Implementing an Array Shape Inference System for MATLAB® Using *MATHEMATICA*®

Pramod G. Joisha *Prithviraj Banerjee*

Technical Report No. CPDC–TR–2002–10–003
© 2002 Center for Parallel and Distributed Computing
October 2002

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

Implementing an Array Shape Inference System for MATLAB® Using *MATHEMATICA*® *

Pramod G. Joisha 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, banerjee]@ece.northwestern.edu

Abstract

In implicitly typed array-based languages such as MATLAB, APL and IDL, advance determination of an array's shape empowers the execution of efficient code. Static knowledge of an array's shape could sanction a variety of code optimizations besides permitting compile-time verification of shape correctness and serving as a debugging aid. This report presents a symbolic *array shape inference system for MATLAB, a prototypical array-based language. Unlike prior efforts at array shape determination that have used lattice-theoretic techniques, our methods are based on algebraic systems that exploit properties associated with MATLAB's shape semantics. This confers a unique advantage to our approach in that useful shape information can be deduced even when the actual array extents are compile-time unknowns. The shape inference system has been implemented in Mathematica as part of* MAGICA*, a type inference engine for MATLAB. We report array shape inference measurements on a benchmark suite comprising programs that span the published test suites of some recent research MATLAB compilers, and we show how even when shapes are symbolic, our system detects the equivalence of over 60% of these shapes in 5 out of 6 benchmarks, and over 30% in the sixth. In all remaining situations, all array shapes were inferred to be compile-time constants.*

Keywords array shape determination, computer algebra, Mathematica, MATLAB

1 Introduction

There has been a revival of interest in typeless array-based languages in recent years following the commercial success of a number of problem-solving and visualization environments such as MATLAB and IDL. By melding a lucid syntax with features like implicit typing, multidimensional data structures, real and complex arithmetic, a rich polymorphic function set, and an interactive interpretive execution model, these languages have proven

* *MATHEMATICA*® fonts by Wolfram Research, Inc.

to be ideal tools for an "exploratory" style of program development in which the processes of data examination, manipulation, visualization and incremental code development form a tight, feedback-driven loop.

However, the very same features that have contributed to the popularity of these languages—particularly implicit *array shape* typing and array shape polymorphism—have impeded their compilation, limiting execution scopes to interpretation. Because of this, the last few years have witnessed much activity in the area of array shape determination for such languages since the quality of the generated code is crucially dependent on the kind of shape information available to a compiler. Solutions have ranged from "hand-holding" the compiler through the type determination phase by providing shape declarations in the form of comments, to static inference mechanisms that infer and propagate shape attributes when possible, and that resort to a dynamic interpreter-like strategy otherwise. Static inference mechanisms are attractive due to their automatable nature, but their advantage is currently limited to situations in which the shape attributes are completely determinable at compile time. In situations where shape attributes are unknown, even partially, conservative run-time resolution is relied upon.

As an example, consider the synthetic MATLAB code fragment shown in Figure 1. The symbol ← will be used to denote the assignment operation in MATLAB. The statement `c ← a*b` on line L_4 assigns to program variable `c` the result of `a*b`, where `*` is MATLAB's matrix multiplication operation. If either `a` or `b` is a scalar, the other variable can be an arbitrary array and the result is produced by multiplying the scalar with the elements of the array. On the other hand, if both `a` and `b` are nonscalar, they have to be conforming matrices for the operation to be valid—that is, their shapes must be $p \times q$ and $q \times r$ respectively [Mat97]. In current static shape inference systems for MATLAB [DRP99, QMSZ98, CB98], the shape of `c`, at least on the first iteration, can be inferred if the initial shapes of `a` and `b` are known at compile time. However, if either of these initial shapes are unknown, even partially (some array extents known, others unknown), the shape of `c` will also be considered unknown and conservative code to perform run-time resolution will be generated. Unknown shapes have a propagative effect in that one unknown shape could lead to a whole set of shapes becoming unknown. This could be the case for the code fragment in Figure 1 if it were part of a function in which `a`, `b` and `e` are the formal parameters, and if the function was being analyzed in isolation.

```
L1:  d ← max(b*b, abs(a)*b);
L2:  d ← (b+a)*fix(b);
L3:  while p1,
L4:     c ← a*b;
L5:     a ← c+(a.^b);
L6:     b ← c-(a./c);
L7:     while p2,
L8:        e ← e*d;
        end;
    end;
L9: h ← (b.^b)-a;
```

FIGURE 1: A Motivating Code Fragment

Our premise is that even when shapes are not known at compile time, we can do better if the algebraic properties associated with the language's shape semantics are taken advantage of. For instance, in the case of Figure 1, we shall see that

1. The shapes of `d` on both lines L_1 and L_2 are identical for *any* `a` and `b`. This means that a translator can replace shape conformance checks for lines L_1 and L_2 by a single one for line L_1.

2. At the end of each iteration, the shapes of `a` and `b` on lines L_5 and L_6 will be identical.

3. There can only be three possibilities for the shape of `c`, two for `a` and `b`, and three for `e` during the entire execution lifetime of the loops. We can also arrive at symbolic expressions for these possibilities, which a code generator could use for such tasks as preallocating storage for the arrays.

1.1 Background

As far back as 1974, the potential for performance improvements by statically validating array shape checks was known [BS74]. Early attempts viewed shape determination through the more general problem of automatic type estimation [Ten74, KU78]. In [KU78], a scheme for automatically inferring types in an abstraction of typeless languages such as APL, SNOBOL and SETL, and based on the theory of lattices, was presented. Most subsequent efforts that use lattices to detect some type attribute, be it shape or intrinsic type, appear to be of this flavor [Mil79, Chi86, Bud88, DRP99]. Though [KU78] looked at types in a broad sense, it seems to be most suited to situations in which a language's types can be naturally arranged into a partial order. In particular, types higher up in the lattice hierarchy were *always* considered to be *safe* replacements for types lower down, independent of program context, because the underlying partial order was interpreted to mean value subsumption.[b] "Safeness" means that program correctness is never compromised by such replacements. This makes lattices particularly useful for problems such as *intrinsic type* determination (finding whether a program variable is an integer, a real or a complex number) and they have been successfully used for this purpose both in MATLAB [DRP99] and APL [Chi86, Bud88]. Note that intrinsic type determination is also an issue with MATLAB because, in addition to a structural aspect, its arrays have an independent intrinsic type attribute that defines the arithmetic possible on their contents. For shape determination, it isn't even clear whether a partial order can be defined among individual shapes in the above sense without introducing abstract shapes. Of course, a partial order can always be defined by introducing abstractions that represent *sets* of shapes and this was the approach taken for *approximate* shape determination in [Bud88].

The FALCON compiler used a static inference mechanism that gathered shape information from program constants and that propagated this information using tables [DRP99]. Shapes that were even partially unknown at compile time were handled at run time through the introduction of so-called shadow variables. Both the "Otter" parallel MATLAB compiler [QMSZ98] and the Menhir system [CB98] use similar methods; Menhir additionally relies on user-provided annotations called directives when sufficient type information is lacking.

There has also been work in the bottom-up design of array-based languages that permit a complete static analysis of shape. The FISh language is one such example [JS98]. FISh imposes a number of restrictions to enable this static analysis, some as severe as not allowing the shape of an array to change during execution. These kind of restrictions do not exist in MATLAB. It also isn't clear whether FISh can be used as a production language; the only nontrivial FISh programs on which timings have been reported appear to be the Fast Fourier Transform, quicksort and matrix multiplication [JS98].

2 Overview of Our Approach

Our basic idea is to uncouple the shape semantics of each of the language's operators and to represent them separately by special expressions called *shape-tuple class* (SC) expressions in an intermediate representation of the input program. An SC expression represents a unique array shape. When possible, we evaluate these SC expressions at compile time to arrive at explicit array shapes. Otherwise, the SC expressions remain and get executed at run time as part of the program. In the latter situation, we still take advantage of their algebraic properties so that the run-time overhead associated with their computation is either eliminated or mitigated.

2.1 Uncoupling Shape Semantics

All data in MATLAB is ultimately an array. This remark also applies to scalars, which are considered arrays of size 1×1. We view an array in MATLAB as the value tuple $(\mathcal{T}, \mathcal{D})$ where $\mathcal{T}$ is its type descriptor and $\mathcal{D}$ is its notation. That is, $\mathcal{D}$ is a *depiction* of the array and $\mathcal{T}$ determines the *import* of that depiction to arrive at the array's *value*. An example of a concrete notation would be the sequence of binary digits that stands for the array in computer memory. Another example would be the syntactic phrase used to specify the array. The reason for considering values as tuples of $\mathcal{T}$ and $\mathcal{D}$, rather than as images under some mapping on the

[b] If s and t are two types such that $s \leq t$, then the interpretation was that all values representable by s are also representable by t.

set of notations, is because such a mapping may not be single valued. This becomes clear when we consider a syntactic phrase like `a+b` in MATLAB, which, though a valid depiction of an array in the language, could represent different values depending on the program context.

