# An Algebraic Array Shape Inference System for MATLAB®

PRAMOD G. JOISHA and PRITHVIRAJ BANERJEE
Northwestern University

The problem of inferring array shapes ahead of time in languages that exhibit both implicit and dynamic typing is a critical one because the ramifications of its solution are the better organization of array storage through compaction and reuse, and the generation of high-performance code through specialization by shape. This article addresses the problem in a prototypical implicitly and dynamically typed array language called MATLAB. The approach involves modeling the language's shape semantics using an algebraic system, and applying term rewriting techniques to evaluate expressions under this algebra. Unlike prior efforts at array shape determination, this enables the deduction of valuable shape information even when array extents are compile-time unknowns. Furthermore, unlike some previous methods, our approach doesn't impose monotonicity requirements on an operator's shape semantics. The work also describes an inference methodology and reports measurements from a type inference engine called MAGICA. In a benchmark suite of 17 programs, the shape inference subsystem in MAGICA detected the equivalence of over 61% of the symbolic shapes in six programs, and over 57% and 37% of the symbolic shapes in two others. In the remaining nine programs, all array shapes were inferred to be compile-time constants.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Very high-level languages*; D.3.4 [**Programming Languages**]: Processors—*Interpreters*; *compilers*; *optimization*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*; I.1.1 [**Symbolic and Algebraic Manipulation**]: Expressions and Their Representation—*Simplification of expressions*; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms

General Terms: Algorithms, Design, Languages, Theory

Additional Key Words and Phrases: Typeless array languages, shape algebras, term rewriting

## 1. INTRODUCTION

Array-centric models of computation in the APL mold have witnessed a renaissance of sorts in the last two decades through the widespread success of

This research was supported by DARPA under Contract F30602-98-2-0144, and by NASA under Contract 276685/NAS5-00212. The *MATHEMATICA* fonts in this paper are due to Wolfram Research, Inc.

Authors' addresses: P. G. Joisha, Microsoft Research, One Microsoft Way, Redmond, WA 98052-6399; email: pjoisha@microsoft.com; P. Banerjee, University of Illinois at Chicago, Chicago, IL 60607-7043; email: prith@uic.edu.

```
1   d ← max(b*b, abs(a)*b);
2   a ← atan2(b, b');
3   d ← (b+d)*fix(b);
4   while $p_1$,
5      c ← a*b;
6      a ← c+a.^b;
7      b ← c-a./c;
8      while $p_2$,
9         e ← e*d;
       end;
    end;
10  h ← b.^b-a;
```

Fig. 1. Implicit and dynamic assignment of array types.

very high-level proprietary languages like MATLAB and IDL. By melding a lucid syntax with features such as implicit typing, multidimensional data structures, real and complex arithmetic, a rich polymorphic function set, and an interpretive execution model, these languages have proven to be ideal tools for an "exploratory" style of program development in which the processes of data examination, manipulation, visualization, and incremental code development are combined to form a tight feedback-driven loop. Although such features—particularly implicit and dynamic *array rank* and *array shape* typing, as well as array polymorphism—may have greatly promoted the level of abstraction possible with these languages and their ease of use, they have also severely burdened their execution supports, forcing implementations to usually cope through interpretation. (The term rank in this article has the FORTRAN 90 and APL connotation; it means the dimensionality of an array. The term shape means the collection of extents of an array along each of its dimensions.) Prior knowledge of an array's type attributes, such as rank, shape, and intrinsic type,[1] is thus desirable because of its potential to improve execution performance and efficiency. For instance, knowing an array's shape in advance could eliminate redundant shape conformance checks, permit compile-time verification of shape correctness, and allow for the preallocation of array storage.

## 1.1 Tacit Array Typing

To understand how array typing works in MATLAB, consider the synthetic code fragment in Figure 1. The statement `c ← a*b` on Line 5 assigns to program variable `c` the result of `a*b`, where `*` is MATLAB's matrix-multiplication operation.[2] If either `a` or `b` is 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 have to

[1]Intrinsic type determination is also an issue in MATLAB because its arrays have an orthogonal intrinsic type attribute that dictates the arithmetic possible on the contents.

[2]The symbol ← will be used to denote the assignment operation in MATLAB. The infix notations `+`, `.^`, `./`, and `-` in Figure 1 stand for the plus, array power, right array division, and minus operations, respectively [The MathWorks, Inc. 1997]. These, along with the calls `max` and `atan2`, which compute an elemental maximum and the four-quadrant inverse tangent, are examples of *elementwise* operations in MATLAB. Elementwise operators expect at least one operand to be scalar or both operands to be of the same shape. The unary operations `fix` (rounding towards zero) and `abs` (absolute zero) are similar in that they are applied elementwise on their operands to produce identically shaped results. The only two operations that aren't elementwise in Figure 1 are complex-conjugate transposition (`'`) and matrix multiplication (`*`).

be $p \times q$ and $q \times r$, respectively [The MathWorks, Inc. 1997]. Current static inference systems for MATLAB [Quinn et al. 1998; Chauveau and Bodin 1998; De Rose and Padua 1999] can infer the shape of c, at least on the first iteration, 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 runtime 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 would be the case for the 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.

The thesis of this work is that even when shapes aren't explicitly known, we can do better if the algebraic properties associated with the language's shape semantics are leveraged. For example, in the case of Figure 1, we shall see in Section 7 that:

*Inference* 1. The shapes of d on Lines 1 and 3 are identical for *any* initial a and b. This means that a translator can replace shape conformance checks against Lines 1 and 3 by a single check against Line 1.

*Inference* 2. The shapes of a and b on Lines 6 and 7 will be identical in each iteration.

*Inference* 3. There can only be two possibilities for the shape of c, one for both 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.

## 1.2 "Lossless" Polyrank Routines

A unique feature that distinguishes MATLAB from most other languages is the ability to write polyrank routines (routines that can be invoked on arrays of various ranks) such that the formal parameter's rank always reflects the actual parameter's rank. While languages such as C, FORTRAN 90, and FORTRAN 77 allow the writing of polyrank procedures, their facility is inherently *lossy* in that information pertaining to the actual argument's rank isn't retained by the formal argument within the procedure. In these languages, the rank of an array formal parameter is either set statically by the number of dimensions specified in its declaration, or left unspecified, as may happen in C if pointer parameters are used. The only way to communicate an actual's rank into a user-defined routine, then, is either by passing an additional parameter or by involving globally visible memory locations.

To exemplify this distinction, consider the problem of adding to an array of rank $n$ a generalization of the Hilbert matrix [Weisstein 2005] to $n$ dimensions. Specifically, we define a *Hilbert array* of $n$ dimensions as having the elements

$$H_{i_1 i_2 \ldots i_n} = \frac{1}{i_1 + i_2 + \cdots + i_n - n + 1}.$$

Figure 2 shows a FORTRAN 90 routine that adds to an argument a Hilbert array of the same shape. The routine is polyrank by virtue of the *assumed-size*

```
      SUBROUTINE HILARRAY_ACC(A, RANK, EXTENTS)
      REAL, DIMENSION (*) :: A
      INTEGER :: RANK
      INTEGER, DIMENSION (1:RANK) :: EXTENTS
      INTEGER, DIMENSION (1:RANK) :: INDICES

      ! First index in array element order.
      INDICES(:) = 1

      ! Argument array stepped through in
      ! array element order.
      DO 10 I = 1, PRODUCT(EXTENTS)

         A(I) = A(I)+1/(SUM(INDICES)-RANK+1)

         ! INDICES incremented in
         ! array element order.
         DO 20 K = 1, RANK
            IF (INDICES(K) /= EXTENTS(K)) THEN
               INDICES(K) = INDICES(K)+1
               GOTO 10
            ELSE
               INDICES(K) = 1
            END IF
      20 CONTINUE

   10 CONTINUE

      END SUBROUTINE HILARRAY_ACC
```

Fig. 2. Lossy polyrank Hilbert array accumulation in FORTRAN 90.

```
function b = hilarray_acc(a)

rank = length(size(a));

indices = ones(size(a));
sindices = zeros(size(a));

for k = 1:rank,
    sindices = sindices+ ...
    cumsum(indices, k);
end;

b = a+1./(sindices-rank+1);
```

Fig. 3. Lossless polyrank Hilbert array accumulation in MATLAB.

array dummy parameter [Adams et al. 1992]. However, this polyrank trait is lossy because the dummy is a statically fixed one-dimensional array. In Figure 3 we illustrate a *lossless* polyrank MATLAB version that relies on a reflective call to `size` to access the actual parameter's rank.

Although the ability to write lossless polyrank code permits a more natural expression of multidimensional array logic, it also adds another wrinkle that any array shape inference system for MATLAB has to contend with. In this regard, our shape inference system diverges from all previous efforts at inferring array shapes in MATLAB because these efforts practically considered a monorank MATLAB by restricting the array dimensionalities to two [Quinn et al. 1998; Chauveau and Bodin 1998; De Rose and Padua 1999; Almási and Padua 2002]. This article will later show (in Section 8) how automated inference techniques could be used to deduce that:

*Inference* 4. Irrespective of their ranks, the shapes of `a` and `b` in Figure 3 are always identical under every successful execution of the function.

## 1.3 Contributions

The primary contribution of this work is a uniform approach to the problem of shape inference in implicitly and dynamically typed array languages that is grounded in an algebraic view of array shape. In contradistinction to all past efforts, our approach enables gathering valuable information, even when array ranks and extents are compile-time unknowns. This work systematically categorizes the problem domain and shows how the approach addresses two important classes of language operators. Also, unlike past efforts, our approach imposes no restrictions on an array's rank, and the quality of the inferences produced don't suffer from a lack thereof.

Secondary contributions include symbolic evaluation methods that automate a shape calculus based on algebras, and two applications of the presented framework for obtaining different kinds of shape-related facts: (1) detecting whether the shape of one array expression, even when unknown, tracks the shape of another and (2) determining the set of all possible shapes that an array expression may assume at runtime. The article also presents experimental data that shows how an implementation of the approach fares on real MATLAB programs.

## 2. RELATED WORK

Early research on array shape determination was in the context of the more general problem of automatic type inference. Widely employed methodologies for the latter have been to model the problem using a system of "type equations," and to then solve that system either by a process called unification [Hindley 1969; Milner 1978] or by using lattice-theoretic fixed-point techniques [Tenenbaum 1974; Kaplan and Ullman 1978]. All previous efforts at inferring the shape attribute of type in APL and MATLAB, to the best of our knowledge, have been fixed-point solutions to lattice-theoretic formulations of the problem [Ching 1986; Budd 1988; De Rose and Padua 1999; Almási and Padua 2002]. Though type estimation techniques founded on iterative dataflow analyses—their conceptual underpinning is a bounded lattice—look at types in a broad sense, they are most suited to situations in which a language's types can be naturally arranged into a partial order. In particular, these approaches rely on the existence of lattice-forming partial orders that are also *subtyping* relations, and an assignment operation on which the sense of the subtyping is reversed (i.e., *contravariant*). A subtyping is a binary relation between types that captures the principle of *safe substitution* [Mitchell 1996]: If $s$ and $t$ are two types such that $s \preceq t$ (read as "$s$ is a subtype of $t$"), then any value of type $s$ can be safely *used* wherever any value of type $t$ can be safely used. If $\preceq$ is also given to be contravariant on the assignment operation, then

$$(t \leftarrow u) \preceq (s \leftarrow u) \quad \text{if } s \preceq t. \tag{1}$$

In other words, if $s$ is a subtype of $t$, then an assignment to a variable of type $s$ can always be safely substituted by an assignment to a variable of type $t$. When $\preceq$ is additionally a bounded lattice, the contravariant property implies that the greatest element is always a safe type inference for any program variable.[3] If the greatest element also has a reasonably efficient compile-time implementation, then such lattices are very effective for problems such as *intrinsic type* determination (i.e., finding whether a program variable is an integer, a real, a complex number, and so on), and they have been successfully used for this in both MATLAB [De Rose and Padua 1999; Joisha and Banerjee 2001b; Almási and Padua 2002] and APL [Ching 1986; Budd 1988].

