suppose that the shape tuples of `a` and `b` are $\boldsymbol{s} = \langle p_1, p_2\rangle$ and $\boldsymbol{t} = \langle q_1, q_2\rangle$ respectively. From Equations (34), (35) and (36), we thus obtain

$$\begin{aligned}
\mathcal{R}(\mathtt{c}) &= 2,\\
\theta(\boldsymbol{u}) &= \theta(\boldsymbol{s})\theta(\boldsymbol{t})\delta(p_1 - q_1),\\
\boldsymbol{u} &= \begin{pmatrix} \theta(\boldsymbol{u})p_1 + (1-\theta(\boldsymbol{u}))\pi_1 & 0 \\ 0 & \theta(\boldsymbol{u})(p_2+q_2) + (1-\theta(\boldsymbol{u}))\pi_2 \end{pmatrix},
\end{aligned}$$

where $\boldsymbol{u}$ denotes the shape tuple of `c`. ■

Table 3: Shape-Tuple Class Algebras

Shape-Tuple Class Operator	Identity	Associativity	Commutativity	Idempotent Law
$\dot{\circledast}$ (e.g., `a*b`)	$\bar{\iota}$	✗	✗	✗
$\dot{\oplus}$ (e.g., `a+b`, `a-b`, `a.*b`)	$\bar{\iota}$	✓	✓	✓
$\dot{\nabla}$ (e.g., `fft(a)`)	-	-	-	-
$\dot{\odot}$ (e.g., `a^b`)	$\bar{\iota}$	✓	✓	✗
$\dot{\neg}$ (e.g., `a'`)	-	-	-	-
$\dot{\oslash}$ (e.g., `a/b`)	✗	✗	✗	✗
$\dot{\circ}$ (e.g., `a.\b`)	✗	✗	✗	✗
$\dot{\circledcirc}$ (e.g., `[a; b]`)	✗	✓	✓	✗
$\dot{\ominus}$ (e.g., `[a, b]`)	✗	✓	✓	✗

8 Extensions to Simple, Type II Built-in Functions

The shape inferring framework can be extended to handle Simple, Type II built-in functions. Consider the `rand` built-in function discussed in § 6:

```
c ← rand(a, b);
```

When the MATLAB run-time system executes the above statement, it expects `a` and `b` to be scalars. It then considers the run-time *integer values* a' and b' of `a` and `b` respectively, and creates a matrix of size $((a' + |a'|)/2) \times ((b' + |b'|)/2)$ filled with random real numbers. If an operand is an empty array, zero is taken to be its integer value. Here, $|a'|$ and $|b'|$ indicate the absolute values of a' and b' respectively. We therefore obtain

$$\mathcal{R}(\texttt{c}) = 2, \tag{37}$$

$$\theta(\boldsymbol{u}) = 1, \tag{38}$$

$$\boldsymbol{u} = \frac{a' + |a'|}{2}\boldsymbol{\Gamma_1} + \frac{b' + |b'|}{2}\boldsymbol{\Gamma_2}. \tag{39}$$

In general, if either `a` or `b` are nonempty arrays, MATLAB considers only their first elements while evaluating `rand(a, b)`—it issues a warning if either `a` or `b` are nonempty and nonscalar. The framework could have chosen to either flag such warnings as errors, or ignore them and therefore regard them as being legal. In the present version of the framework, we chose the latter since this simplifies the expressions for the correctness shape predicate and the shape tuple $\boldsymbol{u}$ of the result, as can be seen from Equations (38) and (39).

9 Comparisons

We now show how the shape inferring framework performs using two examples. In both of these examples, the algebraic properties of the shape-tuple operators involved are exploited to perform a compile-time inference.

Example 3: *Comparisons with De Rose's Approach*
Let us reconsider the code fragment shown in Figure 1. In De Rose's approach, the static inferring mechanism will fail because the extents of the matrices `a` and `b` will not be known exactly at compile time. For both `a` and `b`, shadow variables will be generated at compile time to resolve the shape information at run time. The approach will not be capable of important static inferences such as (1) if the assignment to `d` succeeds, then the subsequent assignment to `e` will also succeed and (2) that both `e` and `d` will then have the same shape. In our framework, we obtain the following two equations corresponding to those two statements:

$$\overline{\boldsymbol{s_d}} = \overline{\boldsymbol{s_c}} \dot{\oplus} \overline{\boldsymbol{s_a}}, \tag{Eg-3.1}$$