The type descriptor in turn can be viewed as a tuple $(\mathcal{I}, \mathcal{S}, \dots)$ comprised of type attributes such as the intrinsic type $\mathcal{I}$ and the shape $\mathcal{S}$. We regard the array operations built into the MATLAB language (called built-in functions [Mat97]) as *partial functions* that operate on value tuples and produce value tuples as their result.[c]

Consider a built-in function f that produces a value tuple by operating upon k value tuples:

$$((\mathcal{I}, \mathcal{S}, \dots), \mathcal{D}) \leftarrow f\big(((\mathcal{I}_1, \mathcal{S}_1, \dots), \mathcal{D}_1), ((\mathcal{I}_2, \mathcal{S}_2, \dots), \mathcal{D}_2), \dots, ((\mathcal{I}_k, \mathcal{S}_k, \dots), \mathcal{D}_k)\big).$$

We can think of the above assignment being replaced by a sequence of assignments that calculate each of the type attributes and the array notation separately:

$$\begin{aligned}
\mathcal{I} &\leftarrow f_{\mathcal{I}}\big(((\mathcal{I}_1, \mathcal{S}_1, \dots), \mathcal{D}_1), ((\mathcal{I}_2, \mathcal{S}_2, \dots), \mathcal{D}_2), \dots, ((\mathcal{I}_k, \mathcal{S}_k, \dots), \mathcal{D}_k)\big),\\
\mathcal{S} &\leftarrow f_{\mathcal{S}}\big(((\mathcal{I}_1, \mathcal{S}_1, \dots), \mathcal{D}_1), ((\mathcal{I}_2, \mathcal{S}_2, \dots), \mathcal{D}_2), \dots, ((\mathcal{I}_k, \mathcal{S}_k, \dots), \mathcal{D}_k)\big),\\
&\vdots\\
\mathcal{D} &\leftarrow f_{\mathcal{D}}\big(((\mathcal{I}_1, \mathcal{S}_1, \dots), \mathcal{D}_1), ((\mathcal{I}_2, \mathcal{S}_2, \dots), \mathcal{D}_2), \dots, ((\mathcal{I}_k, \mathcal{S}_k, \dots), \mathcal{D}_k)\big).
\end{aligned}$$

We call this separation *uncoupling* and the shape-tuple class expression mentioned earlier is basically the $\mathcal{S}$ in the second assignment above. As indicated, the *shape-tuple class operator* $f_{\mathcal{S}}$ is completely responsible for calculating shape; for instance, a case of nonconforming shapes would be intercepted in its computation of $\mathcal{S}$. At the very least, $f_{\mathcal{S}}$ could be the original f with everything except $\mathcal{S}$ discarded from the result. The code generation phase sees only these uncoupled assignments. The objective is to either fully evaluate at compile time the uncoupled assignments pertaining to type, or to at least algebraically simplify them, so that the work of computing them at run time is either removed or reduced. In this report, we will only be concerned with $f_{\mathcal{S}}$ and the calculation of $\mathcal{S}$.

2.2 Shape Inference Flow

Input programs are first subject to the SSA transformation [CFRW91] that results in every program variable having exactly one definition. The intermediate representation is then passed to Mathematica by the front-end for shape analysis. Mathematica cleaves the shape semantics from those MATLAB expressions in which the operators are built-in functions and represents them as separate shape-tuple class expressions. Assignments of program constants give rise to constant SC expressions. Loops and conditional statements remain unchanged. To propagate shapes across user-defined functions, user-defined function interfaces are modified to take the SC expressions of arguments as additional parameters. After cleaving, Mathematica evaluates or simplifies as many of the SC expressions as possible. It may also determine a *set* of SC expressions that describe the possible shapes of a MATLAB expression during the entire execution lifetime of the program. The result is passed back to the front-end, which uses the gathered shape information for further optimizations and code generation.

3 Shape-Tuple Class Expressions

The shape tuple $\langle p_1, p_2, \dots, p_m \rangle$ will be used to describe the extents of an m-dimensional MATLAB array in which p_i is the extent along the ith dimension. The number of components m in this notation is always at least 2 because arrays in MATLAB have at least two dimensions.

[c] Programming language functions, of course, aren't the same as the mathematical concept. But we believe that this modelling is a good approximation and sufficiently broad for our purposes. For example, built-in functions that depend on some internal state can be thought of as depending on a value tuple that persists across invocations.

3.1 Mathematical Preliminaries

Let $\mathbb{L}_{\mathbb{S}}$ be the set of all shape tuples whose components belong to the set of nonnegative integers $\mathbb{W}$. We choose $\mathbb{W}$ so as to encompass the empty array construct in MATLAB [Mat97]. We shall often overload the notation $\langle p_1, p_2, \ldots, p_n \rangle$ to mean an $n \times n$ square diagonal matrix in which each p_i $(1 \leq i \leq n)$ is a principal diagonal element. This allows us to bring to bear the machinery of matrix arithmetic when manipulating shape tuples on paper, avoiding the invention of new notation to a large extent. For example, the determinant $|\boldsymbol{s}|$ can be used to express the size of a shape tuple $\boldsymbol{s}$.

3.1.1 Illegal Shape Tuples

To capture shape errors, we include "illegal" shape tuples by considering a set $\mathbb{I}_{\mathbb{S}}$ 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.[d] The canonical illegal shape tuple $\langle \pi_1, \pi_2 \rangle$ will be represented by the symbol $\boldsymbol{\pi}$. The augmented set $\mathbb{S} = \mathbb{L}_{\mathbb{S}} \cup \mathbb{I}_{\mathbb{S}}$ is the universe of shape tuples that we will be dealing with.

3.1.2 Shape-Tuple Equivalence

In MATLAB, any m-dimensional array can always be considered to have n dimensions, where $n > m$, simply by treating higher dimensions to have unit extents. Since higher dimensions are indicated to the right in the shape-tuple notation, shape tuples that only differ in trailing unit extents represent the same shape in MATLAB. Specifically, because all arrays in MATLAB have at least two dimensions, unit extents that trail after the second component can be discarded. Thus, we can define an equivalence relation $\wp$ on $\mathbb{S}$ such that two shape tuples are related by $\wp$ if and only if they represent the same shape. That is, 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

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

If $\overline{\boldsymbol{s}}$ is the equivalence class of a shape tuple $\boldsymbol{s}$, observe that the set of illegal shape tuples $\mathbb{I}_{\mathbb{S}}$ forms an equivalence class $\overline{\boldsymbol{\pi}}$ by this relation. Furthermore, observe that the shape tuple of a MATLAB expression can be any element in some shape-tuple class (i.e., equivalence class) under $\wp$. The simplest such shape tuple (one in which all unnecessary trailing unit extents are cast out) will be referred to as the *canonical* shape tuple of the equivalence class. The set of all shape-tuple classes under $\wp$, called the *quotient set* of $\mathbb{S}$ by $\wp$ (see [TM75]), will be denoted by $\mathbb{S}_{\wp}$. These notions were introduced in [JB01].

3.1.3 Shape-Tuple Algebras

There exists a class of built-in functions in MATLAB, called Type I, for which the shapes of the outputs are completely determined by the shapes of the inputs [JB01]. This class forms the majority in the language and contains such common operations as matrix multiplication and array addition. Most other built-in functions fall into the Type II category and the rest into the Type III group [JSB00]; though our framework and implementation also handle members of the Type II class, algebraic simplifications are currently aimed at only the Type I class.

By definition, for any Type I built-in function $\star$, there is a mapping $\ddot{\star}$ from $\mathbb{S} \times \mathbb{S} \times \cdots \times \mathbb{S}$ to $\mathbb{S}$ that expresses the shape tuple of the built-in function's output in terms of the shape tuples of its inputs. The set $\mathbb{S}$ and the

[d] In this report, π_1 will be set to -1 and π_2 to 1.

n-ary operation $\ddot{\star}$ are therefore said to form the *algebraic system* $[\mathbb{S}, \ddot{\star}]$ (see [TM75]). Algebras like $[\mathbb{S}, \ddot{\star}]$, in which the n-ary function is a shape-tuple operator, will be specifically referred to as *shape-tuple algebras*.

In [JB01], we showed how the shape-tuple algebra of a Type I built-in function can be formally described using systematic matrix arithmetic on the shape tuples of the function's arguments. The motivation for formulating shape-tuple algebras is to have a reference "recipe" for the mechanical computation of shape tuples.[e] In this way, shape-tuple algebras form the basis for precisely expressing the language's shape semantics.

3.1.4 Shape-Tuple Class Algebras

Using shape-tuple algebras, *shape-tuple class* (SC) algebras can be built that operate on shape-tuple classes rather than individual shape tuples [JB01]. Because they describe shape semantics to within shape-tuple equivalence, SC algebras are useful in characterizing the shape-tuple algebras that underlie them.

For example, let $\ddot{\oplus}$ be the shape-tuple operator of the array addition built-in function in MATLAB. Example infix and prefix invocations of this built-in function are `a+b` and `plus(a, b)` respectively. Using the shape-tuple algebra $[\mathbb{S}, \ddot{\oplus}]$, an SC algebra $[\mathbb{S}_{\wp}, \dot{\oplus}]$ can be built that operates on the quotient set $\mathbb{S}_{\wp}$ rather than the shape-tuple set $\mathbb{S}$. For any three shape-tuple classes $\overline{\boldsymbol{s}}$, $\overline{\boldsymbol{t}}$ and $\overline{\boldsymbol{u}}$, $[\mathbb{S}_{\wp}, \dot{\oplus}]$ can be shown to have a number of nice properties [JSB00]:

$$\overline{\boldsymbol{\pi}}\dot{\oplus}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{\pi}} = \overline{\boldsymbol{\pi}}, \quad \text{(Annihilation)} \quad (3)$$

$$\overline{\boldsymbol{\iota}}\dot{\oplus}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{\iota}} = \overline{\boldsymbol{s}}, \quad \text{(Identity)} \quad (4)$$

$$\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}, \quad \text{(Idempotency)} \quad (5)$$

$$\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}} = \overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{s}}, \quad \text{(Commutativity)} \quad (6)$$

$$\overline{\boldsymbol{s}}\dot{\oplus}(\overline{\boldsymbol{t}}\dot{\oplus}\overline{\boldsymbol{u}}) = (\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}})\dot{\oplus}\overline{\boldsymbol{u}} \quad \text{(Associativity)} \quad (7)$$

where $\overline{\boldsymbol{\iota}}$ is the equivalence class of the scalar shape tuple $\boldsymbol{\iota} = \langle 1, 1\rangle$. This means that shape-tuple expressions such as $\boldsymbol{s}\ddot{\oplus}\boldsymbol{t}$ and $\boldsymbol{t}\ddot{\oplus}\boldsymbol{s}$, though not necessarily identical, will still be the same to within shape-tuple equivalence—that is, they will fall into the same shape-tuple class $\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}}$.