## 2.1 Limitations of Lattice-Based Approaches

Suppose for the moment that we are only to consider shape-correct MATLAB programs. A variable in such a program might take on one of many shapes in the course of execution. Each assumed shape could be described by a *shape tuple* $\langle p_1, p_2, \ldots, p_k\rangle$ that denotes a $k$-dimensional array with extent $p_i$ along the $i$th dimension, where each $p_i$ is an element in the set of nonnegative integers $\mathbb{W}$.[4] If $\mathbb{L}_\mathbb{S}$ is the collection of all such shape tuples, then a lattice could be defined on the power set $2^{\mathbb{L}_\mathbb{S}}$ in which the subset relation is the partial order and the greatest element is $\mathbb{L}_\mathbb{S}$. Then, the partial order would also be a subtyping that is contravariant on the assignment operation. Thus, $\mathbb{L}_\mathbb{S}$ would always be a safe shape inference for any program variable. However, as will be discussed in Section 2.1.1, such a lattice becomes ineffectual when analyzing functions in isolation because there exists the likelihood of the greatest element being returned as the shape of every variable local to the function. Since the greatest element represents any shape, a translator would then be forced to emit the most general code for the function's body, even though opportunities for specialization may exist due to inter-shape relationships.

This poses the following interesting question: Could a bounded lattice be defined for MATLAB so that its greatest element is some *finite* subset of the set of all possible legal shape tuples, and such that its partial order is still a subtyping contravariant on the assignment operation? If such a lattice could be defined, then its greatest element would also be a safe fallback inference for which code with some degree of specialization (because it isn't the universal set of shapes) might be generated. Unfortunately, as Theorem 1 to follow will show, it is impossible to define such a lattice for MATLAB because of the nature of shape in the language. Theorem 1 is significant because there have been attempts to the contrary at using lattices in which the greatest element is some specific shape (see Section 2.1.2).

[3]Some runtime type checking in the form of checked casts may still be needed.
[4]We choose $\mathbb{W}$ so as to include the empty array construct.

```
function S = quadrature(a, b)

h ← (b-a)/2;
mid ← (a+b)/2;

Fa ← 13.*(a-a.^2).*exp(-3.*a./2);
Fmid ← 13.*(mid-mid.^2).*exp(-3.*mid./2);
Fb ← 13.*(b-b.^2).*exp(-3.*b./2);

S ← h*(Fa+4*Fmid+Fb)/3;
```

Fig. 4. Numerical computation of $\int_a^b 13(x - x^2)e^{\frac{-3x}{2}}\,dx$ by Simpson's rule.

THEOREM 1. *It is impossible to define a bounded lattice on $2^{\mathbb{L}_\mathbb{S}}$ for shape inference in MATLAB that meets all of the following conditions:* (1) *the lattice elements span $\mathbb{L}_\mathbb{S}$,* (2) *the greatest element is a finite subset of $\mathbb{L}_\mathbb{S}$, and* (3) *the partial order is a subtyping that is contravariant on the assignment operation.*

PROOF. Consider a bounded lattice on some subset $\{l_1, l_2, \ldots\}$ of $2^{\mathbb{L}_\mathbb{S}}$ such that

$$\bigcup_{j \geq 1} l_j = \mathbb{L}_\mathbb{S} \tag{2}$$

and let $\mathbf{1}$ be its greatest element. If $\mathbf{1}$ is a finite subset of $\mathbb{L}_\mathbb{S}$, then there exists a shape tuple $\langle z_1, z_2, \ldots, z_k \rangle$, where $z_k \neq 1$, such that all shape tuples of the form

$$\langle z_1, z_2, \ldots, z_k, \overbrace{1, \ldots, 1}^{\text{0 or more}} \rangle$$

don't belong to $\mathbf{1}$ (if this weren't true, then $\mathbf{1}$ would have been an infinite set).

Now from Equation (2), there exists a lattice point $l$ such that $\langle z_1, z_2, \ldots, z_k \rangle \in l$, where $l < \mathbf{1}$. Consider an assignment to a variable `X` whose shape is described by $l$. If the lattice's partial order is a subtyping that is indeed contravariant on the assignment operation, then it should be possible to replace the assignment to `X` by an assignment to a variable `Y` whose shape is described by $\mathbf{1}$. However, because

$$\langle z_1, z_2, \ldots, z_k, \overbrace{1, \ldots, 1}^{\text{0 or more}} \rangle \notin \mathbf{1},$$

such a substitution could change the meaning of the program. As an example,

```
X ← ...;
disp(size(X));
```

may produce a different output if `X` were to be replaced by `Y`. □

2.1.1 *The Problem of Separate Compilation.* From Theorem 1, the greatest element in any bounded lattice used for inferring array shapes in MATLAB must be some abstraction that represents an infinite collection of shapes. This means that if the greatest element is returned as an inference, no useful specialization may be possible. For example, if the lattice in De Rose [1996], and De Rose and Padua [1999] were used to infer the shapes in the code of Figure 4,

```
function unsafe()

x ← rand(3, 2);
a ← rand;

while p,
   a ← [a, rand];
end;
y ← a*x;
```

Fig. 5. Eliciting an unsafe shape solution from MaJIC.

excerpted from the adaptive quadrature program in the FALCON benchmark suite [De Rose 1996], its greatest element ⟨UNKNOWN, UNKNOWN⟩ would be returned as the shape of every variable defined in the procedure.[5] However, due to the shape semantics of the various operators used in the `quadrature` function, we shall later see that:

*Inference* 5. The shapes of `h`, `mid`, `Fa`, `Fmid`, and `Fb` in Figure 4 are always identical.

*Inference* 6. For `quadrature` to be well-defined, `a` and `b` must be square matrices, through not necessarily of the same shape (this includes the case in which they are scalars). Then, the shapes of all the lefthand side variables will be identical.

A translator could use Inference 6 both to collapse the shape conformance checks for the function to a single check situated at its entry, and to do a Chaitin-style coalescing of the storages assigned to `Fa`, `Fmid`, `Fb`, and `S` [Joisha and Banerjee 2003a].

2.1.2 *Unsafe Shape Solutions in MaJIC.* Nearly every previous lattice used for deducing shapes in both MATLAB and APL has inadvertently adhered to the requirements of Theorem 1.[6] An exception has been the lattice

$$L_s = \{\mathbb{W} \times \mathbb{W}, \bot_s, \top_s, \sqsubseteq_s\}$$

of two-dimensional shapes used in Almási [2001], and Almási and Padua [2002], where $\bot_s = \langle 0, 0\rangle$ and $\top_s = \langle \infty, \infty\rangle$ were the least and greatest elements in $L_s$, respectively, and the lattice's partial order $\sqsubseteq_s$ was defined as

$$\langle a, b\rangle \sqsubseteq_s \langle c, d\rangle \quad \text{iff } a \le c \text{ and } b \le d \text{ for all } a, b, c, d \in \mathbb{W}. \tag{3}$$

It can be easily verified that $\sqsubseteq_s$ isn't a subtyping and that $\top_s$ isn't always a safe solution. A contrived example that exposes its unsafeness is shown in Figure 5. The function `unsafe` incrementally grows `a` within a `while` loop into a row vector whose size is statically indeterminate. Because the shape lattice $L_s$ in MaJIC is of infinite height, any initially chosen solution for the shape of `a` won't converge. MaJIC accommodates such "runaway" situations by setting the shape of `a` to $\top_s$ after a fixed number of iterations so as to force convergence [Almási 2001, 33]. However, $\top_s$ won't always be a safe solution for `a` because it may not conform

[5]The lattice defined in De Rose [1996] and De Rose and Padua [1999] is at the granularity of array extents; the Cartesian product of this lattice with itself gives the shape lattice effectively used.
[6]Theorem 1 can also be similarly argued for APL.

to the rules of the matrix-multiplication operation on the last line, even though the `while` loop test $p$ may be such that at runtime, the loop either doesn't iterate or iterates exactly thrice, thereby ensuring that the program is always well-defined.[7]

From another perspective, it can be shown that not all of the *shape transfer functions* comprising $L_s$'s function space are monotonic with respect to $\sqsubseteq_s$.

THEOREM 2. *The shape transfer function for MATLAB's matrix-multiplication operation isn't monotonic on the lattice $L_s$.*

PROOF. Let $\circledast$ be the shape transfer function for the matrix-multiplication operation considering only two-dimensional shapes. According to MATLAB's definition of this operation in the two-dimensional case [Almási 2001, 83],

$$\langle 1, x\rangle \circledast \langle 1, 1\rangle = \langle 1, x\rangle, \tag{4}$$

$$\langle 1, x\rangle \circledast \langle x, 1\rangle = \langle 1, 1\rangle \tag{5}$$

for any $x$ in $\mathbb{W}$. From Equation (3),

$$\langle 1, 1\rangle \sqsubseteq_s \langle y, 1\rangle$$

for any $y \geq 1$. From the assumed monotonicity of $\circledast$, this gives us

$$\langle 1, y\rangle \circledast \langle 1, 1\rangle \sqsubseteq_s \langle 1, y\rangle \circledast \langle y, 1\rangle.$$

But from Equations (4) and (5), the preceding becomes

$$\langle 1, y\rangle \sqsubseteq_s \langle 1, 1\rangle,$$

which contradicts the definition of the partial order in Equation (3) when $y > 1$. □

## 2.2 Alternate Approaches

Just as our work seeks to identify redundant array shape checks, there has also been research in the area of array bounds checking that has aimed to deduce redundant runtime array index checks. These approaches range from flow analysis techniques that propagate and eliminate bounds checks [Gupta 1993] to type-based, annotation-aided, constraint-solving methods [Xi and Pfenning 1998].

In Ancourt and Nguyen [2001], the problem of calculating the exact size of FORTRAN arrays that have the `REAL(1)` or `REAL(*)` declarations is considered. The article discusses a method based on array region analysis to determine the set of array elements referenced during program execution. The quality of the results produced, however, hinges on the accuracy of the array region analysis. While the article outlines a scheme for the analysis, it isn't clear how effective this would be in an APL or MATLAB setting. The primary motivation for the work was to clean up and tighten the large number of imprecise array declarations in legacy FORTRAN code, declarations that would otherwise hinder

[7]Under finite precision arithmetic, $\top_s$ would presumably map to $\langle \mathcal{J}, \mathcal{J}\rangle$, where $\mathcal{J}$ is some large (perhaps the largest) machine-representable positive integer.

program analysis, parallelization, verification, and code readability [Ancourt and Nguyen 2001].