$$\overline{\boldsymbol{s_e}} = \overline{\boldsymbol{s_d}} \dot{\oplus} \overline{\boldsymbol{s_a}}, \tag{Eg-3.2}$$

where $\boldsymbol{s_c}$, $\boldsymbol{s_a}$, $\boldsymbol{s_d}$ and $\boldsymbol{s_e}$ are the shape tuples of `c`, `a`, `d` and `e` respectively. By substituting Equation (Eg-3.1) into Equation (Eg-3.2), we obtain

$$\overline{\boldsymbol{s_e}} = (\overline{\boldsymbol{s_c}} \dot{\oplus} \overline{\boldsymbol{s_a}}) \dot{\oplus} \overline{\boldsymbol{s_a}}.$$

By associativity,

$$\overline{\boldsymbol{s_e}} = \overline{\boldsymbol{s_c}} \dot{\oplus} (\overline{\boldsymbol{s_a}} \dot{\oplus} \overline{\boldsymbol{s_a}}).$$

By the idempotent law, the last equation becomes

$$\overline{\boldsymbol{s_e}} = \overline{\boldsymbol{s_c}} \dot{\oplus} \overline{\boldsymbol{s_a}}. \tag{Eg-3.3}$$

Comparing Equations (Eg-3.1) and (Eg-3.3), we conclude that $\overline{\boldsymbol{s_e}} = \overline{\boldsymbol{s_d}}$. Thus, if the assignment to `d` succeeds (in which case $\overline{\boldsymbol{s_d}}$ won't be $\overline{\boldsymbol{\pi}}$), the subsequent assignment to `e` will also succeed and both `e` and `d` would then have the same shape. Therefore at run time, we need to perform conformability checking only for the first statement.

Observe that this result is deducible by our framework even when `a` and `b` are *arbitrary* arrays, not necessarily just matrices. For example, if the last three statements in Figure 1 were part of a function definition in which `a` and `b` were the formal parameters, the framework would still arrive at the above result. Such a generalized inference is not possible in De Rose's scheme. ■

Example 4: *Inferring in the Presence of Loops*
Consider the following code fragment that involves a `while` loop:

```
S1:  a ← ...;
S2:  b ← ...;
S3:  while (...),
S4:        c ← a.*b;
S5:        a ← c;
S6:  end;
```

From statement S_4 and Tables 2 and 3, we get

$$\overline{\boldsymbol{u_i}} = \overline{\boldsymbol{s_{i-1}}}\dot{\oplus}\overline{\boldsymbol{t}}. \tag{Eg-4.1}$$

where $\boldsymbol{s_i}$, $\boldsymbol{t}$ and $\boldsymbol{u_i}$ denote the respective shape tuples of `a`, `b` and `c` in the ith iteration ($i \geq 1$) of the loop. From statement S_5, we also have

$$\overline{\boldsymbol{s_i}} = \overline{\boldsymbol{u_i}}. \tag{Eg-4.2}$$

Hence, by substituting Equation (Eg-4.1) into Equation (Eg-4.2), we arrive at

$$\overline{\boldsymbol{s_i}} = \overline{\boldsymbol{s_{i-1}}}\dot{\oplus}\overline{\boldsymbol{t}}.$$

Therefore,

$$\overline{\boldsymbol{s_i}} = (\overline{\boldsymbol{s_{i-2}}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{t}} = \overline{\boldsymbol{s_{i-2}}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{t}}).$$

Hence,

$$\overline{\boldsymbol{s_i}} = \overline{\boldsymbol{s_{i-2}}}\dot{\oplus}\overline{\boldsymbol{t}}.$$

Proceeding thus, we can arrive at the following:

$$\overline{\boldsymbol{s_i}} = \overline{\boldsymbol{s_0}}\dot{\oplus}\overline{\boldsymbol{t}} \text{ for all } i \geq 1. \tag{Eg-4.3}$$

The result in Equation (Eg-4.3) is important because it leads to the following useful inferences and optimizations:

- If the assignments to `a` and `b` in statements S_1 and S_2 are shape correct, then the code fragment is itself shape correct so long as `a` and `b` are initially shape conforming with respect to the `.*` built-in function. (These two requirements are captured by the single condition $\overline{\boldsymbol{s_0}}\dot{\oplus}\overline{\boldsymbol{t}} \neq \overline{\boldsymbol{\pi}}$).
- We now know that `c`'s shape will remain the same throughout the loop's execution.
- The result indicates that `a` could potentially change shape only at the first iteration of the loop.
- The result therefore enables us to preallocate `c` and resize `a` before executing the loop.

