Computing Array Shapes in MATLAB*

Pramod G. Joisha, U. Nagaraj Shenoy, and Prithviraj Banerjee

Department of Electrical and Computer Engineering, Northwestern University, USA.
[pjoisha, nagaraj, banerjee]@ece.nwu.edu

Abstract. This paper deals with the problem of statically inferring the shape of an array in languages such as MATLAB. Inferring an array's shape is desirable because it empowers better compilation and interpretation; specifically, knowing an array's shape could permit reductions in the number of run-time array conformability checks, enable memory preallocation optimizations, and facilitate the in-lining of "scalarized" code. This paper describes how the shape of a MATLAB expression can be determined statically, based on a methodology of systematic matrix formulations. The approach capitalizes on the algebraic properties that underlie MATLAB's shape semantics and exactly captures the shape that the MATLAB expression assumes at run time. Some of the highlights of the approach are its applicability to a large class of MATLAB functions and its uniformity. Our methods are compared with the previous shadow variable scheme, and we show how the algebraic view allows inferences not deduced by the traditional approach.

1 Introduction

In languages such as MATLAB[1] and APL that lack type declarations, static knowledge of an array's intrinsic type and shape could improve the translated code's execution efficiency. For instance, it could enable 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.[2] Here, the invocations `rand(m, n)` and `rand(x, y)` return a pair of two-dimensional arrays (i.e.,

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

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

[2] The symbol ← will be used to denote the assignment operation in MATLAB.

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 of c be? Note that c is the outcome of a*b where * is the MATLAB matrix multiply operation [10]. According to the semantics of this operation, 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 statically ascribed to c, should we maintain a list of candidate 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?

```
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;
f ← e./d;
```

Fig. 1. A Motivating Code Fragment

1.2 Related Work

In the recent past, the compiler community has witnessed much activity in the area of compilation for the MATLAB language [4, 5, 13, 2, 9, 11]. The work due to Kaplan et al. [8], 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 [1], 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 [1] 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 [5] was among the early works to examine the type determination problem in MATLAB. The FALCON system relies on a static shape inference mechanism that essentially propagates an array's "rank" and shape when possible, and resorts to a dynamic strategy based on shadow variables otherwise. Similar techniques have been adopted in the "Otter" MATLAB compiler [13] and in Menhir [2]. Investigations into shape, using alternate approaches such as category theory, have also been done [6]. 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 unlabeled 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.

1.3 Contributions

This paper 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.
- 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.

1.4 Outline

The rest of this paper is organized as follows. We begin with the underpinnings of the framework in § 2. In § 3, we describe the framework by showing its application to an important operator in MATLAB. Continuing with the same operator, we show in § 4 how the framework uncovers some of the important properties associated with its shape semantics. In § 5, we discuss the applicability of the framework. Comparisons with the current state of the art in shape determination are done in § 6. In § 7, we explain how the framework can handle arbitrary control flow. Two important optimizations that the framework allows, namely reduction in the array conformability check overhead and array preallocation, are presented and discussed in § 8 and § 9. Finally, we conclude the paper in § 10.

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 have n dimensions, 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. 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. We shall use the phrase "higher dimensional array" to describe legal arrays whose canonical ranks are at least 3. 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. MATLAB also supports *empty arrays* [10]; these are legal arrays that contain no data but yet have a shape. To encompass the empty array construct, we allow the shape-tuple components to also be zero.

2.2 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 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.

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 [14])—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*. The shape-tuple operator (also called the *shape-tuple function*) gives us the shape of a MATLAB function's result, given the shapes of its operands.

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 outcome `c` can be computed. We begin by reprising the shape semantics of the matrix multiply operation in MATLAB [10]:

> 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 § 2.2, it will be equivalent to a shape tuple having $\max(k, l)$ components. Therefore, we can conservatively determine the rank of `c` at compile time as being

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

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, we enlist the services of three "shape-predicate" functions—θ, β and α—that map a shape tuple $\boldsymbol{s}$ to the 0/1 set $\mathbb{B}$. 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:

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

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` $\leftarrow$ `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 shape-tuple 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

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

In the above equation, each of the quantities $\boldsymbol{\pi}^*$, $\boldsymbol{s}^*$, $\boldsymbol{t}^*$, $\boldsymbol{\iota}^*$, $\boldsymbol{\Gamma_1}$ and $\boldsymbol{\Gamma_2}$ designate $\mathcal{R}(\mathsf{c}) \times \mathcal{R}(\mathsf{c}) = \max(k,l) \times \max(k,l)$ 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 $\mathcal{R}(\mathsf{c})$ components by appending unit extents. The quantities $\boldsymbol{s}^*$ and $\boldsymbol{t}^*$ are also obtained by promoting $\boldsymbol{s}$ and $\boldsymbol{t}$ to $\mathcal{R}(\mathsf{c})$ components. By having all the matrices in Equation (8) to be of the same size (i.e., $\mathcal{R}(\mathsf{c}) \times \mathcal{R}(\mathsf{c})$), the computation in the equation is well defined.