The main attraction of SC algebras is that only a few of them suffice to describe MATLAB's shape semantics. For example, there are over 20 built-in functions, spanning a number of common operations, whose SC algebras are isomorphic to $[\mathbb{S}_{\wp}, \dot{\oplus}]$ [JSB00]. In fact, so far, we have been able to uncover nine quotient algebras to which are isomorphic the shape semantics of over 50 Type I built-in functions in MATLAB [JSB00].

3.2 SC Expressions in Mathematica

We begin by issuing the `Off` command to switch off certain diagnostic messages that Mathematica would otherwise emit whenever new symbols are created.[f]

```
In[1]:= Off[General::spell,
        General::spell1]

In[2]:= Overscript[⟨1, 9⟩] // FullForm
```

```
Out[2]//FullForm=
        OverBar[AngleBracket[1, 9]]

In[3]:= Overscript[⟨x_, y_, z___, (1) ..⟩] := Overscript[⟨x, y, z⟩]
```

All expressions in Mathematica are treated in a uniform way. Every expression consists of a *head* and a sequence of arguments, all of which can themselves be arbitrary expressions. For example, when `x+y` is typed as input, Mathematica considers it internally as the expression `Plus[x, y]` in which the head expression is the *function object* `Plus` and the argument expressions are the objects `x` and `y`. When printing out this expression, Mathematica gives it once again as `x+y`. Similarly, an input such as $\overline{e}$, where e is any Mathematica expression, is treated internally as `OverBar[`e`]`. This full form can be seen by using e `// FullForm`, which is the postfix invocation of `FullForm[`e`]`. Thus on line *In[2]* above, the input results in Mathematica outputting the full form of the SC expression $\overline{\langle 1, 9\rangle}$.

We next need to "simplify" SC expressions when possible. This is achieved by the definition on line *In[3]*, which basically says that the equivalence class of a shape tuple having trailing unit extents from the

[e] An implementation is free to use alternate ways to compute a shape tuple so long as the mapping specified by the shape-tuple algebra is honored to within shape-tuple equivalence.

[f] The Mathematica outputs shown in this report can be exactly reproduced by typing the code shown against each of the *In[n]:=* prompts into a notebook interface to version 4.1 of Mathematica. The typeset inputs can be entered *as is* into a notebook using special control sequences [Wol99].

third component onward is the same as the equivalence class of a shape tuple in which these trailing extents are dropped. The definition, made against the `OverBar` object, carries four parameter specifications. These specifications are Mathematica *patterns* that represent entire classes of expressions [Wol99]. While the patterns `x_` and `y_` stand for arbitrary single expressions in Mathematica, the pattern `z___` stands for zero or more such expressions. Repeated occurrences of particular expressions can also be specified—the pattern `(1)..` stands for one or more occurrences of the integer literal `1`.

When evaluating an expression, Mathematica looks up the definitions that it knows against the head of the expression, and determines an appropriate one to apply using pattern matching.

```
In[4]:= ⟨8, 9, 3, 3, 1, 1⟩
Out[4]= ⟨8, 9, 3, 3⟩
```

```
In[5]:= ⟨x, 2, 3⟩
Out[5]= ⟨x, 2, 3⟩
```

```
In[6]:= π = ⟨-1, 1⟩;
In[7]:= ι = ⟨1, 1⟩;
```

For example, on line *In[4]* above, Mathematica determines that the definition on line *In[3]* can be used to evaluate the expression because the pattern $\langle$`x_`, `y_`, `z__`, `(1)..`$\rangle$ matches the expression. In particular, the patterns `x_` and `y_` match the objects `8` and `9`, while the pattern `z__` matches the sequence of objects `3` and `3`. Thus the right-hand side of the definition evaluates to $\langle 8, 9, 3, 3\rangle$. The expression $\langle 8, 9, 3, 3\rangle$, however, cannot be evaluated further because none of the currently known definitions against the `OverBar` object apply—that is, none of them have left-hand sides that match $\langle 8, 9, 3, 3\rangle$. It is for the same reason that the input expression on line *In[5]* remains unchanged.

3.3 Codifying SC Operators in Mathematica

In Mathematica, several attributes can be declared against function objects that affect their treatment during pattern matching and evaluation. For example, when a function object `plusSC` is assigned the attribute `Flat`, as has been done on line *In[8]* below, Mathematica automatically flattens nested expressions like `plusSC[`x`, plusSC[`y`, `z`]]` and `plusSC[plusSC[`x`, `y`], `z`]` to `plusSC[`x`, `y`, `z`]`, thus producing the effect of associativity. Commutativity can be similarly simulated by assigning the `Orderless` attribute that causes expressions to be reduced to a canonical form. Though the influence of these attributes is entirely at a structural level, the concept of structure in Mathematica is a sophisticated one.

```
In[8]:= SetAttributes[plusSC, {Flat,
          Orderless, OneIdentity}]
In[9]:= plusSC[ι, s_:ι] := s
```

```
In[10]:= plusSC[π, _:ι] = π;
In[11]:= plusSC[s_, s_] := s
```

3.3.1 The $\dot{\oplus}$ SC Operator

The definitions against the `plusSC` object on lines *In[9]* through *In[11]* take advantage of these properties while capturing the identity, annihilation and idempotency properties of $\dot{\oplus}$.[g] That is, additional definitions aren't necessary to handle invocations such as `plusSC[t, `$\overline{\iota}$`]` and `plusSC[s, t, `$\overline{\iota}$`]`. This is because the `Orderless` attribute automatically takes care of rearranging `plusSC[t, `$\overline{\iota}$`]` to `plusSC[`$\overline{\iota}$`, t]`, enabling the definition on line *In[9]* to be used for its evaluation. In the case of `plusSC[s, t, `$\overline{\iota}$`]`, the pattern matcher automatically rearranges it to `plusSC[`$\overline{\iota}$`, plusSC[s, t]]` due to the `Flat` and `Orderless` attributes, allowing the definition on line *In[9]* to be once again used for its evaluation.

Example invocations of `plusSC` are shown below. Line *Out[12]* essentially reflects the fact that when we add a 2×3 matrix to a scalar in MATLAB, the result will be a 2×3 matrix. On line *Out[13]*, we see that the result of adding two $4 \times 1 \times 0$ empty arrays in MATLAB is also a $4 \times 1 \times 0$ empty array.

```
In[12]:= plusSC[⟨2, 3, 1⟩, ⟨1, 1⟩]
Out[12]= ⟨2, 3⟩
```

```
In[13]:= plusSC[⟨4, 1, 0⟩, ⟨4, 1, 0⟩]
Out[13]= ⟨4, 1, 0⟩
```

But it is on line *Out[14]* below that we see a more interesting *symbolic* result, and that is the algebraic identity $\overline{s}\dot{\oplus}(\overline{t}\dot{\oplus}\overline{u})\dot{\oplus}\overline{v} = \overline{s}\dot{\oplus}(\overline{v}\dot{\oplus}\overline{u})\dot{\oplus}\overline{t}$.

[g] The pattern `s_:`$\overline{\iota}$ stands for an expression, which when omitted, is taken to have the value $\overline{\iota}$ [Wol99].

```
In[14]:= plusSC[s, plusSC[t, u], v] === plusSC[s, plusSC[v, u], t]
Out[14]= True
```

Additional definitions can be provided to catch cases of nonconforming shapes. For example, the definition on line *In[15]* below will be considered by Mathematica only after earlier definitions provided against the `plusSC` object are considered and don't apply. Thus, given the expression `plusSC[s, t]` where `s` and `t` are two *explicit* SC expressions, the definition on line *In[15]* will apply only if the definitions given on lines *In[9]* through *In[11]* don't apply. But since that would mean `s` and `t` represent two explicit array shapes that don't conform under MATLAB's array addition rules, the resulting shape-tuple class would have to be $\overline{\boldsymbol{\pi}}$. An example of this situation is shown on line *Out[16]*, which basically corresponds to the illegal attempt of adding a 7×4 matrix to a 7×5 matrix in MATLAB.

```
In[15]:= plusSC[⟨_Integer, __Integer⟩, ⟨_Integer, __Integer⟩] = π̄;
```

```
In[16]:= plusSC[⟨7, 4⟩, ⟨7, 5⟩]
Out[16]= ⟨-1, 1⟩
```

Line *In[15]* also illustrates how Mathematica definitions can be set up to map explicit shapes to an explicit shape.

3.3.2 The $\dot{\circledast}$ SC Operator

Another SC algebra is $[\mathbb{S}_{\wp}, \dot{\circledast}]$, which describes the shape semantics of `*`, MATLAB's matrix multiplication operation. The SC operator $\dot{\circledast}$ is unlike $\dot{+}$ in a number of respects: for instance, idempotency doesn't hold because two identically shaped arrays may not conform under MATLAB's matrix multiplication rules, and properties such as commutativity and associativity don't hold. However, $\dot{\circledast}$ does satisfy the identity and annihilation properties [JSB00]. These properties are coded against the `mtimesSC` object on lines *In[17]* to *In[19]*.

```
In[17]:= mtimesSC[ῑ, s_] := s
In[18]:= mtimesSC[s_, ῑ] := s
```

```
In[19]:= mtimesSC[___, π̄, ___] = π̄;
In[20]:= mtimesSC[⟨p_, q_⟩, ⟨q_, r_⟩] := ⟨p, r⟩
```

Line *In[20]* expresses the fact that whenever we matrix multiply a $p \times q$ matrix by a $q \times r$ matrix in MATLAB, the result will *always* be a $p \times r$ matrix.