Note that the above inferences would not have been easily drawn had De Rose's approach been used. ■

10 Handling Control Flow

To handle arbitrary control flow, we consider the SSA representation [CFRW91] of a MATLAB program. By introducing an ancillary variable P, called the *shadow-path variable*, the framework could be extended to support the ϕ construct that is central to the SSA representation.

Consider a join node `c` $\leftarrow$ ϕ`(a, b)` in the SSA form [CFRW91] of a MATLAB program. The rank, correctness shape predicate and shape tuple of `c` can be inferred as follows:

$$\mathcal{R}(\mathtt{c}) = \max(\mathcal{R}(\mathtt{a}), \mathcal{R}(\mathtt{b})), \tag{40}$$

$$\theta(\boldsymbol{u}) = \delta(P)\theta(\boldsymbol{s}) + (1 - \delta(P))\theta(\boldsymbol{t}), \tag{41}$$

$$\boldsymbol{u} = (1 - \theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\big(\boldsymbol{s}^*\delta(P) + \boldsymbol{t}^*(1 - \delta(P))\big). \tag{42}$$

In the above, $\boldsymbol{s}$ and $\boldsymbol{t}$ are the shape tuples of `a` and `b`, and P is the *shadow path variable* that plays the role of a run-time selector. Every join node in a program's control-flow graph (CFG) has its own unique selector P. Because a join node in a MATLAB CFG can be assumed to have only two predecessors, its shadow-path variable takes on a value of 0 or 1 to indicate the incident edge along which control flows into it; this value can flip-flop because CFG nodes may be revisited in the course of a program's execution.

11 Reducing Array Conformability Checks

By enabling the computation of a shape-tuple expression prior to invoking the associated built-in function, the shape inferring framework effectively permits an implementation to in-line a built-in function's conformability checking code at the call site. This in turn may facilitate a reduction in the overall conformability checking overhead through the application of traditional compiler techniques such as copy propagation, common-subexpression elimination (CSE) and dead-code elimination.

Figure 4 shows a translation of the code excerpt in Figure 1, with code due to the shape inferring framework indicated by a ▶ prefix. The inference that was made in Example 3 is responsible for the assignment $\boldsymbol{s_e} \leftarrow \boldsymbol{s_d}$. The actual conformability checks occur through the *assert* calls—*assert*(B) tests whether the Boolean expression B is true at run time and exits if false. Note that for the first four shape tuples in Figure 4, no run-time assertions need to be made since the correctness shape predicates for them are statically determinable. The same applies to the correctness shape predicates $\theta(\boldsymbol{s_a})$ and $\theta(\boldsymbol{s_b})$ since, by Equation (38), they are statically known to be 1 each. After applying copy propagation to $\boldsymbol{s_e}$, a redundant call to *assert*$\big(\theta(\boldsymbol{s_d}) = 1\big)$ is generated; this is shown in Figure 5. By applying CSE, this redundant call can be identified and eliminated. Dead-code elimination could then be used on the shape-tuple computations to produce the final result shown in Figure 6.

```
▶ $\boldsymbol{s_m} \leftarrow \langle 1, 1\rangle$
m ← round(4*rand+1);
▶ $\boldsymbol{s_n} \leftarrow \langle 1, 1\rangle$
n ← round(5*rand+1);
▶ $\boldsymbol{s_x} \leftarrow \langle 1, 1\rangle$
x ← round(5*rand+1);
▶ $\boldsymbol{s_y} \leftarrow \langle 1, 1\rangle$
y ← round(6*rand+1);

▶ $\boldsymbol{s_a} \leftarrow \langle (m' + |m'|)/2, (n' + |n'|)/2\rangle$
a ← rand(m, n);
▶ $\boldsymbol{s_b} \leftarrow \langle (x' + |x'|)/2, (y' + |y'|)/2\rangle$
b ← rand(x, y);

▶ $\boldsymbol{s_c} \leftarrow \boldsymbol{s_a} \ddot{\circledast} \boldsymbol{s_b}$; $\mathit{assert}\big(\theta(\boldsymbol{s_c}) = 1\big)$
c ← mtimes(a, b);
▶ $\boldsymbol{s_d} \leftarrow \boldsymbol{s_c} \dot{\oplus} \boldsymbol{s_a}$; $\mathit{assert}\big(\theta(\boldsymbol{s_d}) = 1\big)$
d ← plus(c, a);
▶ $\boldsymbol{s_e} \leftarrow \boldsymbol{s_d}$; $\mathit{assert}\big(\theta(\boldsymbol{s_e}) = 1\big)$
e ← minus(d, a);
```