Example 1: *Matrix Multiplication*
Let us reconsider the previous MATLAB statement `c ← a*b`, and suppose that the shape tuples for `a` and `b` are $\boldsymbol{s} = \langle p_1, p_2 \rangle$ and $\boldsymbol{t} = \langle q_1, q_2, q_3 \rangle$ respectively. From Equation (3), $\mathcal{R}(\mathsf{c}) = 3$; after promoting the shape tuples $\boldsymbol{s}$ and $\boldsymbol{t}$ to $\mathcal{R}(\mathsf{c})$ components, we get

$$\boldsymbol{s}^* = \begin{pmatrix} p_1 & 0 & 0 \\ 0 & p_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad \boldsymbol{t}^* = \begin{pmatrix} q_1 & 0 & 0 \\ 0 & q_2 & 0 \\ 0 & 0 & q_3 \end{pmatrix}.$$

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 $\theta(\boldsymbol{u}) = \theta(\boldsymbol{s})\theta(\boldsymbol{t})\Big(1 - \big(1 - \delta(p_1 - 1)\delta(p_2 - 1)\big)\big(1 - \delta(q_1 - 1)\delta(q_2 - 1)\delta(q_3 - 1)\big)\big(1 - \theta(\boldsymbol{s})\theta(\boldsymbol{t})\delta(q_3 - 1)\delta(p_2 - q_1)\big)\Big)$. Hence, from Equation (8), we get the shape tuple $\boldsymbol{u}$ of `c` to be

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

where $A = \alpha(\boldsymbol{s})$, $B = \alpha(\boldsymbol{t})$, $C = \theta(\boldsymbol{u})$ 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 Exposing an Algebra

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

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

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 $(1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^*$ contributes to the sum. Thus, the expression computed in Equation (8) will always equal one of the following: $\boldsymbol{\pi}^*$, $\boldsymbol{s}^*$, $\boldsymbol{t}^*$ or $\boldsymbol{s}^*\boldsymbol{\Gamma_1}+\boldsymbol{t}^*\boldsymbol{\Gamma_2}+\boldsymbol{\iota}^*-\boldsymbol{\Gamma_1}-\boldsymbol{\Gamma_2}$. Since these are all clearly members of $\mathbb{S}$, Equation (8) defines a mapping $\ddot{\circledast}$ from $\mathbb{S}\times\mathbb{S}$ to $\mathbb{S}$. In other words, $[\mathbb{S},\ddot{\circledast}]$ forms an *algebraic system* [14].

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 $(x_1\bullet x_2)E(x_1'\bullet x_2')$ [14]. It can be shown that with respect to the algebraic system $[\mathbb{S},\ddot{\circledast}]$, the equivalence relation $\wp$ has the substitution property [7]. The substitution property implies that it does not matter which among the equivalent shape tuples is chosen while computing Equation (8); we are guaranteed to always arrive at shape tuples that will at worst differ only by trailing extents of unity.

4.2 A Simpler Algebra

Equivalence relations such as $\wp$ that satisfy the substitution property with respect to some algebraic system are usually called *congruence relations* [14]. 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 $\wp$. Algebraic systems such as $[\mathbb{S}_\wp,\dot{\circledast}]$, called *quotient algebras* [14], 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 instance, if $\overline{\boldsymbol{s}}$ were to denote the equivalence class of the shape tuple $\boldsymbol{s}$ under $\wp$, then, for $[\mathbb{S}_\wp,\dot{\circledast}]$, the following two properties can be shown to hold [7]:

$$\overline{\boldsymbol{\pi}}\dot{\circledast}\overline{\boldsymbol{s}}=\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{\pi}}=\overline{\boldsymbol{\pi}}, \qquad \text{(Annihilation)} \tag{9}$$
$$\overline{\boldsymbol{\iota}}\dot{\circledast}\overline{\boldsymbol{s}}=\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{\iota}}=\overline{\boldsymbol{s}}. \qquad \text{(Identity)} \tag{10}$$

5 Shape Inferring for MATLAB's Built-in Functions

From the perspective of shape determination, 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 composing the shape-tuple functions across the program.