3.3.3 Algebraic Simplifications

By using the algebraic properties of SC operators, it may often be possible to simplify SC expressions. For example, two identically shaped arrays conform to the rules of matrix multiplication in MATLAB only when their shapes are of the form $p \times p$. In that case, the product's shape is also $p \times p$ [Mat97]. Thus, for any SC expression $\overline{\boldsymbol{s}}$, the SC expression $\overline{\boldsymbol{s}} \dot{\circledast} \overline{\boldsymbol{s}}$ can be either $\overline{\boldsymbol{s}}$ or $\overline{\boldsymbol{\pi}}$. From Equations (3) and (5), this means that the following identity holds:

$$\overline{\boldsymbol{s}} \dot{+} (\overline{\boldsymbol{s}} \dot{\circledast} \overline{\boldsymbol{s}}) = \overline{\boldsymbol{s}} \dot{\circledast} \overline{\boldsymbol{s}}. \tag{8}$$

We shall adopt the convention that $\dot{\circledast}$ has a higher precedence than $\dot{+}$; this will make the use of parentheses unnecessary whenever operators can be bound according to precedence.

A number of other identities involving $\dot{\circledast}$ and $\dot{+}$ can be shown to hold. Each input line, from *In[21]* to *In[32]* below, codifies one such identity; only 12 identities have been coded because they are enough for the example in Figure 1. The proofs of these identities follow.

```
In[21]:= plusSC[s_,
         t:mtimesSC[s_, s_]] := t
In[22]:= mtimesSC[t:mtimesSC[s_,
         s_], t_] := t
In[23]:= mtimesSC[plusSC[s_, t_],
         plusSC[s_, t_]] :=
         plusSC[mtimesSC[s, s],
         mtimesSC[t, t]]
In[24]:= mtimesSC[plusSC[mtimesSC[
         s_, t_], s_, t_], plusSC[
         mtimesSC[s_, t_], s_,
         t_]] := plusSC[mtimesSC[s,
         s], mtimesSC[t, t]]
In[25]:= mtimesSC[plusSC[s_, t_],
         t_] := plusSC[mtimesSC[t,
         t], s]
In[26]:= mtimesSC[s_, plusSC[s_,
         t_]] := plusSC[mtimesSC[s,
         s], t]
```

```
In[27]:= plusSC[u:mtimesSC[s_, s_],
         mtimesSC[s_, t_]] :=
         plusSC[u, t]
In[28]:= plusSC[mtimesSC[s_, t_],
         u:mtimesSC[t_, t_]] :=
         plusSC[s, u]
In[29]:= mtimesSC[plusSC[u:mtimesSC[
         s_, s_], t_], plusSC[
         v:mtimesSC[t_, t_], s_]] :=
         plusSC[u, v]
In[30]:= mtimesSC[plusSC[u:mtimesSC[
         t_, t_], s_], t_] :=
         plusSC[s, u]
In[31]:= mtimesSC[s_, plusSC[
         u:mtimesSC[s_, s_], t_]] :=
         plusSC[u, t]
In[32]:= mtimesSC[u:mtimesSC[
         mtimesSC[mtimesSC[s_, t_],
         t_], t_], t_] := u
```

Lemma 1

$$(\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\oplus}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}. \tag{1.1}$$

Proof. When $\overline{s}$ is $\overline{\iota}$, both sides of Equation (1.1) become $\overline{t}\dot{\circledast}\overline{t}$. The same is true when $\overline{t}$ is $\overline{\iota}$, and when $\overline{s} = \overline{t}$. So let $\overline{s} \neq \overline{\iota}$, $\overline{t} \neq \overline{\iota}$ and $\overline{s} \neq \overline{t}$. Then

$$(\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\oplus}\overline{t}) = \overline{\pi}\dot{\oplus}\overline{\pi} = \overline{\pi}. \tag{$\mathfrak{P}$1.1}$$

Now, $\overline{s}\dot{\circledast}\overline{s}$ is either $\overline{s}$ or $\overline{\pi}$ for any $\overline{s}$. Similarly, $\overline{t}\dot{\circledast}\overline{t}$ is either $\overline{t}$ or $\overline{\pi}$ for any $\overline{t}$. Therefore, $\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$ will be either $\overline{\pi}$ or $\overline{s}\dot{\oplus}\overline{t}$. But when $\overline{s} \neq \overline{\iota}$, $\overline{t} \neq \overline{\iota}$ and $\overline{s} \neq \overline{t}$, $\overline{s}\dot{\oplus}\overline{t}$ will also be $\overline{\pi}$. Thus

$$\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t} = \overline{\pi} \tag{$\mathfrak{P}$1.2}$$

whenever $\overline{s} \neq \overline{\iota}$, $\overline{t} \neq \overline{\iota}$ and $\overline{s} \neq \overline{t}$. Hence, $(\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\oplus}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$ is always true. □

Lemma 2

$$(\overline{s}\dot{\circledast}\overline{s})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{s}) = \overline{s}\dot{\circledast}\overline{s}, \tag{2.1}$$

$$\overline{s}\dot{\circledast}(\overline{s}\dot{\circledast}\overline{s}) = \overline{s}\dot{\circledast}\overline{s}, \tag{2.2}$$

$$(\overline{s}\dot{\circledast}\overline{s})\dot{\circledast}\overline{s} = \overline{s}\dot{\circledast}\overline{s}. \tag{2.3}$$

Proof. Because $\overline{s}\dot{\circledast}\overline{s}$ is either $\overline{s}$ or $\overline{\pi}$, $(\overline{s}\dot{\circledast}\overline{s})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{s})$ is always $\overline{s}\dot{\circledast}\overline{s}$. Equations (2.2) and (2.3) also follow by similar arguments. □

Lemma 3

$$(\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}. \tag{3.1}$$

Proof. When $\overline{s} = \overline{\iota}$, or $\overline{t} = \overline{\iota}$, the result trivially follows. When $\overline{s} = \overline{t}$, the left-hand side of Equation (3.1) becomes

$$(\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{s})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{s}),$$

which, by Equation (8), becomes $(\overline{s}\dot{\circledast}\overline{s})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{s})$. This in turn simplifies to $\overline{s}\dot{\circledast}\overline{s}$ by Equation (2.1). But when $\overline{s} = \overline{t}$, the right-hand side of Equation (3.1) is also $\overline{s}\dot{\circledast}\overline{s}$. Hence

$$(\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$$

is also true in this case.

Now let $\overline{s} \neq \overline{\iota}$, $\overline{t} \neq \overline{\iota}$ and $\overline{s} \neq \overline{t}$. Then

$$(\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}) = (\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{\pi})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{\pi}) = \overline{\pi}\dot{\circledast}\overline{\pi} = \overline{\pi}.$$

By Equation ($\mathfrak{P}1.2$), the right-hand side of Equation (3.1) is also $\overline{\pi}$ in this case. Hence, $(\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$ is always true. □

Lemma 4

$$(\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}\overline{t} = \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}, \tag{4.1}$$

$$\overline{s}\dot{\circledast}(\overline{s}\dot{\oplus}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}. \tag{4.2}$$

Proof. When $\overline{s} = \overline{\iota}$, or $\overline{t} = \overline{\iota}$, the result trivially follows. When $\overline{s} = \overline{t}$, the right-hand side of Equation (4.1) becomes

$$\overline{s}\dot{\oplus}\overline{s}\dot{\circledast}\overline{s}$$

which, by Equation (8), becomes $\overline{s}\dot{\circledast}\overline{s}$. But when $\overline{s} = \overline{t}$, the left-hand side of Equation (4.1) is also $\overline{s}\dot{\circledast}\overline{s}$. Therefore,

$$(\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}\overline{t} = \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$$

is also true in this case.

When $\overline{s} \neq \overline{\iota}$, $\overline{t} \neq \overline{\iota}$, and $\overline{s} \neq \overline{t}$, the left-hand side of Equation (4.1) becomes $\overline{\pi}$. The right-hand side becomes $\overline{\pi}$ or $\overline{s}\dot{\oplus}\overline{t}$. But since $\overline{s} \neq \overline{\iota}$, $\overline{t} \neq \overline{\iota}$, and $\overline{s} \neq \overline{t}$, $\overline{s}\dot{\oplus}\overline{t}$ is also $\overline{\pi}$. Therefore $(\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}\overline{t} = \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$ is always true. Equation (4.2) can be similarly proved. □

Lemma 5

$$\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{s}\dot{\circledast}\overline{t} = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \tag{5.1}$$

$$\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t} = \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}. \tag{5.2}$$

Proof. When $\overline{s} = \overline{\iota}$, or $\overline{t} = \overline{\iota}$, the result trivially follows. When $\overline{s} = \overline{t}$, the result follows by Equation (8).

Now let $\overline{s} \neq \overline{\iota}$, $\overline{t} \neq \overline{\iota}$ and $\overline{s} \neq \overline{t}$. Then both sides of Equation (5.1) become $\overline{\pi}$ so that $\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{s}\dot{\circledast}\overline{t} = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}$ is always true. Equation (5.2) can be similarly shown to hold. □

Lemma 6

$$(\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}. \tag{6.1}$$

Proof. When $\overline{s} = \overline{\iota}$, or $\overline{t} = \overline{\iota}$, the result trivially follows. When $\overline{s} = \overline{t}$, the result follows by Equations (8) and (2.1).

Now let $\overline{s} \neq \overline{\iota}$, $\overline{t} \neq \overline{\iota}$ and $\overline{s} \neq \overline{t}$. Then both sides of Equation (6.1) become $\overline{\pi}$ so that $(\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}$ is always true. □

Lemma 7

$$(\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}. \tag{7.1}$$

Proof. When $\overline{s} = \overline{\iota}$, or $\overline{t} = \overline{\iota}$, the result follows by Equations (2.2), (2.3) and (8). When $\overline{s} = \overline{t}$, the result follows by Equations (8) and (2.1).