Another approach to shape inference is to use propositional logic [McCosh 2003]. This approach represents the shape constraints of a statement as a sequence of clauses, where each clause is a valid assignment of types to program variables. A graph is then used to combine the constraints over the entire function and the solution process is reduced to that of finding $n$-cliques over the graph. Loops are handled by casting the CFG into the SSA form and modeling the $\phi$-function introduced at join points. The work in McCosh [2003] differs from ours in its main objective: to generate type-based specializations of libraries for specific client contexts.

### 2.3 Shape-Aware Array Language Designs

There has also been work on the bottom-up design of tacitly typed array languages that permit a complete static analysis of shape. An example is the FISh language [Jay and Steckler 1998], which achieves this by the imposition of shape constraints, such as not allowing an array's shape to change during execution. Restrictions of this kind don't exist in MATLAB. Also, it isn't clear whether the facilities in FISh are sufficient for the language to be used in a production setting; the only nontrivial FISh programs on which timings have been reported appear to be the fast Fourier transform, quicksort, and matrix multiplication [Jay and Steckler 1998].

## 3. OVERVIEW OF OUR APPROACH

Let $\mathbb{S}_{\wp}$ be the set of array shapes, $\mathbb{C}$ the set of complex numbers, and $\Delta = \bigcup_{k\geq 0} \mathbb{C}^k$ the set of all tuples consisting of complex numbers (including the empty tuple). Then, every $n$-element array $e$ in a language such as MATLAB could be thought of as corresponding to an element $(\chi(e), \kappa(e))$ in $\Delta \times \mathbb{S}_{\wp}$, where $\chi(e)$ is the tuple of the array elements in $e$ taken in some particular order, and $\kappa(e)$ is a (unique) denotation of the array structure on these elements. An operation $f$ in MATLAB can thus be considered a mapping from $(\Delta \times \mathbb{S}_{\wp})^n$ to $\Delta \times \mathbb{S}_{\wp}$.[8] Our basic idea to shape determination is to decouple $f$ into a pair of mappings, $f_d : (\Delta \times \mathbb{S}_{\wp})^n \mapsto \Delta$ and $\mathring{f} : (\Delta \times \mathbb{S}_{\wp})^n \mapsto \mathbb{S}_{\wp}$, that *separately* describes its elemental and shape effects, and to symbolically evaluate the computations associated with $\mathring{f}$ at translation-time.

There are two gains from this approach:

(1) For a large class of operations that we call Type I, the shape operator $\mathring{f}$ of $f$ is independent of $\Delta$, that is, $\mathring{f}$ is simply a mapping from $\mathbb{S}_{\wp}^n$ to $\mathbb{S}_{\wp}$. Consequently, their shape computations could be evaluated independently of the main program.

(2) For the Type I class, the collection of shape operators defines a rich algebraic structure on $\mathbb{S}_{\wp}$. Properties in this algebra could permit the simplification of shape computations, even though they may not be reducible to an explicit

[8]The arity $n$ includes all operands that may influence $f$, including implicit influences such as global variables. The inputs to $f$ are assumed to be such that it doesn't go into an infinite loop. Also, we presently ignore ill-conforming inputs, that is, those that may cause $f$ to abnormally exit.

```
     ▶ $\kappa(d_1) \leftarrow (\kappa(b_0)\dot{*}\kappa(b_0))\dot{+}(\kappa(a_0)\dot{*}\kappa(b_0))$
1    $d_1$ ← max($b_0$*$b_0$, abs($a_0$)*$b_0$);
     ▶ $\kappa(a_1) \leftarrow \kappa(b_0)\dot{+}(\dot{\neg}\,\kappa(b_0))$
2    $a_1$ ← atan2($b_0$, $b_0$');
     ▶ $\kappa(d_2) \leftarrow (\kappa(b_0)\dot{+}\kappa(d_1))\dot{*}\kappa(b_0)$
3    $d_2$ ← ($b_0$+$d_1$)*fix($b_0$);
     ▶ $\kappa(a_2) \leftarrow \dot{\Phi}(P)(\kappa(a_1), \kappa(a_3))$
     $a_2$ ← $\Phi(P)$($a_1$, $a_3$);
     ▶ $\kappa(b_1) \leftarrow \dot{\Phi}(P)(\kappa(b_0), \kappa(b_2))$
     $b_1$ ← $\Phi(P)$($b_0$, $b_2$);
     ▶ $\kappa(e_1) \leftarrow \dot{\Phi}(P)(\kappa(e_0), \kappa(e_2))$
     $e_1$ ← $\Phi(P)$($e_0$, $e_2$);
4    while $p_1$,
          ▶ $\kappa(c) \leftarrow \kappa(a_2)\dot{*}\kappa(b_1)$
5         c ← $a_2$*$b_1$;
          ▶ $\kappa(a_3) \leftarrow \kappa(c)\dot{+}(\kappa(a_2)\dot{+}\kappa(b_1))$
6         $a_3$ ← c+$a_2$.^$b_1$;
          ▶ $\kappa(b_2) \leftarrow \kappa(c)\dot{+}(\kappa(a_3)\dot{+}\kappa(c))$
7         $b_2$ ← c-$a_3$./c;
          ▶ $\kappa(e_2) \leftarrow \dot{\Phi}(Q)(\kappa(e_1), \kappa(e_3))$
          $e_2$ ← $\Phi(Q)$($e_1$, $e_3$);
8         while $p_2$,
               ▶ $\kappa(e_3) \leftarrow \kappa(e_2)\dot{*}\kappa(d_2)$
9              $e_3$ ← $e_2$*$d_2$;
          end;
     end;
     ▶ $\kappa(h) \leftarrow (\kappa(b_1)\dot{+}\kappa(b_1))\dot{+}\kappa(a_2)$
10   h ← $b_1$.^$b_1$-$a_2$;
```

Fig. 6. Pruned SSA form of Figure 1, with decoupled shape semantics.

shape at translation-time. This means that the runtime overhead associated with their execution can be either mitigated or eliminated.

Figure 6 shows the shape computations for a pruned static single assignment (SSA) form [Cytron et al. 1991] of Figure 1.[9] Each shape computation immediately precedes its corresponding MATLAB statement, and is shown using a ▶ prefix (to keep the exposition simple, the figure ignores the explicit representation of the $f_d$ part of a statement's $f$). For instance, in Line 2, only the shape of $a_1$ is affected. Its new shape can be described by the computation

$$\kappa(a_1) \leftarrow \kappa(b_0)\dot{+}(\dot{\neg}\kappa(b_0)),$$

where $\dot{\neg}$ and $\dot{+}$ are, respectively, the shape operators corresponding to the complex conjugate transpose and array addition operations in MATLAB. In this case, the shape of $a_1$ is fully describable, given the shape of $b_0$. In fact, the shapes of all the expressions in Figure 6 are fully describable, given the gate variables $P$ and $Q$ and the shapes of $a_0$, $b_0$, and $e_0$. The shape computations can thus be regarded as forming a "shape program" that can be reasoned over and partially evaluated.

The rest of this article is organized as follows. We begin by formalizing the notion of an array shape in Section 4. The section addresses the issue of array ranks and describes a key property exhibited by MATLAB that we call *selective rank demotion*, which allows all shape computations for the language to be viewed in a rank-neutral manner. The modeling of these shape computations and the algebraic structure that they induce is discussed in Section 5. Section 6 then presents a method to mechanically evaluate shape computations. At the heart of the method is a rewriting process that represents algebraic identities using rules and that handles the properties of commutativity and associativity by special structural transformations. Two important and illustrative applications of the method to shape analysis are discussed in Section 7. In the first, we perform a kind of peephole analysis on stretches of code paths to determine

[9]A pruned SSA form forgoes $\phi$-function assignments to variables that aren't subsequently used. The injected $\phi$-function assignments that achieve the SSA property are shown in gated curried form (see Section 6.6), with $P$ and $Q$ serving as the gate variables.

identical shapes. In the second, we use the symbolic evaluation method in an abstract interpretation of the control-flow graph so as to approximately calculate the set of shapes assumed by a MATLAB expression. The applicability of the symbolic evaluation scheme to other classes of language operators is then discussed in Section 8. Finally, Section 9 reports measurements of an implementation on a benchmark suite that is composed mainly of publicly available MATLAB programs.

## 4. THE SEMANTICS OF SELECTIVE RANK DEMOTION—BASIS FOR A FIXED-RANK SHAPE CALCULUS FOR MATLAB

Logically, all arrays in MATLAB have at least two dimensions. For example, a scalar is a $1 \times 1$ matrix and a column vector is an $x \times 1$ matrix. A pivotal decision that went into MATLAB's design, intentional or not, was to completely ignore trailing extents of unity from the third dimension onwards in its operational semantics. For instance, an array of shape $\langle 2, 3, 1, 2, 1\rangle$ is always treated as if it is of shape $\langle 2, 3, 1, 2\rangle$. This selective demotion of an array's rank implies that any array shape analysis for MATLAB can assume all shapes as being padded by an indefinite run of unit extents; that is, all arrays as having a very large, common fixed rank.

Assuming that the set $\mathbb{L}_{\mathbb{S}}$ of shape tuples is

$$\mathbb{L}_{\mathbb{S}} = \{\langle w_1, w_2, \ldots, w_m\rangle \mid m \geq 2 \wedge w_i \in \mathbb{W} \text{ for all } 1 \leq i \leq m\}, \tag{6}$$

selective rank demotion in MATLAB is, in essence, a subtyping $\wp$ on $\mathbb{L}_{\mathbb{S}}$ such that the two shape tuples $\boldsymbol{s} = \langle p_1, p_2, \ldots, p_k\rangle$ and $\boldsymbol{t} = \langle q_1, q_2, \ldots, q_l\rangle$ are related by $\wp$ if and only if they differ at most by trailing extents of unity beyond the second dimension:

$$\boldsymbol{s}\wp\boldsymbol{t} \Leftrightarrow (\boldsymbol{s} = \boldsymbol{t})\vee(\boldsymbol{s} = \langle q_1, q_2, \ldots, q_l, 1, \ldots, 1\rangle)\vee(t = \langle p_1, p_2, \ldots, p_k, 1, \ldots, 1\rangle). \tag{7}$$

### 4.1 Canonical Shape Tuples and Ranks

Clearly, $\wp$ is also an equivalence relation on $\mathbb{L}_{\mathbb{S}}$. If $\bar{\boldsymbol{s}}$ is the equivalence class under $\wp$ of the shape tuple $\boldsymbol{s}$ of a MATLAB expression $e$, then any element in $\bar{\boldsymbol{s}}$ would be an equally valid denotation of $e$'s shape. Therefore, the term *shape* in this article will formally mean an equivalence class, or *shape tuple class*, under $\wp$. The concept of equivalent shape tuples leads to the idea of an array's *canonical* shape tuple and rank: 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, and its canonical rank is the smallest rank that can be ascribed to it.[10]

4.1.1 *Shape Errors.* To accommodate shape errors, we include "illegal" shape tuples by considering a set $\mathbb{I}_{\mathbb{S}}$ whose members don't belong to $\mathbb{L}_{\mathbb{S}}$. There could be a number of suitable choices for $\mathbb{I}_{\mathbb{S}}$; one such choice that doesn't clash with any subset of $\mathbb{L}_{\mathbb{S}}$ and also forms an equivalence class under $\wp$ is:

$$\mathbb{I}_{\mathbb{S}} = \{\langle \pi_1, \pi_2\rangle, \langle \pi_1, \pi_2, 1\rangle, \langle \pi_1, \pi_2, 1, 1\rangle, \ldots\}. \tag{8}$$

[10]The canonical rank is hence the number of components in the array's canonical shape tuple.