The shape inferring framework presented here is aimed at a particular class of MATLAB functions that we call Type I. (A detailed discussion of a novel shape-based taxonomy of MATLAB's built-in functions is available in [7].) Members of the Type I class, which appear to be a significant majority in the language, produce results whose shapes are completely determined by the shapes of the inputs. Common MATLAB operators, such as matrix multiply and array addition, are Type I; in fact, we have have been able to so far uncover nine quotient algebras to which are isomorphic the shape semantics of over 50 Type I built-in functions [7]. These quotient algebras, called *shape-tuple class algebras*, are summarized in Table 1. The table displays the various shape-tuple class operators (such as the $\dot{\circledast}$ operator discussed in § 4), along with specimen MATLAB expressions whose shape semantics they capture, as well as certain common properties that they can be shown to possess (or not possess) [7]. As we shall in § 6, these simple properties can often be leveraged to make useful inferences even when the shapes are not statically known.

Table 1. Shape-Tuple Class Algebras

Shape-Tuple Class Operator	Identity	Associativity	Commutativity	Idempotent Law
$\dot{\circledast}$ (e.g., `a*b`)	$\overline{\imath}$	✗	✗	✗
$\dot{\oplus}$ (e.g., `a+b`, `a-b`, `a.*b`)	$\overline{\imath}$	✓	✓	✓
$\dot{\nabla}$ (e.g., `fft(a)`)	-	-	-	-
$\dot{\odot}$ (e.g., `a^b`)	$\overline{\imath}$	✓	✓	✗
$\dot{\neg}$ (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]`)	✗	✓	✓	✗

6 Comparisons

The following two examples demonstrate the power of the framework. In both of these examples, the algebraic properties of the shape-tuple operators involved are exploited to perform a static inference.

Example 2: *Comparisons with Rose's Approach*
For the code fragment shown in Figure 1, the static inference mechanism due to Rose 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 `e` and `d` will then have the same shape. In our framework, we obtain the following two equations corresponding to those two

statements after consulting Table 1:

$$\overline{\boldsymbol{s_d}} = \overline{\boldsymbol{s_c}} \dot{\oplus} \overline{\boldsymbol{s_a}}, \tag{Eg-2.1}$$

$$\overline{\boldsymbol{s_e}} = \overline{\boldsymbol{s_d}} \dot{\oplus} \overline{\boldsymbol{s_a}}, \tag{Eg-2.2}$$

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-2.1) into Equation (Eg-2.2), we obtain

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

which by associativity becomes $\overline{\boldsymbol{s_e}} = \overline{\boldsymbol{s_c}} \dot{\oplus} (\overline{\boldsymbol{s_a}} \dot{\oplus} \overline{\boldsymbol{s_a}})$. By the idempotent law, this simplifies to

$$\overline{\boldsymbol{s_e}} = \overline{\boldsymbol{s_c}} \dot{\oplus} \overline{\boldsymbol{s_a}}. \tag{Eg-2.3}$$

Comparing Equations (Eg-2.1) and (Eg-2.3), we can 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 `e` and `d` will then have the same shape. Therefore at run time, we need to perform conformability checking only for the first statement. Furthermore, since `e` and `d` will always have the same shape, a simpler version of the `./` operator could be used to compute `f`, which incidentally, can be inferred to have the same shape as `e` and `d`.

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 four 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 Rose's scheme. ■

EXAMPLE 3: *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 Table 1, we get

$$\overline{\boldsymbol{u_i}} = \overline{\boldsymbol{s_{i-1}}} \dot{\oplus} \overline{\boldsymbol{t}}, \tag{Eg-3.1}$$

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

$$\overline{\boldsymbol{s_i}} = \overline{\boldsymbol{u_i}}. \tag{Eg-3.2}$$

Hence, by substituting Equation (Eg-3.1) into Equation (Eg-3.2), we arrive at

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

Reusing the above, we get

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

Proceeding thus, we can arrive at the following:

$$\overline{\boldsymbol{s_i}} = \overline{\boldsymbol{s_0}} \dot{\oplus} \overline{\boldsymbol{t}} \text{ for all } i \geq 1. \tag{Eg-3.3}$$

The result in Equation (Eg-3.3) is important because it leads to the following useful inferences and optimizations: (1) the code fragment is *shape correct* if the assignments to `a` and `b` in S_1 and S_2 are shape correct, and if `a` and `b` are initially shape conforming with respect to the `.*` built-in function (both of these requirements are expressible by the single condition $\overline{\boldsymbol{s_0}} \dot{\oplus} \overline{\boldsymbol{t}} \neq \overline{\boldsymbol{\pi}}$); (2) the shape of `c` will remain the same throughout the loop's execution; (3) the shape of `a` can potentially change only at the first iteration of the loop; and (4) `c` can therefore be preallocated and `a` resized before executing the loop.