Now let $\overline{s} \neq \overline{\iota}$, $\overline{t} \neq \overline{\iota}$ and $\overline{s} \neq \overline{t}$. Then both sides of Equation (7.1) become $\overline{\pi}$ so that $(\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t})\dot{\circledast}(\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}$ is always true. □

Lemma 8

$$\overline{s}\dot{\circledast}(\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}) = \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \tag{8.1}$$

$$(\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t})\dot{\circledast}\overline{t} = \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}. \tag{8.2}$$

Proof. Consider Equation (8.1). When $\overline{s} = \overline{\iota}$, the result follows trivially. When $\overline{t} = \overline{\iota}$, the result follows by Equation (2.2). When $\overline{s} = \overline{t}$, the result follows by Equations (8) and (2.2).

Now let $\overline{\boldsymbol{s}} \neq \overline{\boldsymbol{\iota}}$, $\overline{\boldsymbol{t}} \neq \overline{\boldsymbol{\iota}}$ and $\overline{\boldsymbol{s}} \neq \overline{\boldsymbol{t}}$. Then both sides of Equation (8.1) become $\overline{\boldsymbol{\pi}}$ so that $\overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}}) = \overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{s}}\dot{\oplus}\overline{\boldsymbol{t}}$ is always true. We can similarly prove Equation (8.2). □

Lemma 9

$$\overline{\boldsymbol{s}}\dot{\circledast}\big(\overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{t}}))\big) = \overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{t}})), \tag{9.1}$$

$$\big(((\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{t}})\dot{\circledast}\overline{\boldsymbol{t}})\dot{\circledast}\overline{\boldsymbol{t}}\big)\dot{\circledast}\overline{\boldsymbol{t}} = ((\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{t}})\dot{\circledast}\overline{\boldsymbol{t}})\dot{\circledast}\overline{\boldsymbol{t}}. \tag{9.2}$$

Proof. Consider Equation (9.1). When $\overline{\boldsymbol{s}} = \overline{\boldsymbol{\iota}}$, or $\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{t}} = \overline{\boldsymbol{\pi}}$, the result follows trivially. When $\overline{\boldsymbol{t}} = \overline{\boldsymbol{\iota}}$, the result follows by Equation (2.2). The only remaining possibility is $\overline{\boldsymbol{s}}$ and $\overline{\boldsymbol{t}}$ being of the form $\overline{\langle p, q\rangle}$ and $\overline{\langle q, r\rangle}$ respectively. Then $\overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{t}}))$ would have to be either $\overline{\boldsymbol{\pi}}$, or $\overline{\langle q, r\rangle}$ if $p = q$. But this would mean $\overline{\boldsymbol{s}}\dot{\circledast}\big(\overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{t}}))\big) = \overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}(\overline{\boldsymbol{s}}\dot{\circledast}\overline{\boldsymbol{t}}))$. Equation (9.2) can be similarly proved.

A point to note here is that the number of applications of $\dot{\circledast}$ in both Equations (9.1) and (9.2) is significant. For instance, $\overline{\langle 1,2\rangle}\dot{\circledast}(\overline{\langle 1,2\rangle}\dot{\circledast}(\overline{\langle 1,2\rangle}\dot{\circledast}\overline{\langle 2,1\rangle})) = \overline{\boldsymbol{\pi}} \neq \overline{\langle 1,2\rangle}\dot{\circledast}(\overline{\langle 1,2\rangle})\dot{\circledast}\overline{\langle 2,1\rangle}) = \overline{\langle 1,2\rangle}$. □

4 Join Nodes

Consider a join node `c` $\leftarrow \phi$`(a, b)` in the SSA form of a MATLAB program. We can characterize the shape-tuple class $\overline{\boldsymbol{u}}$ of `c`, given the shape-tuple classes $\overline{\boldsymbol{s}}$ and $\overline{\boldsymbol{t}}$ of `a` and `b`, as

$$\overline{\boldsymbol{u}} = \dot{\Phi}_P(\overline{\boldsymbol{s}}, \overline{\boldsymbol{t}}) = \begin{cases} \overline{\boldsymbol{s}} & \text{if } P = 0, \\ \overline{\boldsymbol{t}} & \text{if } P = 1. \end{cases} \tag{9}$$

In the above, P 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 run-time selector 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. Note that this modelling of the ϕ function is only for the purpose of analysis at the intermediate stages because the final transformed code will be free of ϕ functions.

4.1 ϕ_μ and ϕ_ν Functions

Because of the structured nature of control flow in the MATLAB language (conditional statements, structured loops, and no `goto` statements), join nodes in the SSA form will always have lexically preceding definitions, except for loop-header join nodes for which one definition will be lexically preceding and the other will be lexically succeeding. We distinguish the two kinds of ϕ functions by referring to loop-header ϕ functions as ϕ_μ functions and ϕ functions that occur at the end of loops and conditional statements as ϕ_ν functions.

4.2 Representing Join Nodes in Mathematica

Every kind of construct in MATLAB can be represented symbolically in Mathematica. For example, a MATLAB expression like `a+b`, where `a` and `b` are MATLAB variables, can be represented in Mathematica by the expression `plus[a, b]` where `a` and `b` are Mathematica symbols. A ϕ_μ function can be denoted by the Mathematica expression `phi`μ`[P][a, b]` where `P` is the ϕ_μ function's run-time selector. The MATLAB assignment statement `c` $\leftarrow e$ can be represented as `LeftArrow[c, e]`, which in turn can be directly entered as `c` $\leftarrow e$ into Mathematica. Thus, a join node like `c` $\leftarrow \phi_\mu$`(a, b)` in a MATLAB CFG can be denoted by the Mathematica expression `c` $\leftarrow$ `phi`μ`[P][a, b]`. Higher-level MATLAB constructs such as `while` loops can also be represented; in this report, we shall use the Mathematica expression `while[{`s_1`, `s_2`, ..., `s_n`}]` to stand for a MATLAB `while` loop whose body consists of a sequence of statements.[h] On line *In[33]* below, we show how the SSA form of the MATLAB program in Figure 1 can be entered into Mathematica.

[h] A `while` loop's test expression isn't represented because it isn't used in this report.

```
In[33]:= stmts := {d1 ← max[mtimes[b0, b0], mtimes[abs[a0], b0]], d2 ←
           mtimes[plus[b0, a0], fix[b0]], while[{a1 ← phiμ[P][a0, a2],
           b1 ← phiμ[P][b0, b2], c ← mtimes[a1, b1], a2 ← plus[c,
           power[a1, b1]], b2 ← minus[c, rdivide[a2, c]], while[{e1 ←
           phiμ[Q][e0, e2], e2 ← mtimes[e1, d2]}], a3 ← phiν[R][a0, a2],
           b3 ← phiν[R][b0, b2], h ← minus[power[b3, b3], a3]}]}
```

Observe the use of the *common* run-time selector `P` among the join nodes in the loop header of the outer `while` loop, the use of the run-time selector `Q` in the single join node of the inner `while` loop, and the use of the run-time selector `R` among the join nodes that come at the end of the outer `while` loop.

5 Uncoupling Array Shape Semantics

On lines *In[34]* to *In[41]* below are definitions against the Mathematica object σ that is used to uncouple the shape semantics from a MATLAB expression.

```
In[34]:= σ[mtimes[x_, y_]] :=
           mtimesSC[σ[x], σ[y]]

In[35]:= σ[plus[x_, y_]] :=
           plusSC[σ[x], σ[y]]

In[36]:= σ[minus[x_, y_]] :=
           plusSC[σ[x], σ[y]]

In[37]:= σ[power[x_, y_]] :=
           plusSC[σ[x], σ[y]]
```

```
In[38]:= σ[rdivide[x_, y_]] :=
           plusSC[σ[x], σ[y]]

In[39]:= σ[max[x_, y_]] :=
           plusSC[σ[x], σ[y]]

In[40]:= σ[abs[x_]] := σ[x];
           σ[fix[x_]] := σ[x]

In[41]:= σ[(phiμ|phiν)[P_][x_, y_]] :=
           phiSC[P][σ[x], σ[y]]
```

As an example, the definition on line *In[34]* is responsible for cleaving the shape semantics from a matrix multiplication operation in MATLAB. In this case, the uncoupled shape semantics is represented by the SC expression `mtimesSC[σ[x], σ[y]]`, where `σ[x]` and `σ[y]` are the SC expressions of the MATLAB expressions that correspond to the `x_` and `y_` patterns. Because the array addition (`plus`), array subtraction (`minus`), array power (`power`), right array division (`rdivide`) and `max` built-in functions in MATLAB have SC algebras that are isomorphic to $[\mathbb{S}_{\wp}, \dot{\oplus}]$, their SC expressions have the same head `plusSC`. In the case of the built-in functions `abs` and `fix`, the SC operators are identity mappings. Line *In[41]* maps both ϕ_μ and ϕ_ν functions to the same SC expression; if there is a need to distinguish between ϕ_μ and ϕ_ν functions at the level of SC expressions, different Mathematica objects can be used to represent their SC expression heads.