The aforementioned symbols $\pi_1$ and $\pi_2$ could be integers such that $\pi_1 < 0$ or $\pi_2 < 0$, or both. We will represent the canonical illegal shape tuple $\langle \pi_1, \pi_2 \rangle$ as $\boldsymbol{\pi}$. The augmented set $\mathbb{S} = \mathbb{L}_{\mathbb{S}} \cup \mathbb{I}_{\mathbb{S}}$ will be the universe of shape tuples in this article. The *quotient set* of $\mathbb{S}$ by $\wp$ [Tremblay and Manohar 1975], which is the set of all shape tuple classes under $\wp$, will be denoted by $\mathbb{S}_{\wp}$.

4.1.2 *Terminology.* When describing MATLAB array structure in this article, the following terms will be used:

*Illegal Array*. An array whose canonical shape tuple is $\boldsymbol{\pi}$. Illegal arrays abstract MATLAB expressions that are ill-formed according to the shape rules.

*Matrix*. A legal array whose canonical rank is two.

*Scalar*. A matrix with unit extents along the first and second dimensions.

*Row Vector*. A matrix with unit extent along the first dimension.

*Column Vector*. A matrix with unit extent along the second dimension.

Observe the overlap in some of these definitions—for example, that which is a scalar is also a matrix and a vector. We shall use the phrase "higher-dimensional array" when referring to legal arrays whose canonical ranks are at least three. 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.

## 4.2 What Does Selective Rank Demotion Buy?

An important issue that must be addressed in the design of a shape analyzer for any array language is how a program's meaning will be affected by an array's rank. The key benefit of selective rank demotion is that it enables a simpler shape analysis (since all arrays can be assumed to be of the same rank $\omega$, where $\omega$ is an ordinal number larger than any finite integer[11]) without altering a program's meaning.

While selective rank demotion and its incorporation into an array language's operational semantics may seem obvious, it doesn't exist in languages such as FORTRAN 90 and APL. An example of this in APL is a matrix with one element. This has a different meaning than a column vector with one element, and both have meanings different from a scalar. By contrast, the three are indistinguishable in MATLAB. An example of this in FORTRAN 90 are the arrays

```
REAL, DIMENSION (3, 2, 1) :: A
REAL, DIMENSION (3, 2) :: B
```

The shapes of `A` and `B` in the preceding, as far as FORTRAN 90 is concerned, are distinct. An easy way to substantiate this difference is to invoke the array inquiry intrinsic function `SHAPE` on `A` and `B` and compare the results.

[11] $\omega$ can be considered to be the first transfinite number, which is the "smallest" infinity with this property.

### 4.3 Empty Arrays

The framework of this article automatically encompasses the empty array construct, which both MATLAB and APL support. Empty arrays are legal arrays that contain no data, but still have a shape. To quote from The MathWorks, Inc. [1997]:

> The basic paradigm for empty matrices is that any operation that is defined for $m$-by-$n$ matrices, and that produces a result whose dimension is some function of $m$ and $n$, should still be allowed when $m$ and $n$ are 0. The size of the result should be that same function, evaluated at 0.

An interesting consequence of this paradigm is the ability to conjure up arrays out of nothing. For example, if we were to multiply a $\langle 2, 0\rangle$ matrix by a $\langle 0, 3\rangle$ matrix, the product would be a $\langle 2, 3\rangle$ matrix with all of its elements set to 0.

## 5. A SHAPE ALGEBRAIC SYSTEM

This section lays out a shape calculus for MATLAB that capitalizes on the property of selective rank demotion. The approach is to represent array shapes via shape tuples and to model the language's shape semantics using shape tuple equations. *A shape tuple equation* describes how the shape of a language operator's output is related to the shapes of its inputs when the latter are fixed at certain ranks. This mathematical couching is done by capturing the various possibilities admitted by the operator's shape semantics using a combination of matrix arithmetic and predicates. By virtue of the property of selective rank demotion, shape tuple equations can be regarded as defining a one-to-one correspondence among shape tuple classes. In this manner, they unmask an algebraic structure on the set of shape tuple classes. Algebraic structures are attractive because general properties about them can be characterized. These properties can then be exploited by a code translator, even when explicit information on array shapes is lacking at translation-time.

### 5.1 Shape Tuple Notation Duality

We first need a methodology to manipulate shape tuples. One systematic way of doing this is to overload the notation $\langle p_1, p_2, \dots, p_n\rangle$ to mean an $n \times n$ square diagonal matrix in which each $p_i$ $(1 \le i \le n)$ is a principal diagonal element:

$$\langle p_1, p_2, \dots, p_n\rangle = \begin{pmatrix} p_1 & 0 & \dots & 0 \\ 0 & p_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & p_n \end{pmatrix}. \tag{9}$$

This allows the machinery of matrix arithmetic to be used when working with shape tuples, avoiding to a large extent the need to invent new notation. An example is the determinant $|\boldsymbol{s}|$ that expresses the size of an array whose shape tuple is $\boldsymbol{s}$.

The notation $\langle p_1, p_2, \ldots, p_n \rangle^{\bullet}$ will be used to mean the shape tuple with $\omega$ components that is equivalent to $\langle p_1, p_2, \ldots, p_n \rangle$ by the subtyping $\wp$ in Equation (7):

$$\langle p_1, p_2, \ldots, p_n \rangle^{\bullet} = \begin{pmatrix} p_1 & 0 & \ldots & 0 & 0 & \ldots \\ 0 & p_2 & \ldots & 0 & 0 & \ldots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \\ 0 & 0 & \ldots & p_n & 0 & \ldots \\ 0 & 0 & \ldots & 0 & 1 & \ldots \\ \vdots & \vdots & & \vdots & \vdots & \ddots \end{pmatrix}. \tag{10}$$

From Equation (10), it is clear that $\boldsymbol{s}^{\bullet\bullet} = \boldsymbol{s}^{\bullet}$ for any shape tuple $\boldsymbol{s}$.

### 5.2 Set-of-Shapes Predicates

The shape semantics of expressions in MATLAB are often conditioned on the operands being of certain shapes. For instance, in the MATLAB statement

```
c ← a'
```

the operand a to the complex-conjugate transpose operation has to be a matrix (see Section 4.1.2 for terminology) for the operation to be legal [The MathWorks, Inc. 1997]. We say that the shape semantics in this case is conditioned on a being a matrix.

*A set-of-shapes* (SS) predicate is a formalism that captures such conditions. In general, an SS predicate flags the membership of a shape in a set of shapes. If $S \subseteq \mathbb{S}_{\wp}$, an SS predicate $P_S$ is a mapping from $\mathbb{S}$ to the set $\mathbb{B} = \{0, 1\}$ such that

$$P_S(\boldsymbol{s}) = \begin{cases} 1 & \text{if } \bar{\boldsymbol{s}} \in S, \\ 0 & \text{if } \bar{\boldsymbol{s}} \notin S. \end{cases} \tag{11}$$

#### 5.2.1 *An Extended Discrete Delta Function.*

All SS predicates can be uniformly expressed by using an extension of the discrete delta function $\delta$ (which is usually defined on scalar reals) to square matrices. If $\boldsymbol{s}$ is an $n \times n$ matrix, then $\delta(\boldsymbol{s})$ is either the $n \times n$ identity matrix $\boldsymbol{I}$ or the $n \times n$ zero matrix $\mathbf{0}$, as follows:[12]

$$\delta(\boldsymbol{s}) = \begin{cases} \boldsymbol{I} & \text{if } \boldsymbol{s} = \mathbf{0}, \\ \mathbf{0} & \text{if } \boldsymbol{s} \neq \mathbf{0}. \end{cases} \tag{12}$$

Hence, any SS predicate $P_S$ can be formulated as

$$P_S(\boldsymbol{s}) = |\boldsymbol{\iota}^{\bullet} - \prod_{l} (\boldsymbol{\iota}^{\bullet} - \delta(\boldsymbol{s}^{\bullet} - \boldsymbol{t}_l^{\bullet}))|, \tag{13}$$

where $\boldsymbol{\iota} = \langle 1, 1 \rangle$ is the scalar shape tuple, and $S = \bigcup_l \{\bar{\boldsymbol{t}}_l\}$.

#### 5.2.2 *Some Special Set-of-Shapes Predicates.*

Certain SS predicates occur so often that we use short forms to denote them. These are the *scalar* SS

[12]This extension differs slightly from that in Joisha et al. [2000] and Joisha and Banerjee [2001a].

predicate $\alpha$ that flags a scalar shape, the *legal* SS predicate[13] $\theta$ that indicates a legal shape, and the *matrix* SS predicate $\beta$ that flags a matrix shape:

$$\alpha(\boldsymbol{s}) = P_{\{\overline{\iota}\}}(\boldsymbol{s}) = |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{\iota}^{\cdot})|, \tag{14}$$

$$\theta(\boldsymbol{s}) = P_{\{\overline{\boldsymbol{t}} \,|\, \overline{\boldsymbol{t}} \neq \overline{\boldsymbol{\pi}}\}}(\boldsymbol{s}) = |\boldsymbol{\iota}^{\cdot} - \delta(\boldsymbol{s}^{\cdot} - \boldsymbol{\pi}^{\cdot})|, \tag{15}$$

$$\beta(\boldsymbol{s}) = P_{\{\overline{\boldsymbol{t}} \,|\, \overline{\boldsymbol{t}} \neq \overline{\boldsymbol{\pi}} \wedge \boldsymbol{t}^{\cdot}\boldsymbol{\epsilon}^{\cdot} = \boldsymbol{\epsilon}^{\cdot}\}}(\boldsymbol{s}) = \theta(\boldsymbol{s})|\delta(\boldsymbol{s}^{\cdot}\boldsymbol{\epsilon}^{\cdot} - \boldsymbol{\epsilon}^{\cdot})|. \tag{16}$$

In Equation (16), $\boldsymbol{\epsilon} = \langle 0, 0 \rangle$ is the shape tuple of a particular kind of empty matrix.