It should be emphasized that these deductions are possible even when full knowledge of the initial shapes of `a` and `b` is lacking; such inferences cannot be drawn if Rose's approach is used. ■

7 Handling Control Flow

To handle arbitrary control flow, we consider the SSA representation [3] 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` $\leftarrow \phi$`(a, b)` in the SSA form of a MATLAB program. The shape of `c` could be inferred as follows:

$$\mathcal{R}(\mathtt{c}) = \max(\mathcal{R}(\mathtt{a}), \mathcal{R}(\mathtt{b})), \tag{11}$$

$$\theta(\boldsymbol{u}) = \delta(P-h)\theta(\boldsymbol{s}) + (1-\delta(P-h))\theta(\boldsymbol{t}), \tag{12}$$

$$\boldsymbol{u} = (1-\theta(\boldsymbol{u}))\boldsymbol{\pi}^* + \theta(\boldsymbol{u})\big(\boldsymbol{s}^*\delta(P-h) + \boldsymbol{t}^*(1-\delta(P-h))\big). \tag{13}$$

In Equations (12) and (13), P takes on an integer value at run time depending on how execution flows. Each of the edges in the program's control-flow graph that merge at a join node are labeled with integers. At run time, the shadow-path variables assume these values whenever control flows along those edges. The particular value h in Equations (12) and (13) is the integer label of the edge between the definition node for `a` and the join node in question. Thus, though it may not be possible to exactly determine $\boldsymbol{u}$ at compile time in such situations, we will still have an exact and compact symbolic representation for it.

8 Reducing Array Conformability Checks

By enabling the computation of a shape-tuple expression prior to invoking the associated built-in function, the 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 2 shows a translation of the code excerpt in Figure 1, with code due to the framework indicated by a ▶ prefix. The inferences that were made in Example 2 are responsible for the invocation $\mathtt{rdivide}_{=}$ (a version of `./` that expects identically shaped

arguments), and for the assignments $s_e \leftarrow s_d$ and $s_f \leftarrow s_d$. 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 2, 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 $\theta(s_a)$ and $\theta(s_b)$ since they can be statically ascertained to be 1 each [7]. After applying copy propagation to s_e and s_f, two redundant calls to $assert\big(\theta(s_d) = 1\big)$ are generated. By applying CSE, these two redundant calls 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 3.

```
▶ s_m ← ⟨1, 1⟩
m ← round(4*rand+1);
▶ s_n ← ⟨1, 1⟩
n ← round(5*rand+1);
▶ s_x ← ⟨1, 1⟩
x ← round(5*rand+1);
▶ s_y ← ⟨1, 1⟩
y ← round(6*rand+1);

▶ s_a ← ⟨m, n⟩
a ← rand(m, n);
▶ s_b ← ⟨x, y⟩
b ← rand(x, y);

▶ s_c ← s_a ⊛̈ s_b; assert(θ(s_c) = 1)
c ← mtimes(a, b);
▶ s_d ← s_c ⊕̈ s_a; assert(θ(s_d) = 1)
d ← plus(c, a);
▶ s_e ← s_d; assert(θ(s_e) = 1)
e ← minus(d, a);
▶ s_f ← s_d; assert(θ(s_f) = 1)
f ← rdivide_=(e, d);
```

Fig. 2. Checking Code In-lined

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

▶ s_a ← ⟨m, n⟩
a ← rand(m, n);
▶ s_b ← ⟨x, y⟩
b ← rand(x, y);

▶ s_c ← s_a ⊛̈ s_b; assert(θ(s_c) = 1)
c ← mtimes(a, b);
▶ s_d ← s_c ⊕̈ s_a; assert(θ(s_d) = 1)
d ← plus(c, a);
e ← minus(d, a);
f ← rdivide_=(e, d);
```

Fig. 3. After CSE and Dead-code Elimination

Note that all of the shape-tuple component arithmetic in the embedded code of Figure 3 can be efficiently mapped by an interpreter or a compiler to a machine's instruction set since they only involve scalar, floating-point calculations.

9 Preallocation

Preallocation is an optimization that can often improve the performance of MATLAB and APL codes. In [12], an improvement by a factor of 4 was observed for the Euler-Cromer program in the FALCON benchmark suite, when this optimization was manually applied. 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 4. Given `for i =` *expr*`, ...; end`, MATLAB executes the loop n times, where n is the number of columns in the MATLAB expression *expr* [10]. 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.)

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

Fig. 4. A MATLAB `for` Loop