Figure 4: Checking Code Inlined

```
▶ $\boldsymbol{s_m} \leftarrow \langle 1, 1\rangle$
m ← round(4*rand+1);
▶ $\boldsymbol{s_n} \leftarrow \langle 1, 1\rangle$
n ← round(5*rand+1);
▶ $\boldsymbol{s_x} \leftarrow \langle 1, 1\rangle$
x ← round(5*rand+1);
▶ $\boldsymbol{s_y} \leftarrow \langle 1, 1\rangle$
y ← round(6*rand+1);

▶ $\boldsymbol{s_a} \leftarrow \langle (m' + |m'|)/2, (n' + |n'|)/2\rangle$
a ← rand(m, n);
▶ $\boldsymbol{s_b} \leftarrow \langle (x' + |x'|)/2, (y' + |y'|)/2\rangle$
b ← rand(x, y);

▶ $\boldsymbol{s_c} \leftarrow \boldsymbol{s_a} \ddot{\circledast} \boldsymbol{s_b}$; $\mathit{assert}\big(\theta(\boldsymbol{s_c}) = 1\big)$
c ← mtimes(a, b);
▶ $\boldsymbol{s_d} \leftarrow \boldsymbol{s_c} \dot{\oplus} \boldsymbol{s_a}$; $\mathit{assert}\big(\theta(\boldsymbol{s_d}) = 1\big)$
d ← plus(c, a);
▶ $\boldsymbol{s_e} \leftarrow \boldsymbol{s_d}$; $\mathit{assert}\big(\theta(\boldsymbol{s_d}) = 1\big)$
e ← minus(d, a);
```

Figure 5: After Copy Propagation

```
m ← round(4*rand+1);
n ← round(5*rand+1);
x ← round(5*rand+1);
y ← round(6*rand+1);

▶ $\boldsymbol{s_a} \leftarrow \langle (m' + |m'|)/2, (n' + |n'|)/2\rangle$
a ← rand(m, n);
▶ $\boldsymbol{s_b} \leftarrow \langle (x' + |x'|)/2, (y' + |y'|)/2\rangle$
b ← rand(x, y);

▶ $\boldsymbol{s_c} \leftarrow \boldsymbol{s_a} \ddot{\circledast} \boldsymbol{s_b}$; $\mathit{assert}(\theta(\boldsymbol{s_c}) = 1)$
c ← mtimes(a, b);
▶ $\boldsymbol{s_d} \leftarrow \boldsymbol{s_c} \dot{\oplus} \boldsymbol{s_a}$; $\mathit{assert}(\theta(\boldsymbol{s_d}) = 1)$
d ← plus(c, a);
e ← minus(d, a);
```

Figure 6: After CSE and Dead-Code Elimination

Note that all of the shape-tuple component arithmetic in the embedded code of Figure 6 can be efficiently mapped by an interpreter or a compiler to a machine's instruction set since they only involve scalar, floating-point calculations.

12 Preallocation

Preallocation is an optimization that can often improve the performance of MATLAB and APL codes. In [MP99a], an improvement by a factor of 4 due to this optimization is reported for the Euler-Cromer program in the FALCON benchmark suite. The basic idea behind using the framework to realize this optimization is to move all shape-tuple computations associated with the body of the loop, outside the loop. This can be done if all the shape-tuple computations are of the Type I kind, since in that case, each shape-tuple expression would be dependent only on earlier shape-tuple expressions. For example, consider the `for` loop construct shown in Figure 7. Given `for i =` *expr*`, ...; end`, MATLAB executes the loop n times, where n is the number of columns in the MATLAB expression *expr* [Mat97]. With every iteration of the loop, `i` will be assigned the successive column vectors in *expr*. Modifications to either *expr* or `i` within the body of the loop do not change the initially determined iteration count n. (In that way, these loops resemble the `do` loops in FORTRAN 77.)