## 5.3 Shape Tuple Equations

For certain functions in MATLAB we refer to as Type I, knowing the shapes of the operands suffice to determine the shapes of all their results (some MATLAB functions return more than one result). Consider the MATLAB statement

$$\texttt{c} \leftarrow f\ \texttt{(a, b, ...)},$$

where $f$ is a Type I language operator of arity $m$ ($m \geq 0$). Then, if $\boldsymbol{u}$ denotes the shape tuple of c, and $\boldsymbol{s_i}$ the shape tuple of the $i$th operand ($1 \leq i \leq m$) to $f$, the following shape tuple equation can be shown to always hold among them:

$$\boldsymbol{u}^{\cdot} = \overbrace{(1 - \theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m}))\boldsymbol{\pi}^{\cdot}}^{\text{illegal shape tuple combo}} + \underbrace{\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})\mathcal{H}_f(\boldsymbol{s_1}^{\cdot}, \boldsymbol{s_2}^{\cdot}, \ldots, \boldsymbol{s_m}^{\cdot})}_{\text{legal shape tuple combo}}. \tag{17}$$

To understand Equation (17), we begin by observing that $\boldsymbol{u}$ is always either a legal or illegal shape tuple. Moreover, by the definition of a Type I language operator it should be possible to distinguish these two conditions, given the shape tuples $\boldsymbol{s_i}$ of the operands to $f$. This suggests the existence of a function $\theta_f : \mathbb{S}^m \mapsto \mathbb{B}$ that could be used to perform such a discrimination. Clearly, when all the $\boldsymbol{s_i}$ are legal shape tuples and comply with the shape semantics of $f$, $\boldsymbol{u}$ must also be a legal shape tuple; we signal these situations by setting $\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})$ to 1. When combinations of legal shape tuples don't adhere to the shape requirements of $f$, $\boldsymbol{u}$ must be an illegal shape tuple; we mark these cases by setting $\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})$ to 0. What about the abstract situation in which one or more of the $\boldsymbol{s_i}$ are illegal shape tuples? In this case, we postulate that $\boldsymbol{u}$ is also an illegal shape tuple. Doing so ensures that in the denotational semantics, the information pertaining to a shape error, once engendered, is propagated through the rest of the program from the point of origination. Hence, $\theta_f$ can be formally defined as follows:

$$\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m}) = \begin{cases} 1 & \text{if } \overline{\boldsymbol{s_i}} \neq \overline{\boldsymbol{\pi}} \text{ for all } 1 \leq i \leq m \text{, and} \\ & \text{the } \boldsymbol{s_i}\text{s conform to the shape rules of } f, \\ 0 & \text{otherwise.} \end{cases} \tag{18}$$

The function $\mathcal{H}_f : \mathbb{S}_\omega^m \mapsto \mathbb{S}_\omega$ describes how combinations of legal shapes that accord with the shape semantics of $f$ map to a legal shape. In particular, $\mathbb{S}_\omega$

[13]This is referred to as the *correctness shape predicate* in Joisha et al. [2000] and Joisha and Banerjee [2001a].

is the subset of all shape tuples in $\mathbb{S}$ having $\omega$ components. By limiting the codomain of $\mathcal{H}_f$ to $\mathbb{S}_\omega$, the matrix arithmetic in Equation (17) is guaranteed to be well-defined. By limiting the domain of $\mathcal{H}_f$ to $\mathbb{S}_\omega^m$, we are assured that related shape tuples are mapped the same way by $\mathcal{H}_f$, that is,

$$\mathcal{H}_f(\boldsymbol{s_1}^{\bullet}, \boldsymbol{s_2}^{\bullet}, \ldots, \boldsymbol{s_m}^{\bullet}) = \mathcal{H}_f(\boldsymbol{t_1}^{\bullet}, \boldsymbol{t_2}^{\bullet}, \ldots, \boldsymbol{t_m}^{\bullet}) \tag{19}$$

whenever $\boldsymbol{s_i} \wp \boldsymbol{t_i}$ for all $1 \le i \le m$, because $\boldsymbol{s_i}^{\bullet}$ then equals $\boldsymbol{t_i}^{\bullet}$ for all $i$.

Since the effect of $\mathcal{H}_f(\boldsymbol{s_1}^{\bullet}, \boldsymbol{s_2}^{\bullet}, \ldots, \boldsymbol{s_m}^{\bullet})$ in Equation (17) comes through to the lefthand side only when $\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m}) = 1$, $\mathcal{H}_f$ has free reign on how to map combinations that involve illegal shapes, or combinations of legal shapes, that don't conform to the shape rules of $f$. This is because its contribution in these cases is suppressed by a zero-valued $\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})$. While this implies that a family of $\mathcal{H}_f$ exists for a given $f$, we try to formulate an $\mathcal{H}_f$ that is as simple as possible.

## 5.4 Sample Shape Tuple Equations

Thus, to form Equation (17) for a given $f$, only $\theta_f$ and an $\mathcal{H}_f$ need to be ascertained. This can be done by couching the English descriptions of $f$ in the language of matrix arithmetic. For instance, MATLAB's complex-conjugate transpose operation requires the operand to be a matrix; it is illegal to apply it on a higher-dimensional array. When applied on matrices, it resembles the standard transpose operation in linear algebra. This yields the following $\theta_f$ and $\mathcal{H}_f$:

$$\theta_{'}(\boldsymbol{s}) = \beta(\boldsymbol{s}), \tag{20}$$

$$\mathcal{H}_{'}(\boldsymbol{s}^{\bullet}) = \boldsymbol{\Psi}\boldsymbol{s}^{\bullet}\boldsymbol{\Psi}, \tag{21}$$

where $\boldsymbol{\Psi}$ stands for a particular $\omega \times \omega$ elementary matrix [Banerjee 1993]:

$$\boldsymbol{\Psi} = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 & 0 & \ldots \\ 1 & 0 & 0 & \ldots & 0 & 0 & \ldots \\ 0 & 0 & 1 & \ldots & 0 & 0 & \ldots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \\ 0 & 0 & 0 & \ldots & 1 & 0 & \ldots \\ 0 & 0 & 0 & \ldots & 0 & 1 & \ldots \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \ddots \end{pmatrix}. \tag{22}$$

Elementary matrices arise in connection with the elementary row and column operations in linear algebra [Banerjee 1993]. A multiplication with these matrices results in a *reversal* (multiplication of a row or column by $-1$), an *interchange* (exchange of two rows or columns), or a *skew* (addition of an integer multiple of a row or column to another row or column, respectively). In the case of $\boldsymbol{\Psi}$, a premultiplication causes the first two rows to be interchanged, while a postmultiplication results in the first two columns being interchanged. Thus, the net effect in Equation (21) is to exchange the first two principal diagonal elements in $\boldsymbol{s}^{\bullet}$.

As another illustration of how $\theta_f$ and $\mathcal{H}_f$ may be determined, consider

```
c ← a+b,
```

where $f$ is the array addition operation in MATLAB. The shape semantics of this operation entails at least one operand with the scalar shape or operands with identical shapes. Hence, if $\boldsymbol{s}$ and $\boldsymbol{t}$ are the shape tuples of a and b, respectively, then

$$\theta_+(\boldsymbol{s}, \boldsymbol{t}) = \theta(\boldsymbol{s})\theta(\boldsymbol{t})(\alpha(\boldsymbol{s}) + \alpha(\boldsymbol{t}) + |\delta(\boldsymbol{s}^\centerdot - \boldsymbol{t}^\centerdot)| - 2\alpha(\boldsymbol{s})\alpha(\boldsymbol{t})), \tag{23}$$

$$\mathcal{H}_+(\boldsymbol{s}^\centerdot, \boldsymbol{t}^\centerdot) = \alpha(\boldsymbol{s}^\centerdot)\boldsymbol{t}^\centerdot + (1 - \alpha(\boldsymbol{s}^\centerdot))\boldsymbol{s}^\centerdot. \tag{24}$$

Notice that Equation (23) complies with Equation (18) and that Equation (24) indeed defines a function from $\mathbb{S}_\omega^2$ to $\mathbb{S}_\omega$ that portrays how legal shapes map to a legal shape under the shape semantics of MATLAB's array addition operation.

Table I displays the $\theta_f$ and $\mathcal{H}_f$ for a few Type I built-in functions $f$. For both a discussion on how these were obtained and a longer list, see Joisha et al. [2000]. The symbols $\boldsymbol{\epsilon_1}$ and $\boldsymbol{\epsilon_2}$ in the table stand for $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$, respectively.

5.4.1 *An Example with All Array Extents Known.* Consider the statement

```
c ← a*b
```

and suppose that the shape tuples of a and b are $\boldsymbol{s_1} = \langle 7, 5\rangle$ and $\boldsymbol{s_2} = \langle 5, 6\rangle$, respectively. Then from Equations (12), (14), (15), and (16):

$$\begin{aligned}
&|\delta(\boldsymbol{s_1}^\centerdot\boldsymbol{\epsilon}_{\boldsymbol{1}}^\centerdot - \boldsymbol{\Psi}\boldsymbol{s_2}^\centerdot\boldsymbol{\Psi}\boldsymbol{\epsilon}_{\boldsymbol{1}}^\centerdot)| = |\delta(\langle 0, 5\rangle^\centerdot - \langle 0, 5\rangle^\centerdot)| = |\delta(\boldsymbol{0})| = |\boldsymbol{I}| = 1,\\
&\alpha(\boldsymbol{s_1}) = \alpha(\boldsymbol{s_1}^\centerdot) = \alpha(\boldsymbol{s_2}) = \alpha(\boldsymbol{s_2}^\centerdot) = 0,\\
&\theta(\boldsymbol{s_1}) = \theta(\boldsymbol{s_2}) = 1,\\
&\beta(\boldsymbol{s_1}) = \beta(\boldsymbol{s_2}) = 1.
\end{aligned}$$

Consulting Table I, we thus have

$$\begin{aligned}
&\theta_*(\boldsymbol{s_1}, \boldsymbol{s_2}) = 1,\\
&\mathcal{H}_*(\boldsymbol{s_1}^\centerdot, \boldsymbol{s_2}^\centerdot) = \langle 7, 6\rangle^\centerdot.
\end{aligned}$$

Hence, from Equation (17) the shape tuple of c can be any $\boldsymbol{u}$ such that

$$\boldsymbol{u}^\centerdot = \langle 7, 6\rangle^\centerdot.$$

5.4.2 *An Example with Some Array Extents Unknown.* Consider the statement

```
c ← a+b
```

and suppose that the shape tuples of a and b are $\boldsymbol{s_1} = \langle p_1, 3\rangle$ and $\boldsymbol{s_2} = \langle 2, q_2, q_3\rangle$, respectively, where $p_1$, $q_2$, and $q_3$ are all unknown. Then from Equation (12)

$$|\delta(\boldsymbol{s_1}^\centerdot - \boldsymbol{s_2}^\centerdot)| = \delta(p_1 - 2)\delta(q_2 - 3)\delta(q_3 - 1),$$

where the righthand side $\delta$ is the discrete delta function as applied to integers:

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0,\\ 0 & \text{if } x \neq 0. \end{cases} \tag{25}$$