5.1 Uncoupling Shape Semantics from MATLAB Statements

The definitions on lines *In[42]* to *In[44]* below generate SC expressions from MATLAB assignment statements, statement lists and `while` loops respectively.

```
In[42]:= σ[x_ ← y_] :=
           σ[x] → σ[y]
```

```
In[43]:= σ[s_List] :=
           Flatten[σ /@ s]
```

```
In[44]:= σ[while[b_List]
           ] := σ[b]
```

The definition on line *In[42]* generates an SC expression *rewriting rule* [Wol99] from a MATLAB assignment statement. That is, given the assignment `c` $\leftarrow e$, it generates the rewriting rule `σ[c]` $\rightarrow$ `σ[`e`]` that indicates the SC expression of `c` to be `σ[`e`]`. Rewriting rules were chosen because they can be conveniently used for further manipulations. The Mathematica expression g `/@` $\{e_1,\ e_2,\ \ldots\}$ applies the Mathematica expression g on each element in a list, producing $\{g[e_1],\ g[e_2],\ \ldots\}$ as the result; this is how σ is applied on a sequence of MATLAB statements on line *In[43]*. The `Flatten` built-in Mathematica object [Wol99] flattens out nested lists that may arise in σ `/@ s`. Finally, line *In[45]* below shows an application of σ on the Mathematica representation of the MATLAB program in Figure 1.

```
In[45]:= scs = σ[stmts]

Out[45]= {σ[d1] → plusSC[mtimesSC[σ[b0], σ[b0]], σ[a0]],
           σ[d2] → plusSC[mtimesSC[σ[b0], σ[b0]], σ[a0]],
           σ[a1] → phiSC[P][σ[a0], σ[a2]],
           σ[b1] → phiSC[P][σ[b0], σ[b2]], σ[c] → mtimesSC[σ[a1], σ[b1]],
           σ[a2] → plusSC[σ[a1], σ[b1], σ[c]],
           σ[b2] → plusSC[σ[a2], σ[c]], σ[e1] → phiSC[Q][σ[e0], σ[e2]],
           σ[e2] → mtimesSC[σ[e1], σ[d2]], σ[a3] → phiSC[R][σ[a0], σ[a2]],
           σ[b3] → phiSC[R][σ[b0], σ[b2]], σ[h] → plusSC[σ[a3], σ[b3]]}
```

5.2 Removing Redundant Array Shape Computations

On examining line *Out[45]* above, we can immediately see that the SC expressions of `d1` and `d2` are the same. This corresponds to Inference 1 on page 2 and happens because of a number of "behind the scenes" transformations that Mathematica applies during the evaluation process, due to the definitions that were made earlier against the various objects that occur in the SC expressions. In particular, the identities on lines *In[28]* and *In[25]* in § 3.3.3, along with the `Orderless` attribute, cause the SC expressions of `d1` and `d2` to be reduced to the same form.

5.2.1 Forward Substitution

By using traditional compiler techniques like forward substitution, it may be possible to expose other identical SC expressions.

```
In[46]:= fsub[{x___, l_ → r_}] := Flatten[{#, l → (r /.
         #)}]&[fsub[{x}]]; fsub[{}] = {};
```

For instance, the recursive Mathematica function object `fsub` defined above takes an arbitrary list of SC expression rewriting rules and forward substitutes every definition into its lexically succeeding use. An SC expression that substitutes a use is itself forward substituted prior to the substitution. Before showing the result of applying `fsub` on the output on line *Out[45]*, we make three special formatting definitions against the `mtimesSC`, `plusSC` and `phiSC` objects that only affect the way expressions involving these objects are typeset and displayed. This will make reading of the outputs easier.

```
In[47]:= mtimesSC /: MakeBoxes[mtimesSC[x_, y_], StandardForm] :=
         Module[{a, b, c, d},
             {a, b} = If[Head[x] === mtimesSC ∨ Head[x] === plusSC,
                       {"(", ")"}, {{}, {}}];
             {c, d} = If[Head[y] === mtimesSC ∨ Head[y] === plusSC,
                       {"(", ")"}, {{}, {}}];
             RowBox[Flatten[{a, MakeBoxes[x, StandardForm], b,
                 RowBox[{OverscriptBox["○", "."],
                 AdjustmentBox[StyleBox["*", FontFamily → "Times"],
                 BoxMargins → {{-0.81, 0}, {0, 0}}, BoxBaselineShift →
                 -0.125]}], c, MakeBoxes[y, StandardForm], d}]]]

In[48]:= plusSC /: MakeBoxes[plusSC[x_, y_], StandardForm] :=
         RowBox[{MakeBoxes[x, StandardForm], OverscriptBox["⊕", "."],
         MakeBoxes[y, StandardForm]}]

In[49]:= phiSC /: MakeBoxes[phiSC[P_][x_, y_], StandardForm] :=
         RowBox[{SubscriptBox[OverscriptBox["Φ", "."],
         MakeBoxes[P, StandardForm]], "(", MakeBoxes[x, StandardForm],
         ",", MakeBoxes[y, StandardForm], ")"}]
```

The Mathematica expression e `// Timing` produces the result $\{t',\ e'\}$, where t' is the time taken by the Mathematica kernel to evaluate e to e'.[i]

```
In[50]:= fsub[scs] // Timing

Out[50]= {0. Second, {σ[d1] → σ[b0] ⊛σ[b0] ⊕σ[a0], σ[d2] → σ[b0] ⊛σ[b0] ⊕σ[a0],
  σ[a1] → Φ̇P (σ[a0], σ[a2]), σ[b1] → Φ̇P (σ[b0], σ[b2]),
  σ[c] → Φ̇P (σ[a0], σ[a2]) ⊛Φ̇P (σ[b0], σ[b2]), σ[a2] →
   Φ̇P (σ[a0], σ[a2]) ⊛Φ̇P (σ[b0], σ[b2]) ⊕Φ̇P (σ[a0], σ[a2]) ⊕Φ̇P (σ[b0], σ[b2]),
  σ[b2] → Φ̇P (σ[a0], σ[a2]) ⊛Φ̇P (σ[b0], σ[b2]) ⊕
    Φ̇P (σ[a0], σ[a2]) ⊕Φ̇P (σ[b0], σ[b2]), σ[e1] → Φ̇Q (σ[e0], σ[e2]),
  σ[e2] → Φ̇Q (σ[e0], σ[e2]) ⊛ (σ[b0] ⊛σ[b0] ⊕σ[a0]),
  σ[a3] → Φ̇R (σ[a0], Φ̇P (σ[a0], σ[a2]) ⊛Φ̇P (σ[b0], σ[b2]) ⊕
     Φ̇P (σ[a0], σ[a2]) ⊕Φ̇P (σ[b0], σ[b2])),
  σ[b3] → Φ̇R (σ[b0], Φ̇P (σ[a0], σ[a2]) ⊛Φ̇P (σ[b0], σ[b2]) ⊕
     Φ̇P (σ[a0], σ[a2]) ⊕Φ̇P (σ[b0], σ[b2])),
  σ[h] → Φ̇R (σ[a0], Φ̇P (σ[a0], σ[a2]) ⊛Φ̇P (σ[b0], σ[b2]) ⊕
      Φ̇P (σ[a0], σ[a2]) ⊕Φ̇P (σ[b0], σ[b2])) ⊕Φ̇R (σ[b0], Φ̇P (σ[a0], σ[a2])
       ⊛Φ̇P (σ[b0], σ[b2]) ⊕Φ̇P (σ[a0], σ[a2]) ⊕Φ̇P (σ[b0], σ[b2]))}}
```

[i] The shown timings are on a 440 MHz UltraSPARC-IIi running Solaris 7 and having 128MB of main memory.

5.2.2 Common SC Expression Elimination

On inspecting line *Out[50]* above, we can see that the SC expressions of `a2` and `b2` are also the same. This corresponds to Inference 2 on page 2. A common-subexpression elimination pass, either in the front-end or in Mathematica itself, can now be applied on these SC expressions so that `b2` and `d2` use the same SC expressions as `a2` and `d1` respectively. In this way, the overall overhead due to array shape computation in the translated code can be reduced. Moreover, since successful shape checks on σ`[d1]` and σ`[a2]` would respectively imply the success of shape checks on σ`[d2]` and σ`[b2]`, the latter shape checks become redundant. This fact can also be taken advantage of by a translator.

6 Symbolic Fixed-Point Solutions

Let $2^{\mathbb{S}_\wp}$ denote the power set of $\mathbb{S}_\wp$, and let $\dot{\star}$ be the SC operator of $\star$, a binary Type I MATLAB function. We construct the *set-of-shapes* (SS) algebra $[2^{\mathbb{S}_\wp}, \breve{\star}]$ by considering the mapping $\breve{\star} : 2^{\mathbb{S}_\wp} \times 2^{\mathbb{S}_\wp} \mapsto 2^{\mathbb{S}_\wp}$ defined as

$$S \breve{\star} T = \{\overline{\boldsymbol{u}} | \exists \overline{\boldsymbol{s}} \in S \wedge \exists \overline{\boldsymbol{t}} \in T \text{ such that } \overline{\boldsymbol{u}} = \overline{\boldsymbol{s}} \dot{\star} \overline{\boldsymbol{t}}\}. \tag{10}$$

SS algebras are important because they provide us with a way to conservatively determine the set of possible shapes of a given MATLAB expression. They are therefore the "broadest" of the shape-related algebras because they operate on *sets* of SC expressions rather than individual SC expressions as is the case for SC algebras.

On the set $2^{\mathbb{S}_\wp}$, a lattice $\mathcal{L}$ can be defined in which the set subset relation $\subseteq$ is the partial order, $\emptyset$ is the least element, $\mathbb{S}_\wp$ is the greatest element, and set union $\cup$ and set intersection $\cap$ are the join and meet operations. From the definition in Equation (10), it is easy to see that the SS operator of *every* Type I MATLAB function will be monotonic with respect to $\mathcal{L}$.

6.1 Building SS Expressions

A set-of-shapes expression, defined on the domain $2^{\mathbb{S}_\wp}$, can be denoted in Mathematica as `set[`s_1`, `s_2`, ..., `s_n`]` where each s_i $(1 \leq i \leq n)$ is an SC expression. Lines *In[51]* and *In[52]* below show how the lattice join of SS expressions can be defined.

```
In[51]:= join[___, Sφ, ___] = Sφ;
```

```
In[52]:= join[x__set] := Union[x]
```

While line *In[51]* reflects the fact that a join with the greatest element of a lattice is always the greatest element, line *In[52]* states that the join of other SS expressions that have the object `set` as their heads is simply the union of their representative `set` expressions. Note that $\mathbb{S}_\wp$ is used in a purely symbolic manner on line *In[51]*.