The first statement in the loop body of Figure 7 will cause the array `a` to grow; the construction `[a; b]` concatenates `a` and `b` along the first dimension. If an interpreter were to directly execute the loop, the array `a` would be incrementally increased in size with every iteration of the loop, thus impacting performance. Instead, we move the shape-tuple computations associated with these two MATLAB statements, which are

$\boldsymbol{s_a} \leftarrow \boldsymbol{s_a} \ddot{\odot} \boldsymbol{s_b}$ and $\boldsymbol{s_c} \leftarrow \boldsymbol{s_a} \ddot{\oplus} \boldsymbol{s_c}$ from Table 2, outside the loop. This is shown in Figure 8, where for brevity, the conformability checking code has been omitted. The code hoisting is valid because these shape-tuple computations are only dependent on the initial values of $\boldsymbol{s_a}$, $\boldsymbol{s_b}$ and $\boldsymbol{s_c}$. In addition, we execute the hoisted shape-tuple computations $p_2 \times \cdots \times p_k$ times where $\boldsymbol{s_e} = \langle p_1, p_2, \ldots, p_k \rangle$, since this is the number of times that the original loop would actually be executed. In the hoisted code, we also track the maximum sizes of `a` and `c` through the variables A and C; these are updated in every iteration to the maximum of their current value and the determinant of the corresponding shape tuple. Thus, once the hoisted code finishes execution, we would know exactly the sizes to which `a` and `c` must be finally grown.

```
a ← ...;
b ← ...;
c ← ...;
e ← ...;
for i = e,
    a ← [a; b];
    c ← a.^c;
end;
```

FIGURE 7: A MATLAB `for` Loop

```
▶ $\boldsymbol{s_a} \leftarrow \ldots$
a ← ...;
▶ $\boldsymbol{s_b} \leftarrow \ldots$
b ← ...;
▶ $\boldsymbol{s_c} \leftarrow \ldots$
c ← ...;
▶ $\boldsymbol{s_e} \leftarrow \langle p_1, p_2, \ldots, p_k \rangle$
e ← ...;
▶ $A \leftarrow 0$
▶ $C \leftarrow 0$
▶ for $j$ from 1 to $p_2 \times \cdots \times p_k$
      $\boldsymbol{s_a} \leftarrow \boldsymbol{s_a} \ddot{\odot} \boldsymbol{s_b}$
      $\boldsymbol{s_c} \leftarrow \boldsymbol{s_a} \ddot{\oplus} \boldsymbol{s_c}$
      $A \leftarrow \max(A, |\boldsymbol{s_a}|)$
      $C \leftarrow \max(C, |\boldsymbol{s_c}|)$
  endfor
▶ resize a to $A$ elements
▶ resize c to $C$ elements
for i = e,
    a ← [a; b];
    c ← a.^c;
end;
```

FIGURE 8: After Preallocation

Note that the shape tuples themselves do not arbitrarily grow in size. This follows from the bounded property (see § 6.2). Thus, if k, l, m are the initial ranks of `a`, `b` and `c` respectively, then during the loop's execution, the canonical ranks of `a` and `c` will not be larger than $\max(k, l)$ and $\max(k, l, m)$ respectively. (This can be verified by consulting the rank expressions in Table 1.) Hence, because the substitution property is honored by the corresponding shape-tuple functions, we can perform all the shape-tuple computations in Figure 8 assuming $\max(k, l, m)$ components. In this way, none of the shape tuples have to be grown at all.

Note that in the case of Simple, Type I built-in functions, De Rose's approach could be adapted to implement preallocation as described above. This is because for this class of built-in functions, the generated shadow variables will be dependent on only previously generated shadow variables; thus, the shadow variable code will also be eligible for code hoisting. However, unlike De Rose's approach, the framework may permit tighter inferences in specific cases, such as that shown in Example 4.

13 Summary

In this report, we have described a framework using which the shape of a MATLAB expression can be expressed exactly and succinctly at compile time. We showed how such a framework can be consistently applied to handle a large class of MATLAB functions. We also demonstrated how the framework uncovers and exploits the algebras that underlie each of these functions. The unique advantage of our framework over other approaches is that it enables useful static inferences even in the absence of statically determinable array extents. The framework's utility is not restricted to MATLAB alone—it could be applied to perform shape inferring in other array-based languages such as APL that share many of MATLAB's features. The framework described in this report is being integrated into the MATCH compiler, which aims at efficiently compiling a

MATLAB source onto a heterogeneous target consisting of embedded and commercial-off-the-shelf processors [BSC$^+$00].