Table I. $\theta_f$ and $\mathcal{H}_f$ for Some Type I Language Operators in MATLAB

| $f$ | $\theta_f$ | $\mathcal{H}_f$ |
|---|---|---|
| `a * b` | $\theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t}))(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})\lvert\delta(\boldsymbol{s}^{\bullet}\epsilon_1^{\bullet}-\boldsymbol{\Psi}\boldsymbol{t}^{\bullet}\boldsymbol{\Psi}\epsilon_1^{\bullet})\rvert)\big)$ | $\alpha(\boldsymbol{t}^{\bullet})\boldsymbol{s}^{\bullet}+(1-\alpha(\boldsymbol{t}^{\bullet}))\big(\alpha(\boldsymbol{s}^{\bullet})\boldsymbol{t}^{\bullet}+(1-\alpha(\boldsymbol{s}^{\bullet}))(\boldsymbol{s}^{\bullet}\epsilon_2^{\bullet}+\boldsymbol{t}^{\bullet}\epsilon_1^{\bullet}-\iota^{\bullet}\epsilon^{\bullet})\big)$ |
| `a+b` | $\theta(\boldsymbol{s})\theta(\boldsymbol{t})(\alpha(\boldsymbol{s})+\alpha(\boldsymbol{t})+\lvert\delta(\boldsymbol{s}^{\bullet}-\boldsymbol{t}^{\bullet})\rvert-2\alpha(\boldsymbol{s})\alpha(\boldsymbol{t}))$ | $\alpha(\boldsymbol{s}^{\bullet})\boldsymbol{t}^{\bullet}+(1-\alpha(\boldsymbol{s}^{\bullet}))\boldsymbol{s}^{\bullet}$ |
| `a.*b` | | |
| `a.^b` | | |
| `a./b` | | |
| `a.\b` | | |
| `a<b` | | |
| `a\|b` | | |
| `a^b` | $\beta(\boldsymbol{s})\beta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s})\lvert\delta(\boldsymbol{\Psi}\boldsymbol{t}^{\bullet}\boldsymbol{\Psi}-\boldsymbol{t}^{\bullet})\rvert)(1-\alpha(\boldsymbol{t})\lvert\delta(\boldsymbol{\Psi}\boldsymbol{s}^{\bullet}\boldsymbol{\Psi}-\boldsymbol{s}^{\bullet})\rvert)\big)$ | $\alpha(\boldsymbol{s}^{\bullet})\boldsymbol{t}^{\bullet}+(1-\alpha(\boldsymbol{s}^{\bullet}))\boldsymbol{s}^{\bullet}$ |
| `a'` | $\beta(\boldsymbol{s})$ | $\boldsymbol{\Psi}\boldsymbol{s}^{\bullet}\boldsymbol{\Psi}$ |
| `a.'` | | |
| `a/b` | $\theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t}))(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})\lvert\delta(\boldsymbol{s}^{\bullet}\epsilon_1^{\bullet}-\boldsymbol{t}^{\bullet}\epsilon_1^{\bullet})\rvert)\big)$ | $\alpha(\boldsymbol{s}^{\bullet})\boldsymbol{t}^{\bullet}+(1-\beta(\boldsymbol{s}^{\bullet}))\boldsymbol{t}^{\bullet}+(1-\alpha(\boldsymbol{s}^{\bullet}))\beta(\boldsymbol{s}^{\bullet})(\boldsymbol{s}^{\bullet}\epsilon_2^{\bullet}+\boldsymbol{\Psi}\boldsymbol{t}^{\bullet}\boldsymbol{\Psi}\epsilon_1^{\bullet}-\iota^{\bullet}\epsilon^{\bullet})$ |
| `a\b` | $\theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t}))(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})\lvert\delta(\boldsymbol{s}^{\bullet}\epsilon_2^{\bullet}-\boldsymbol{t}^{\bullet}\epsilon_2^{\bullet})\rvert)\big)$ | $\alpha(\boldsymbol{t}^{\bullet})\boldsymbol{s}^{\bullet}+(1-\beta(\boldsymbol{t}^{\bullet}))\boldsymbol{s}^{\bullet}+(1-\alpha(\boldsymbol{s}^{\bullet}))\beta(\boldsymbol{s}^{\bullet})(\boldsymbol{\Psi}\boldsymbol{s}^{\bullet}\boldsymbol{\Psi}\epsilon_2^{\bullet}+\boldsymbol{t}^{\bullet}\epsilon_1^{\bullet}-\iota^{\bullet}\epsilon^{\bullet})$ |
| `[a; b]` | $\theta(\boldsymbol{s})\theta(\boldsymbol{t})\lvert\delta(\boldsymbol{s}^{\bullet}\epsilon_1^{\bullet}-\boldsymbol{t}^{\bullet}\epsilon_1^{\bullet})\rvert$ | $\boldsymbol{s}^{\bullet}+\boldsymbol{t}^{\bullet}(\iota^{\bullet}-\epsilon_1^{\bullet})$ |
| `[a, b]` | $\theta(\boldsymbol{s})\theta(\boldsymbol{t})\lvert\delta(\boldsymbol{s}^{\bullet}\epsilon_2^{\bullet}-\boldsymbol{t}^{\bullet}\epsilon_2^{\bullet})\rvert$ | $\boldsymbol{s}^{\bullet}+\boldsymbol{t}^{\bullet}(\iota^{\bullet}-\epsilon_2^{\bullet})$ |
| `~a` | $\theta(\boldsymbol{s})$ | $\boldsymbol{s}^{\bullet}$ |
| `tan(a)` | | |
| `exp(a)` | | |
| `fix(a)` | | |
| `fft(a)` | | |

From Equations (14) and (15), and the fact that $\pi_1 \neq 2$ and $\pi_2 \neq 3$, we also have

$$\alpha(\boldsymbol{s_1}) = \alpha(\boldsymbol{s_1}^{\bullet}) = \alpha(\boldsymbol{s_2}) = \alpha(\boldsymbol{s_2}^{\bullet}) = 0,$$
$$\theta(\boldsymbol{s_1}) = \theta(\boldsymbol{s_2}) = 1.$$

Therefore from Table I,

$$\theta_+(\boldsymbol{s_1}, \boldsymbol{s_2}) = \delta(p_1 - 2)\delta(q_2 - 3)\delta(q_3 - 1),$$
$$\mathcal{H}_+(\boldsymbol{s_1}^{\bullet}, \boldsymbol{s_2}^{\bullet}) = \langle 2, q_2, q_3\rangle^{\bullet}.$$

Hence, from Equation (17), the shape tuple of c can be any $\boldsymbol{u}$ such that

$$\boldsymbol{u}^{\bullet} = \langle D\pi_1 + (1 - D)2, D\pi_2 + (1 - D)q_2, D + (1 - D)q_3\rangle^{\bullet}, \tag{26}$$

where $D = 1 - \theta_+(\boldsymbol{s_1}, \boldsymbol{s_2}) = 1 - \delta(p_1 - 2)\delta(q_2 - 3)\delta(q_3 - 1)$.

## 5.5 Towards a Congruent Subtyping

An equivalence relation $E$ on some set $S$ is said to have the *substitution property* with respect to a binary operation $\bullet$ that is closed on $S$ if and only if

$$(x\ E\ x') \wedge (y\ E\ y') \Longrightarrow (x \bullet y)\ E\ (x' \bullet y')$$

for all $x, y, x', y' \in S$ [Tremblay and Manohar 1975]. If the righthand side of Equation (17) is viewed as the function $\ddot{f} : \mathbb{S}^m \mapsto \mathbb{S}$, it can be shown that the subtyping $\wp$ has the substitution property with respect to $\ddot{f}$.

THEOREM 3. *Consider two systems of solutions for Equation* (17)*:*

$$\boldsymbol{u}^{\bullet} = \big(1 - \theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})\big)\pi^{\bullet} + \theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})\mathcal{H}_f(\boldsymbol{s_1}^{\bullet}, \boldsymbol{s_2}^{\bullet}, \ldots, \boldsymbol{s_m}^{\bullet}), \tag{27}$$
$$\boldsymbol{v}^{\bullet} = \big(1 - \theta_f(\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_m})\big)\pi^{\bullet} + \theta_f(\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_m})\mathcal{H}_f(\boldsymbol{t_1}^{\bullet}, \boldsymbol{t_2}^{\bullet}, \ldots, \boldsymbol{t_m}^{\bullet}). \tag{28}$$

*In addition, suppose $\boldsymbol{s_i} \wp \boldsymbol{t_i}$ holds for all $1 \le i \le m$. Then $\boldsymbol{u} \wp \boldsymbol{v}$ must also be true.*

PROOF. Because $\boldsymbol{s_i} \wp \boldsymbol{t_i}$ holds for all $1 \le i \le m$, we have

$$\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m}) = \theta_f(\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_m})$$

from Equation (18) and the fact that $\wp$ is a subtyping. That $\boldsymbol{s_i} \wp \boldsymbol{t_i}$ for all $1 \le i \le m$ also gives us $\mathcal{H}_f(\boldsymbol{s_1}^{\bullet}, \boldsymbol{s_2}^{\bullet}, \ldots, \boldsymbol{s_m}^{\bullet}) = \mathcal{H}_f(\boldsymbol{t_1}^{\bullet}, \boldsymbol{t_2}^{\bullet}, \ldots, \boldsymbol{t_m}^{\bullet})$ by Equation (19). Hence, Equations (27) and (28) yield $\boldsymbol{u}^{\bullet} = \boldsymbol{v}^{\bullet}$, thus implying $\boldsymbol{u} \wp \boldsymbol{v}$. □

Equivalence relations such as $\wp$ that satisfy the substitution property with respect to some operation are called *congruences* [Tremblay and Manohar 1975]. Congruences are useful because they expose an algebraic structure among equivalence classes.

To elucidate, let us revisit the MATLAB statement c ← a+b examined in Section 5.4.2, but suppose that the shape tuples of a and b are $\boldsymbol{t_1} = \langle p_1, 3, 1, 1\rangle$ and $\boldsymbol{t_2} = \langle 2, q_2, q_3, 1, 1\rangle$ instead of $\boldsymbol{s_1} = \langle p_1, 3\rangle$ and $\boldsymbol{s_2} = \langle 2, q_2, q_3\rangle$, respectively. Then reworking the steps of the example, we obtain

$$\theta_+(\boldsymbol{t_1}, \boldsymbol{t_2}) = \delta(p_1 - 2)\delta(q_2 - 3)\delta(q_3 - 1),$$
$$\mathcal{H}_+(\boldsymbol{t_1}^{\bullet}, \boldsymbol{t_2}^{\bullet}) = \langle 2, q_2, q_3\rangle^{\bullet},$$

and that the shape tuple of c can be any $\boldsymbol{v}$ such that

$$\boldsymbol{v}^{\boldsymbol{\cdot}} = \langle D'\pi_1 + (1 - D')2, D'\pi_2 + (1 - D')q_2, D' + (1 - D')q_3\rangle^{\boldsymbol{\cdot}}, \tag{29}$$

where $D' = 1 - \delta(p_1 - 2)\delta(q_2 - 3)\delta(q_3 - 1)$. Comparing this with Equation (26), we can therefore conclude that $\boldsymbol{v}$ must be related to the shape tuple $\boldsymbol{u}$ by $\wp$, a conclusion that Theorem 3 presages, since $\boldsymbol{s_1} \wp \boldsymbol{t_1}$ and $\boldsymbol{s_2} \wp \boldsymbol{t_2}$.