A Mathematica object Σ is defined to generate SS expressions from MATLAB expressions, assignment statements, `while` loops and statement lists. In the case of ϕ functions, the SS expression is set to the join of the SS expressions of the ϕ function's arguments; this is shown on line *In[53]* below.

```
In[53]:= Σ[(_phiμ|_phiν)[x_, y_]] := join[Σ[x], Σ[y]]
```

```
In[54]:= Σ[s_List] := Flatten[Σ /@ s]
```

```
In[55]:= Σ[(abs|fix)[x_]] := Σ[x]
```

For unary MATLAB built-in functions like `abs` and `fix` whose SC operators are identity mappings, the SS expression of the output is simply the SS expression of the input; this is shown on line *In[55]* above.

Given the MATLAB assignment `c` $\leftarrow e$, the definition on line *In[56]* generates the SS expression rewriting rule `c` $\rightarrow \Sigma$`[`e`]` that indicates the SS expression of `c` to be Σ`[`e`]`. The definition also records this SS expression as the *upvalue* [Wol99] of `c` so that subsequent instances of Σ`[c]` expand to Σ`[`e`]`.

```
In[56]:= Σ[x_ ← y_] := (Σ[x] ^:= #; x → #)&[Σ[y]]
```

The definition on line *In[57]* below generates an SS expression from a MATLAB expression of the form $h(x, y)$ where h is any of `mtimes`, `plus`, `minus`, `power`, `rdivide` and `max`. It does this by first constructing $\Sigma(x)$ and $\Sigma(y)$, the SS expressions of x and y. If either of these SS expressions is $\mathbb{S}_\wp$, the greatest element of $\mathfrak{L}$, then $\mathbb{S}_\wp$ is returned. Otherwise, the set of products of every SC expression in $\Sigma(x)$ with every SC expression in $\Sigma(y)$ is formed, with the product operator being the SC operator for h.

```
In[57]:= Σ[(h_)[x_, y_]] := If[#1 === 𝕊℘ ∨ #2 === 𝕊℘,
         𝕊℘, Union[Flatten[Outer[Σ[h], #1, #2]]]]&[Σ[x], Σ[y]]
In[58]:= Σ[mtimes] = mtimesSC; Σ[plus|minus|power|rdivide|max] =
         plusSC;
```

6.2 Preventing Exponential Blow-Up

If $\Sigma(x)$ and $\Sigma(y)$ have m and n SC expressions each, their Cartesian product will have mn SC expressions. However, because of algebraic properties like those discussed in § 3.3.3, a number of these SC expressions may simplify to the same SC expression. The `Union` built-in Mathematica object takes care of eliminating such repetitions.

There also exists another kind of simplification that arises from *set-of-shapes identities*. An example of such an identity is

$$\{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}, \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\} = \{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\}.$$

The reason for the above identity is that for a given pair of SC expressions $\overline{s}$ and $\overline{t}$, $\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$ will always be either $\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}$ or $\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$. Lines *In[59]* and *In[60]* below code two such SS identities.

```
In[59]:= set[u___, plusSC[p:mtimesSC[s_, s_], q:mtimesSC[t_, t_]],
         v___] /; MemberQ[{u, v}, plusSC[p, t]] ∧ MemberQ[{u, v},
         plusSC[q, s]] := set[u, v]
In[60]:= set[u___, plusSC[mtimesSC[s_, t_], s_, t_], v___] /;
         MemberQ[{u, v}, plusSC[mtimesSC[s, s], t]] ∧ MemberQ[{u, v},
         plusSC[mtimesSC[t, t], s]] := set[u, v]
```

Lemma 10

$$\{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}, \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\} = \{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\}. \tag{10.1}$$

Proof. Obviously,

$$\{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\} \subseteq \{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}, \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\}. \tag{$\mathfrak{P}$10.1}$$

Now $\overline{t}\dot{\circledast}\overline{t}$ can be either $\overline{t}$ or $\overline{\pi}$. Hence, $\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$ is either $\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}$ or $\overline{\pi}$. When $\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$ is $\overline{\pi}$, $\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$ will also be $\overline{\pi}$. Therefore

$$\{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}, \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\} \subseteq \{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\} \tag{$\mathfrak{P}$10.2}$$

is always true. Equation (10.1) follows from Equations ($\mathfrak{P}$10.1) and ($\mathfrak{P}$10.2). □

Lemma 11

$$\{\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\} = \{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\}. \tag{11.1}$$

Proof. Once again,

$$\{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\} \subseteq \{\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\}. \tag{$\mathfrak{P}$11.1}$$

When $\overline{s} = \overline{\imath}$, or $\overline{t} = \overline{\imath}$, or $\overline{s} = \overline{t}$, $\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}$ is obviously either $\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}$ or $\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$. So let $\overline{s} \neq \overline{\imath}$, $\overline{t} \neq \overline{\imath}$ and $\overline{s} \neq \overline{t}$. Then $\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}$ will be $\overline{\pi}$. But then, both $\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}$ and $\overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}$ will also be $\overline{\pi}$. Therefore

$$\{\overline{s}\dot{\circledast}\overline{t}\dot{\oplus}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\} \subseteq \{\overline{s}\dot{\circledast}\overline{s}\dot{\oplus}\overline{t}, \overline{s}\dot{\oplus}\overline{t}\dot{\circledast}\overline{t}\} \tag{$\mathfrak{P}$11.2}$$

is always true. Equation (11.1) follows from Equations ($\mathfrak{P}$11.1) and ($\mathfrak{P}$11.2). □

6.3 Handling Loops

The definition on line *In[61]* below shows how Σ operates on `while` loops.

```
In[61]:= Σ[while[b_List]] :=
(*1*)Module[{l, fp},
(*2*)       l = Cases[b, lhs_ ← _ → lhs, Infinity];
(*3*)       fp = FixedPoint[Σ[b]&, Σ[b],
                  $NUMITERS]; (* 1 to $NUMITERS+1 iterations. *)
(*4*)       If[Σ[b] =!= fp,
(*5*)          (Σ[#] ^:= 𝕊φ; # → 𝕊φ)& /@ l, (* Force solution to 𝕊φ . *)
(*6*)          fp]]
```

In (*2*), all program variables defined in the body of the `while` loop, including those defined in nested constructs, are determined. `FixedPoint` is a built-in object in Mathematica that applies a function until the result no longer changes; the expression `FixedPoint[`f`,` e`,` n`]` begins with e, and then repeatedly applies f at most n times until two consecutive applications of f produce the same result [Wol99]. Thus the effect produced in (*3*) is to apply Σ on the `while` loop's body at least once and at most `$NUMITERS`+1 times, stopping if a fixed-point value for the list of SS expression rewriting rules is achieved. In (*4*), a check is made if a fixed point has indeed been achieved—this is done by once again applying Σ on the `while` loop's body and symbolically comparing the result with the outcome of (*3*). If so, the fixed-point value is returned; otherwise, the SS expressions of all the variables defined in the loop body are forced to the greatest element $\mathbb{S}_\varphi$ of the lattice.

6.4 Fixed-Point SS Expressions

Before applying Σ on the Mathematica representation of the program in Figure 1, an initial solution needs to be set up. This is done on line *In[62]* below, which sets the initial SS expression of a variable v that is live on entry into the code fragment to `set[`σ`[`v`]]` where σ`[`v`]` is the SC expression that describes v's shape on entry into the code fragment. For variables defined within the code fragment, the initial solution is set to the empty set `set[]`.

```
In[62]:= {Σ[a0], Σ[b0], Σ[e0]} = {set[σ[a0]], set[σ[b0]], set[σ[e0]]};
         Σ[d1|d2|a1|b1|c|a2|b2|e1|e2|a3|b3|h] = set[];

In[63]:= $NUMITERS = ∞; Σ[stmts] // Timing

Out[63]= {0.22 Second,
          {d1 → set[σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]], d2 → set[σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]],
           a1 → set[σ[a0] ⊛ σ[a0] ⊕̇ σ[b0], σ[b0] ⊛ σ[b0] ⊕̇ σ[a0], σ[a0]],
           b1 → set[σ[a0] ⊛ σ[a0] ⊕̇ σ[b0], σ[b0] ⊛ σ[b0] ⊕̇ σ[a0], σ[b0]],
           c → set[σ[a0] ⊛ σ[b0], σ[a0] ⊛ σ[a0] ⊕̇ σ[b0], σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]],
           a2 → set[σ[a0] ⊛ σ[a0] ⊕̇ σ[b0], σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]],
           b2 → set[σ[a0] ⊛ σ[a0] ⊕̇ σ[b0], σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]],
           e1 → set[((σ[e0] ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0])) ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]))
                 ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]), (σ[e0] ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0])) ⊛
                 (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]), σ[e0] ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]), σ[e0]],
           e2 → set[((σ[e0] ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0])) ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]))
                 ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]), (σ[e0] ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]))
                 ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0]), σ[e0] ⊛ (σ[b0] ⊛ σ[b0] ⊕̇ σ[a0])],
           a3 → set[σ[a0] ⊛ σ[a0] ⊕̇ σ[b0], σ[b0] ⊛ σ[b0] ⊕̇ σ[a0], σ[a0]],
           b3 → set[σ[a0] ⊛ σ[a0] ⊕̇ σ[b0], σ[b0] ⊛ σ[b0] ⊕̇ σ[a0], σ[b0]],
           h → set[σ[a0] ⊛ σ[a0] ⊕̇ σ[b0], σ[b0] ⊛ σ[b0] ⊕̇ σ[a0], σ[a0] ⊕̇ σ[b0]]}}
```

Line *In[63]* shows an application of Σ. The application is preceded by the setting of `$NUMITERS` to ∞ to see if a useful fixed-point solution besides the forced case exists. For the given example, this does turn out to be the situation. For other input programs, a non-forced fixed point may either take an indefinite number of iterations because of the infinite height of the lattice $\mathcal{L}$, or may never be reached because of exponential blow-up due to the coded algebraic identities not covering the simplification of the generated SS expressions. The trade-off is achieved by suitably setting `$NUMITERS` to some finite value that is also small. For the example in Figure 1, a value of 3 suffices to reach the fixed point. Observe that the fixed-point solution on line *Out[63]* reveals all the claims made in Inference 3 on page 2. As mentioned before, the symbolic expressions

on line `Out[63]` can be used by a translator for optimizations such as preallocating array storage and verifying array shape correctness ahead of time.