References

[Ban93] Utpal Banerjee. **Loop Transformations for Restructuring Compilers: The Foundations**. Kluwer Academic Publishers, January 1993. ISBN 0–7923–9318–X.

[BSC$^+$00] Prithviraj Banerjee, U. Nagaraj Shenoy, Alok Choudhary, Scott Hauck, Christopher Bachmann, Malay Haldar, Pramod G. Joisha, Alexander Jones, Abhay Kanhere, Anshuman Nayak, Suresh Periyacheri, Michael Walkden, and David Zaretsky. "A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems". In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 1–10 (Session 2: Compilation 1), April 2000.

[Bud88] Timothy Budd. **An APL Compiler**. Springer-Verlag, Inc., 1988. ISBN 0–387–96643–9.

[CB98] Stéphane Chauveau and François Bodin. "Menhir: An Environment for High Performance MATLAB". In *Proceedings of the 4th International Workshop on Languages, Compilers and Run-Time Systems*, volume 1511 of *Lecture Notes in Computer Science*, pages 27–40. Springer-Verlag, May 1998.

[CFRW91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CLR95] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. **Introduction to Algorithms**. The MIT Electrical Engineering and Computer Science Series. The MIT Press, 1995. ISBN 0–262–03141–8.

[DJK93] P. Drakenberg, P. Jacobson, and B. Kågström. "A CONLAB Compiler for a Distributed-Memory Multicomputer". In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 814–821, March 1993.

[DR96] Luiz Antônio De Rose. "Compiler Techniques for MATLAB Programs". Ph.D. dissertation, University of Illinois at Urbana-Champaign, May 1996.

[DRP96] Luiz Antônio De Rose and David A. Padua. "A MATLAB to FORTRAN 90 Translator and Its Effectiveness". In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 309–316, May 1996.

[FAL] At `http://www.csrd.uiuc.edu/falcon/falcon.html`, The FALCON Project Home Page.

[Jay95] Barry C. Jay. "A Semantics for Shape". *Science of Computer Programming*, 25:251–283, 1995.

[Jay96] Barry C. Jay. "Shape in Computing". *ACM Computing Surveys*, 28(2):355–357, 1996.

[JS98] Barry C. Jay and Paul A. Steckler. "The Functional Imperative: Shape!". In *Proceedings of the 7th European Symposium On Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 139–153. Springer-Verlag, March/April 1998.

[KU78] Marc A. Kaplan and Jeffrey D. Ullman. "A General Scheme for the Automatic Inference of Variable Types". In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 60–75, January 1978.

[KU80] Marc A. Kaplan and Jeffrey D. Ullman. "A Scheme for the Automatic Inference of Variable Types". *Journal of the ACM*, 27(1):128–145, January 1980.

[MAJ] At `http://polaris.cs.uiuc.edu/majic/majic.html`, The MAJIC Project.

[MAT] At `http://www.mathtools.com/matcom4.html`, MATCOM—MATLAB Compiler.

[Mat97] The MathWorks, Inc. *MATLAB: The Language of Technical Computing*, January 1997. Using MATLAB (Version 5).

[MCC] At `http://www.mathworks.com/products/compiler/index.shtml`, The MathWorks—MATLAB Compiler.

[MP99a] Vijay Menon and Keshav Pingali. "A Case for Source-Level Transformations in MATLAB". In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 53–65, October 1999.

[MP99b] Vijay Menon and Keshav Pingali. "High-Level Semantic Optimization of Numerical Codes". In *Proceedings of the 13th ACM International Conference on Supercomputing*, pages 434–443, June 1999.

[OS89] Alan V. Oppenheim and Ronald W. Schafer. **Discrete-Time Signal Processing**. Signal Processing Series. Prentice-Hall, Inc., 1989. ISBN 0–13–216292–X.

[PP75] Raymond P. Polivka and Sandra Pakin. **APL: The Language and Its Usage**. Automatic Computation Series. Prentice-Hall, Inc., 1975. ISBN 0–13–038885–8.

[QMSZ98] Michael J. Quinn, Alexey Malishevsky, Nagajagadeswar Seelam, and Yan Zhao. "Preliminary Results from a Parallel MATLAB Compiler". In *Proceedings of the 12th International Parallel Processing Symposium*, pages 81–87, April 1998.

[TM75] J. P. Tremblay and R. Manohar. **Discrete Mathematical Structures with Applications to Computer Science**. Computer Science Series. McGraw-Hill, Inc., 1975. ISBN 0–07–065142–6.