## 5.6 An Algebraic Structure on the Set of Shapes

Let $\dot{f}$ be a binary relation from $\mathbb{S}_{\wp}^m$ to $\mathbb{S}_{\wp}$ such that

$$(\overline{\boldsymbol{s_1}}, \overline{\boldsymbol{s_2}}, \ldots, \overline{\boldsymbol{s_m}})\ \dot{f}\ \overline{\boldsymbol{u}} \quad \begin{array}{l}\text{if } \exists \boldsymbol{y} \in \overline{\boldsymbol{u}}, \text{ and for all } 1 \le i \le m, \exists \boldsymbol{x_i} \in \overline{\boldsymbol{s_i}} \\ \text{such that } \boldsymbol{y}^{\boldsymbol{\cdot}} = \ddot{f}(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_m}),\end{array} \tag{30}$$

where the function $\ddot{f} : \mathbb{S}^m \mapsto \mathbb{S}$ stands for the righthand side of Equation (17). Since there always exists a $\boldsymbol{u}$ that honors Equation (17) for any given collection of $\boldsymbol{s_i}$s, $\dot{f}$ is *total*, meaning that for every $z$ in $\mathbb{S}_{\wp}^m$, there exists a $\overline{\boldsymbol{w}}$ in $\mathbb{S}_{\wp}$ such that $(z, \overline{\boldsymbol{w}})$ belongs to $\dot{f}$. In addition, from Theorem 3, $\dot{f}$ must be *single-valued*, since

$$((z, \overline{\boldsymbol{u}}) \in \dot{f}) \wedge ((z, \overline{\boldsymbol{v}}) \in \dot{f}) \Longrightarrow \overline{\boldsymbol{u}} = \overline{\boldsymbol{v}}$$

for all $z \in \mathbb{S}_{\wp}^m$. Hence, $\dot{f}$ defines an $m$-ary *operation* on $\mathbb{S}_{\wp}$.[14] Recall that an *algebraic structure*, also referred to as an *algebraic system* or simply an *algebra*, is a set with one or more *finitary* operations defined on it [Tremblay and Manohar 1975]. Therefore, if $\dot{\mathcal{F}}$ is the set of all *shape operators* corresponding to Type I MATLAB functions, then $\dot{\mathcal{F}}$ and the set-of-shapes $\mathbb{S}_{\wp}$ form the algebraic system $[\mathbb{S}_{\wp}, \dot{\mathcal{F}}]$. Examples of shape operators are $\dot{+}$ and $\dot{*}$, which respectively denote the shape semantics of the + and * operations in MATLAB.

## 5.7 Characterizing the Shape Algebra

The various shape operators in $\dot{\mathcal{F}}$ have a number of interesting properties that can be derived by consulting their $\theta_f$ and $\mathcal{H}_f$ functions. Lemma 1 shows an example (for the proofs of all lemmas in this article, the reader is referred to Joisha [2003]).

LEMMA 1. *The shape operator $\dot{+}$ is associative.*

A number of other similar identities can be characterized. Some of these are listed in Table II (for derivations, see Joisha et al. [2000] and Joisha and Banerjee [2002]). As Table II shows, even though $\dot{*}$ isn't a fully associative operator,[15] it still exhibits certain restrictive forms of association. More importantly, the number of applications of $\dot{*}$ in each of these forms of association matters! For instance, while both $\overline{\langle 1, 2\rangle}\dot{*}\big(\overline{\langle 1, 2\rangle}\dot{*}(\overline{\langle 1, 2\rangle}\dot{*}(\overline{\langle 1, 2\rangle}\dot{*}\overline{\langle 2, 1\rangle}))\big)$ and $\overline{\langle 1, 2\rangle}\dot{*}(\overline{\langle 1, 2\rangle}\dot{*}(\overline{\langle 1, 2\rangle}\dot{*}\overline{\langle 2, 1\rangle}))$ equal $\overline{\boldsymbol{\pi}}$, $\overline{\langle 1, 2\rangle}\dot{*}(\overline{\langle 1, 2\rangle}\dot{*}(\overline{\langle 1, 2\rangle}\dot{*}\overline{\langle 2, 1\rangle}))$ and $\overline{\langle 1, 2\rangle}\dot{*}(\overline{\langle 1, 2\rangle}\dot{*}\overline{\langle 2, 1\rangle})$ aren't the same.

Characterizations involving a combination of shape operators can also be derived [Joisha and Banerjee 2002] and are shown in Table II. We adopt the

[14] Any binary relation that is total and single-valued is a *function*. And any function from $S^m$ to $S$ constitutes an $m$-ary operation on $S$ [Tremblay and Manohar 1975].

[15] As an example, $(\overline{\langle 1, 2\rangle}\dot{*}\overline{\langle 2, 1\rangle})\dot{*}\overline{\langle 3, 2\rangle} = \overline{\langle 3, 2\rangle}$, whereas $\overline{\langle 1, 2\rangle}\dot{*}(\overline{\langle 2, 1\rangle}\dot{*}\overline{\langle 3, 2\rangle}) = \overline{\boldsymbol{\pi}}$.

Table II. Sample Shape Identities for Type I MATLAB Functions

| Homogenous Identities | | Heterogenous Identities |
|---|---|---|
| $\bar{\boldsymbol{\pi}}\dot{+}\bar{\boldsymbol{s}} = \bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{\pi}} = \bar{\boldsymbol{\pi}}$ | (Annihilation) | $\bar{\boldsymbol{s}}\dot{+}(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}) = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}$ |
| $\bar{\boldsymbol{\iota}}\dot{+}\bar{\boldsymbol{s}} = \bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{\iota}} = \bar{\boldsymbol{s}}$ | (Identity) | $(\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}})\dot{*}(\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}) = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}\dot{*}\bar{\boldsymbol{t}}$ |
| $\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{s}} = \bar{\boldsymbol{s}}$ | (Idempotency) | $(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{t}}\dot{+}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}})\dot{*}(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{t}}\dot{+}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}) = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}\dot{*}\bar{\boldsymbol{t}}$ |
| $\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}} = \bar{\boldsymbol{t}}\dot{+}\bar{\boldsymbol{s}}$ | (Commutativity) | $(\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}})\dot{*}\bar{\boldsymbol{t}} = \bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}\dot{*}\bar{\boldsymbol{t}}$ |
| $(\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}})\dot{+}\bar{\boldsymbol{u}} = \bar{\boldsymbol{s}}\dot{+}(\bar{\boldsymbol{t}}\dot{+}\bar{\boldsymbol{u}})$ | (Associativity) | $\bar{\boldsymbol{s}}\dot{*}(\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}) = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}$ |
| $\bar{\boldsymbol{\pi}}\dot{*}\bar{\boldsymbol{s}} = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{\pi}} = \bar{\boldsymbol{\pi}}$ | (Annihilation) | $\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{t}} = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}$ |
| $\bar{\boldsymbol{\iota}}\dot{*}\bar{\boldsymbol{s}} = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{\iota}} = \bar{\boldsymbol{s}}$ | (Identity) | $\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{t}}\dot{+}\bar{\boldsymbol{t}}\dot{*}\bar{\boldsymbol{t}} = \bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}\dot{*}\bar{\boldsymbol{t}}$ |
| $(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}})\dot{*}(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}) = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}$ | | $(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}})\dot{*}(\bar{\boldsymbol{t}}\dot{*}\bar{\boldsymbol{t}}\dot{+}\bar{\boldsymbol{s}}) = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}\dot{*}\bar{\boldsymbol{t}}$ |
| $\bar{\boldsymbol{s}}\dot{*}(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}) = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}$ | | $(\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}\dot{*}\bar{\boldsymbol{t}})\dot{*}\bar{\boldsymbol{t}} = \bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}\dot{*}\bar{\boldsymbol{t}}$ |
| $(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}})\dot{*}\bar{\boldsymbol{s}} = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}$ | (Restrictive Forms of Association) | $\bar{\boldsymbol{s}}\dot{*}(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}) = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}\dot{+}\bar{\boldsymbol{t}}$ |
| $\bar{\boldsymbol{s}}\dot{*}\big(\bar{\boldsymbol{s}}\dot{*}(\bar{\boldsymbol{s}}\dot{*}(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{t}}))\big) = \bar{\boldsymbol{s}}\dot{*}(\bar{\boldsymbol{s}}\dot{*}(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{t}}))$ | | $\dot{\neg}(\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{t}}) = \dot{\neg}\bar{\boldsymbol{t}}\dot{*}\dot{\neg}\bar{\boldsymbol{s}}$ |
| $\big(((\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{t}})\dot{*}\bar{\boldsymbol{t}})\dot{*}\bar{\boldsymbol{t}}\big)\dot{*}\bar{\boldsymbol{t}} = ((\bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{t}})\dot{*}\bar{\boldsymbol{t}})\dot{*}\bar{\boldsymbol{t}}$ | | $\bar{\boldsymbol{s}}\dot{+}\dot{\neg}\bar{\boldsymbol{s}} = \bar{\boldsymbol{s}}\dot{*}\bar{\boldsymbol{s}}$ |

convention that $\dot{+}$, $\dot{*}$, and $\dot{\neg}$ have increasing precedence; this will make the use of parentheses unnecessary whenever operators can be bound according to precedence.

## 6. SYMBOLIC SHAPE EXPRESSIONS AND THEIR AUTOMATIC EVALUATION

This section presents a symbolic evaluator called MULTIPASS-EVAL that automates the shape calculus described in Section 5. MULTIPASS-EVAL takes in a tree representation of a shape expression and returns its symbolically evaluated version. The kernel in MULTIPASS-EVAL is a "single-step" evaluator called ONE-PASS-EVAL that simplifies shape expressions using previously characterized properties of the shape algebra $[\mathbb{S}_\wp, \dot{\mathcal{F}}]$. With the exception of the commutativity and associativity properties, which are handled by special structural transformations, these characterizations are encoded as rewriting rules and are brought to bear by a rewriting process.

### 6.1 Shape Terms

Given the set $\Lambda^*$ of syntactic representations of elements in $\mathbb{S}_\wp$ and a set $\mathcal{S}$ of "shape variable" symbols, we define the set $\Lambda$ of *well-formed* rooted ordered trees over $\mathcal{S}$ and $\Lambda^*$ as the smallest set containing $\mathcal{S}$ and $\Lambda^*$ such that $\dot{f}_n(t_1, t_2, \ldots, t_n)$ is in $\Lambda$ whenever $\dot{f}_n$ is an $n$-ary shape operator in $\dot{\mathcal{F}}$, and $t_i$ is in $\Lambda$ for all $1 \leq i \leq n$. Using grammar productions, this set of shape terms could be inductively defined as

$$\Lambda \ ::= \mathcal{S} \mid \Lambda^* \mid \dot{\mathcal{F}}_1(\Lambda) \mid \dot{\mathcal{F}}_2(\Lambda, \Lambda) \mid \dot{\mathcal{F}}_3(\Lambda, \Lambda, \Lambda) \mid \ \ldots,$$

where $\dot{\mathcal{F}} = \bigcup_{n\geq 1} \dot{\mathcal{F}}_n$, and $\dot{\mathcal{F}}_n$ is the set of all $n$-ary operators in $\dot{\mathcal{F}}$. We also categorize the set of shape variables $\mathcal{S}$ into two *sorts*, namely, the set $\widehat{\mathcal{S}}$ of shape variables that are used to denote *explicit* shape terms (elements in $\Lambda^*$), and the set $\widetilde{\mathcal{S}}$ of shape variables that are used to denote *symbolic* shape terms (elements in $\Lambda - \Lambda^*$):

$$\mathcal{S} \ ::= \widehat{\mathcal{S}} \mid \widetilde{\mathcal{S}}.$$

The goal of this section is to symbolically evaluate elements in $\Lambda$ by rewriting one shape term into another, using whatever may be the known properties of $[\mathbb{S}_\wp, \dot{\mathcal{F}}]$. A *shape property* for our purposes is an equation of the form $l = r$ between two shape terms $l$ and $r$ that is an identity in $[\mathbb{S}_\wp, \dot{\mathcal{F}}]$. This is modeled as either (or both) of the ordered pairs $(l, r)$ and $(r, l)$ in $\Lambda \times \Lambda$. The ordered pair $(l, r)$ will be referred to as a *rewrite rule* and specifically depicted as $l \rightarrow r$ [Dershowitz and Plaisted 2001]. As we shall see later, it isn't necessary that all the properties of $[\mathbb{S}_\wp, \dot{\mathcal{F}}]$ be known, although the more we know, the greater the potential for a "better" inference, that is, an inference that conveys more specialized information. And the greater the specialized nature of an inference, the greater is the degree of tailoring possible in a compiler's or interpreter's generated code.