7 Results

Apart from demonstrating the feasibility of our ideas, the Mathematica code shown in this report doubles as an initial algorithmic specification of a shape inference system for MATLAB. A full-blown implementation has been completed as part of a type analysis engine for MATLAB called MAGICA. The implementation is extensive and covers close to 70 built-in functions in the language, including a number of important Type II built-in functions like array indexing and colon expressions. In most cases, the original semantics of the built-in functions have been supported. For instance, in array indexing expressions, the subscripts themselves are allowed to be arrays. Conditional statements, all of MATLAB's looping constructs, and user-defined functions have also been covered.

7.1 Benchmarks

Metrics pertaining to shape information were collected over a set of 11 programs obtained from a variety of sources including the test suites of recent research compilers for MATLAB. The benchmarks were organized into input files that are called *M-files* in MATLAB jargon. Table 1 lists these benchmarks, along with brief descriptions, their sources and their sizes in terms of the number of input files and the total number of lines. Each benchmark consists of a separate driver routine from which is invoked the entry function of the benchmark. This organization is taken from FALCON. The current version of MAGICA supports built-in functions such as `disp` and `fprintf` using which output to an external file can be generated. However, no support currently exists for reading data from an external file. Some of the benchmarks did read data from external files; these were modified to include the loaded data in the driver routine. This wasn't found to be a problem because those benchmarks that did load data from external files only read in a couple of scalars. Two benchmarks that manipulate three dimensional arrays have also been included. In fact, in most of the publically available benchmarks, all data were restricted to scalars, vectors and matrices. This is specifically true for the test suites of previous research MATLAB compilers where arrays were confined to scalars, vectors and matrices. This is because early versions of MATLAB only supported matrices; multidimensional array support began in version series 5, the current version series being 6.

Benchmark	Synopsis	Origin	M-Files	Lines
adpt	Adaptive Quadrature by Simpson's Rule	† FALCON	2	79
capr	Transmission Line Capacitance	Chalmers University of Technology, Sweden	5	68
clos	Transitive Closure	☨ OTTER	2	30
crni	Crank-Nicholson Heat Equation Solver	FALCON	3	48
diff	Young's Two-Slit Diffraction Experiment	The MathWorks' Central File Exchange	2	40
dich	Dirichlet Solution to Laplace's Equation	FALCON	2	49
edit	Edit Distance	The MathWorks' Central File Exchange	2	34
❑ fdtd	Finite Difference Time Domain (FDTD) Technique	Chalmers University of Technology, Sweden	2	47
fiff	Finite-Difference Solution to the Wave Equation	FALCON	2	32
nb1d	One-Dimensional N-Body Simulation	OTTER	2	53
❑ nb3d	Three-Dimensional N-Body Simulation	Modified nb1d	2	46

❑ Benchmarks involve three-dimensional arrays.

TABLE 1: Benchmark Suite Description

7.2 Shape Composition

The "Shapes" column of Table 2 presents the total number of inferred shapes in each benchmark. These include the shapes of the original program variables, as well as the shapes of temporary variables that are introduced in the steps that lead up to type determination. For instance, temporaries get introduced during

the SSA conversion step that precedes the type determination phase. It additionally includes variables that are introduced by the effective splitting of $\dot{\Phi}_P(\overline{s}, \overline{t})$ into copies during the SSA inversion phase. The "Explicit Shapes" column indicates the percentage of inferred shapes that are explicit. The column also shows the breakup of these explicit shapes between scalar and nonscalar shapes. Explicit shapes allow translators to generate highly efficient code. For example, if a translator could infer that the variables `a` and `b` in the MATLAB expression `a+b` are scalars, it can generate a simple machine instruction to perform the addition rather than a generalized array addition routine. The measurements are with respect to Version 1.0 of MAGICA.

Benchmark	Shapes	Explicit Shapes			Symbolic Copies	Type Inference Timings (secs)	
		%	Scalar	Nonscalar	(%)	Kernel	MathLink
`adpt`	188	43.09	71	10	63.55	33.30	11.20 (25%)
`capr`	232	30.60	63	8	60.87	39.10	15.80 (29%)
`clos`	64	100	21	43	0.00	5.34	9.56 (64%)
`crni`	139	100	64	75	0.00	11.40	10.60 (48%)
`diff`	92	91.30	70	14	62.50	17.70	61.70 (78%)
`dich`	143	100	94	49	0.00	40.50	16.90 (29%)
`edit`	83	28.92	19	5	62.71	4.30	7.00 (62%)
`fdtd`	192	100	38	154	0.00	5.52	10.18 (65%)
`fiff`	88	100	58	30	0.00	5.16	11.04 (68%)
`nb1d`	163	10.43	12	5	64.38	11.40	6.80 (37%)
`nb3d`	118	16.10	14	5	37.37	7.24	5.56 (43%)

TABLE 2: Shape Composition

All nonexplicit shapes are symbolic. The "Symbolic Copies" column in Table 2 displays the percentage of the symbolic shapes that are inferred to be identical. This deduction happens as a consequence of algebraic simplifications such as those discussed in § 3.3.3. Establishing that one symbolic shape is exactly the same as another allows a translator not only to reduce the overhead of run-time array shape computation but also to avoid unnecessary shape checks. Table 2 also shows that in benchmarks that have symbolic shapes, a usually large percentage of the symbolic shapes can be determined to be identical. This indicates that making such an inference has the potential to considerably lower shape-related run-time overheads. As far as we know, no previous approach to shape determination can make this kind of an inference.

7.3 Timings

Timing measurements are presented in the last column. These were obtained on a 440 MHz UltraSPARC-IIi workstation running Solaris 7 and having 128MB of main memory. The "Kernel" subcolumn displays the time taken by the Mathematica kernel to arrive at *all* type inferences—intrinsic type, value range and shape—for each benchmark.

The "MathLink" subcolumn records the time taken to transfer the intermediate representation of the input program across to the Mathematica kernel, and to transfer the inferred type information back to the front-end. This data movement, which occurs over a special interprocess communication interface called MathLink, can be expensive. As Table 2 shows, in 5 out of 11 benchmarks, the MathLink timings account for over 60% of the *total* type inference times. The MathLink channel has been improving with newer releases and it is possible that the total type inference timings will be closer to that shown in the "Kernels" subcolumn in future versions of Mathematica. Still, considering the quality of the inferences made, we believe that the current timings are acceptable, especially since a typical usage scenario is one where the interpretor is used for prototyping and code development, and an inference-capable compiler is used for final production code generation.

8 Conclusions

This report presented a framework for inferring array shapes in array-based languages like MATLAB, and also showed how such a framework can be implemented using a commercial symbolic analysis package like Mathematica. Unlike all previous approaches, our framework exploits the algebraic properties that underlie MATLAB's shape semantics. This gives our approach a unique advantage in its ability to arrive at useful shape inferences even when array extents aren't compile-time determinable. Some of the inferences that our framework is capable of, as far we are aware, haven't been arrived at by any previous automatic array shape inference technique, be it for MATLAB or another array-based language. Examples of these are the powerful inferences shown on lines `Out[50]` and `Out[63]`.

Of course, the quality of the inferences is crucially dependent on how fully the various algebraic properties relating to array shape semantics are characterized. If none of the coded algebraic identities apply for a certain input program, shape equivalence may not be detected, and a useful fixed-point solution may not be arrived at. But the shape semantics of most of the language operators have simple algebraic properties that can be easily identified and codified, and from which shape inference benefits immediately accrue. Moreover, since not coding some algebraic identity does not mean incorrectness but only a missed opportunity at simplification and a useful inference, such systems can be built incrementally, ultimately trading quality of inference with run-time efficiency. We believe that the implementation of MAGICA, an archetype of such a type inference system, confirms this point in that a system capable of high-quality inferences can also be practically built and used.

References

[BS74] Alan M. Bauer and Harry J. Saal. "Does APL Really Need Run-Time Checking?". *Software—Practice and Experience*, 4(2):129–138, 1974.

[Bud88] Timothy Budd. **An APL Compiler**. Springer-Verlag, Inc., New York City, NY 10010, USA, 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.

[Chi86] Wai-Mee Ching. "Program Analysis and Code Generation in an APL/370 Compiler". *IBM Journal of Research and Development*, 30(6):594–602, November 1986.

[DRP99] Luiz Antônio De Rose and David A. Padua. "Techniques for the Translation of MATLAB Programs into FORTRAN 90". *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, March 1999.

[JB01] Pramod G. Joisha and Prithviraj Banerjee. "Computing Array Shapes in MATLAB". In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2001.

[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, March/April 1998.

[JSB00] Pramod G. Joisha, U. Nagaraj Shenoy, and Prithviraj 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.

[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.

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

[Mil79] Terrence C. Miller. "Type Checking in an Imperfect World". In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 237–243, January 1979.

[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.

[Ten74] Aaron M. Tenenbaum. "Type Determination in Very High-Level Languages". Ph.D. dissertation, Courant Institute of Mathematical Sciences, New York University, October 1974. Report NSO–3.

[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.

[Wol99] Stephen Wolfram. **The Mathematica Book**. Wolfram Media, Inc., Fourth Edition, 1999. ISBN 0–521–64314–7.