```
▶ s_a ← ...; s_b ← ...; s_c ← ...
▶ s_e ← ⟨p_1, p_2, ..., p_k⟩
a ← ...; b ← ...; c ← ...; e ← ...;
▶ A ← 0; C ← 0
▶ for j from 1 to p_2 × ··· × p_k
      s_a ← s_a ⍟ s_b; s_c ← s_a ⍟ s_c
      A ← max(A, |s_a|); C ← max(C, |s_c|)
  endfor
▶ resize a to A elements
▶ resize c to C elements
for i = e,
    a ← [a; b];
    c ← a.*c;
end;
```

Fig. 5. After Preallocation

The first statement in the loop body of Figure 4 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, thereby impacting performance. Instead, we move the shape-tuple computations associated with these two MATLAB statements—which are $\boldsymbol{s_a} \leftarrow \boldsymbol{s_a} \ddot{\circledcirc} \boldsymbol{s_b}$ and $\boldsymbol{s_c} \leftarrow \boldsymbol{s_a} \ddot{\oplus} \boldsymbol{s_c}$ from Table 1—outside the loop. This is shown in Figure 5, 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.

The shape tuples themselves do not arbitrarily grow in size. This is because most Type I built-in functions exhibit an important characteristic known as the *bounded property* [7]: Whenever the ranks of their arguments are bounded by a suitable constant, the ranks of their results will also be bounded by the same constant. Thus, if we were to consider the MATLAB statement `c` $\leftarrow$ φ`(a, b)` where φ is a Type I built-in function that exhibits the bounded property, it will be possible to find a constant $\mathfrak{R}_\varphi$ such that for all $\mathcal{R}(\mathsf{a})$ and $\mathcal{R}(\mathsf{b})$,

$$\mathcal{R}(\mathsf{a}) \leq \mathfrak{R}_\varphi \wedge \mathcal{R}(\mathsf{b}) \leq \mathfrak{R}_\varphi \Longrightarrow \mathcal{R}(\mathsf{c}) \leq \mathfrak{R}_\varphi. \tag{14}$$

The bounded property is crucial 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. Thus, in the case of Figure 4, 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 [7]. Hence, because the substitution property is honored by the corresponding shape-tuple functions, we can perform all the shape-tuple computations in Figure 5 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 many Type I built-in functions, 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 Rose's approach, the framework may permit tighter inferences in specific cases, such as that shown in Example 3.

10 Summary

In this paper, we have described a framework using which the shape of a MATLAB expression can be expressed exactly and succinctly at compile time. The framework covers a large class of built-in functions and reveals and exploits the algebras that underlie each of them. 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 infer shapes in other array-based languages such as APL that share many of MATLAB's features.

References

1. T. Budd. **An APL Compiler**. Springer-Verlag, Inc., 1988. ISBN 0–387–96643–9.
2. S. Chauveau and F. Bodin. "Menhir: An Environment for High Performance MATLAB". In the *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.
3. R. Cytron, J. Ferrante, B. K. Rosen, and M. 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.
4. P. Drakenberg, P. Jacobson, and B. Kågström. "A CONLAB Compiler for a Distributed-Memory Multicomputer". In the *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 814–821, March 1993.
5. L. A. De Rose. "Compiler Techniques for MATLAB Programs". Ph.D. dissertation, University of Illinois at Urbana-Champaign, May 1996.
6. B. C. Jay. "A Semantics for Shape". *Science of Computer Programming*, 25:251–283, 1995.

7. P. G. Joisha, U. N. Shenoy, and P. Banerjee. "An Approach to Array Shape Determination in MATLAB". Technical Report CPDC–TR–2000–10–010, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, October 2000.
8. M. A. Kaplan and J. D. Ullman. "A Scheme for the Automatic Inference of Variable Types". *Journal of the ACM*, 27(1):128–145, January 1980.
9. The MAJIC Project. At `http://polaris.cs.uiuc.edu/majic/majic.html`.
10. The MathWorks, Inc. *MATLAB: The Language of Technical Computing*, January 1997. Using MATLAB (Version 5).
11. The MathWorks—MATLAB Compiler. At `http://www.mathworks.com/products/compiler/index.shtml`.
12. V. Menon and K. Pingali. "A Case for Source-Level Transformations in MATLAB". In the *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 53–65, October 1999.
13. M. J. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. "Preliminary Results from a Parallel MATLAB Compiler". In the *Proceedings of the 12th International Parallel Processing Symposium*, pages 81–87, April 1998.
14. 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.