## 6.2 Rewriting with Rules

We say that a rule $l \rightarrow r$ rewrites a shape term $s$ if there exists a *substitution* [Robinson 1965] of shape variables to shape terms that causes $l$ to *match* $s$. Though a substitution is defined as a mapping $\xi$ from $\mathcal{S}$ to $\Lambda$, it is used after being homomorphically extended to a mapping from $\Lambda$ to $\Lambda$. That is,

$$\xi(\dot{f}_n(t_1, t_2, \ldots, t_n)) = \dot{f}_n(\xi(t_1), \xi(t_2), \ldots, \xi(t_n)) \tag{31}$$

holds for every shape operator $\dot{f}_n$, where each $t_i$ is a shape term. The matching of $l$ to $s$ is a type of unification [Knight 1989]: If $l$ matches $s$, then there exists a substitution $\xi$ such that $\xi(l) = s$.[16] In the event that $l \rightarrow r$ does rewrite $s$, the rewritten shape term is $\xi(r)$, where $\xi$ is the matching substitution used. The particular variant of unification considered in this article is a many-sorted one [Walther 1988] by which different sorts of variables in $\mathcal{S}$ unify only with terms in $\Lambda$ that are of the same sort or a subsort. For instance, an explicit shape variable $\widehat{x}$ in $\widehat{\mathcal{S}}$ unifies only with terms in $\Lambda^*$, while a symbolic shape variable $\widetilde{y}$ in $\widetilde{\mathcal{S}}$ unifies only with terms in $\Lambda - \Lambda^*$. An unqualified shape variable $z$ can unify with any term in $\Lambda$.

As a concrete example, the shape term

$$\overline{\langle 5, 5\rangle} \dot{+} (\overline{\langle 5, 5\rangle} \dot{*} \overline{\langle 5, 5\rangle})$$

can be rewritten into $\overline{\langle 5, 5\rangle} \dot{*} \overline{\langle 5, 5\rangle}$ by the rule

$$x \dot{+} (x \dot{*} x) \rightarrow x \dot{*} x$$

because there exists the substitution $\xi = \{x \hookrightarrow \overline{\langle 5, 5\rangle}\}$ that allows $x \dot{+} (x \dot{*} x)$ to match $\overline{\langle 5, 5\rangle} \dot{+} (\overline{\langle 5, 5\rangle} \dot{*} \overline{\langle 5, 5\rangle})$ (conventionally, substitutions are expressly written as a set of bindings of only those variables that aren't mapped to themselves). The result of the rewrite is $\overline{\langle 5, 5\rangle} \dot{*} \overline{\langle 5, 5\rangle}$ because $\xi(x \dot{*} x) = \overline{\langle 5, 5\rangle} \dot{*} \overline{\langle 5, 5\rangle}$. Observe that this result conveys the fact that adding the product of any pair of $5 \times 5$ matrices with another $5 \times 5$ matrix has the same shape as the product.

[16]We are only concerned with unification over first-order terms, since our shape variables are permitted to range only over shape terms and not over function symbols.

6.2.1 *Uniqueness of Matching Substitutions.* Let $\mathcal{V}(s)$ stand for the set of all variables in a shape term $s$ and let $\Xi$ be the set of all possible *valid*[17] substitutions $\xi : \mathcal{S} \mapsto \Lambda$. Then, as Theorem 4 to follow will show, if $\xi'$ and $\xi''$ are two mappings in $\Xi$ that match $s$ to a target shape term, the *restrictions* of $\xi'$ and $\xi''$ to $\mathcal{V}(s)$ have to be identical. Recall that the restriction of a function $g : X \mapsto Y$ to some subset $X'$ of $X$ is the mapping $h : X' \mapsto Y$ such that $h(x') = g(x')$ for all $x'$ in $X'$ [Tremblay and Manohar 1975]; this is usually designated as $g/X'$.

THEOREM 4. *If $\xi'$ and $\xi''$ are substitutions in $\Xi$ such that $\xi'(s) = \xi''(s)$, then*

$$\xi'/\mathcal{V}(s) = \xi''/\mathcal{V}(s).$$

PROOF. We shall use structural induction to prove this claim. The base case coincides with $s$ belonging to $\Lambda^* \cup \mathcal{S}$, for which the theorem trivially holds. For the inductive step, we shall prove that if the claim holds for a group of $s_i$ $(1 \le i \le n)$ in $\Lambda$, then it will also hold for $\dot{f}_n(s_1, s_2, \ldots, s_n)$, where $\dot{f}_n$ is an operator in $\dot{\mathcal{F}}$.

Suppose $\xi'(\dot{f}_n(s_1, s_2, \ldots, s_n)) = \xi''(\dot{f}_n(s_1, s_2, \ldots, s_n))$. Then from Equation (31),

$$\dot{f}_n(\xi'(s_1), \xi'(s_2), \ldots, \xi'(s_n)) = \dot{f}_n(\xi''(s_1), \xi''(s_2), \ldots, \xi''(s_n)).$$

Hence for all $1 \le i \le n$, we have by the definition of a match,

$$\xi'(s_i) = \xi''(s_i).$$

However, from the induction hypothesis, $\xi'/\mathcal{V}(s_i) = \xi''/\mathcal{V}(s_i)$. Now, by viewing restrictions as binary relations, we can take their union. This gives us

$$\bigcup_{i=1}^{n} \xi'/\mathcal{V}(s_i) = \bigcup_{i=1}^{n} \xi''/\mathcal{V}(s_i). \tag{32}$$

Since any restriction of $g : X \mapsto Y$ to some subset $X_1 \cup X_2$ of $X$ is expressible as $g/X_1 \cup g/X_2$, Equation (32) can be rewritten as

$$\xi'/\bigcup_{i=1}^{n} \mathcal{V}(s_i) = \xi''/\bigcup_{i=1}^{n} \mathcal{V}(s_i). \tag{33}$$

However, $\cup_{i=1}^{n} \mathcal{V}(s_i) = \mathcal{V}(\dot{f}_n(s_1, s_2, \ldots, s_n))$. Therefore, Equation (33) gives us

$$\xi'/\mathcal{V}(\dot{f}_n(s_1, s_2, \ldots, s_n)) = \xi''/\mathcal{V}(\dot{f}_n(s_1, s_2, \ldots, s_n)), \tag{34}$$

which completes the inductive step. Thus $\xi'/\mathcal{V}(s) = \xi''/\mathcal{V}(s)$ for all $s \in \Lambda$. □

6.2.2 *Connecting the Syntactic and Semantic Domains.* Every element in $\mathbb{S}_{\wp}$ has at least one syntactic representation in $\Lambda$, namely, a representation in $\Lambda^*$. For example, $\overline{\langle 4, 5 \rangle}$, $\overline{\langle 4, 5, 1 \rangle}$, and $\overline{\langle 4, 5, 1, 1 \rangle}$ are all representations of the same shape tuple class. An injective mapping $\Psi$ from $\mathbb{S}_{\wp}$ to $\Lambda^*$ can therefore be defined such that $\Psi(\bar{\mathbf{s}})$ is some unique syntactic denotation of $\bar{\mathbf{s}}$. We set this to $\overline{\langle q_1, q_2, \ldots, q_k \rangle}$, where $\langle q_1, q_2, \ldots, q_k \rangle$ is the unique canonical shape tuple corresponding to $\bar{\mathbf{s}}$.

[17] A valid substitution is only allowed to map variables to their corresponding sorts.

Similarly, a surjective function $\psi$ could be defined that maps every *ground* shape term to its unique meaning in $\mathbb{S}_{\wp}$. Ground shape terms, which form a subset $\Lambda_g$ of $\Lambda$, are either explicit shape terms or operations on other ground shape terms:

$$\Lambda_g \ ::= \Lambda^* \mid \dot{\mathcal{F}}_1(\Lambda_g) \mid \dot{\mathcal{F}}_2(\Lambda_g, \Lambda_g) \mid \dot{\mathcal{F}}_3(\Lambda_g, \Lambda_g, \Lambda_g) \mid \ \ldots$$

The rewrite rule in Equation (35) to follow illustrates the usage of these functions. Observe that the shape term $\dot{f}_n(\widehat{x_1}, \widehat{x_2}, \ldots, \widehat{x_n})$ will only match those shape terms whose root is the symbol $\dot{f}_n$ and whose children are elements in $\Lambda^*$. Hence, what the rule in Equation (35) indicates is how a shape term depicting an operation on explicit shape terms can be rewritten to an explicit shape term (note that $\dot{f}_n$ is used syntactically on the lefthand side and semantically on the righthand side).

$$\dot{f}_n(\widehat{x_1}, \widehat{x_2}, \ldots, \widehat{x_n}) \rightarrow \Psi(\dot{f}_n(\psi(\widehat{x_1}), \psi(\widehat{x_2}), \ldots, \psi(\widehat{x_n}))). \tag{35}$$

## 6.3 Rewrite Tuples and an Ordered Rewriter

A *rewrite tuple* is an element in $(\Lambda \times \Lambda)^k$ and thus an ordered collection of rewrite rules. Given the set of all rewrite tuples $\mathcal{R} = \bigcup_{k \geq 1}(\Lambda \times \Lambda)^k$ and a shape term $s$, a transformation called a *rules applicator* could be defined to wholly rewrite $s$:

$$\mathcal{A}(s, (l_1 \rightarrow r_1, l_2 \rightarrow r_2, \ldots, l_k \rightarrow r_k)) = \\ \begin{cases} \xi(r_1) & \text{if } \exists \xi \in \Xi \text{ such that } \xi(l_1) = s, \\ s & \text{else if } k = 1, \\ \mathcal{A}(s, (l_2 \rightarrow r_2, \ldots, l_k \rightarrow r_k)) & \text{otherwise.} \end{cases} \tag{36}$$

The aforementioned rewriting diverges from traditional rewriting processes, such as that in Dershowitz and Plaisted [2001], in two respects: (1) The rules applicator disregards the rewriting of subterms, limiting itself to whole terms, and (2) the choice of rule for rewriting is deterministic. These two provisions enable the realization of a terminating symbolic evaluator that also satisfies, after one additional stipulation, the Church-Rosser property (see Section 6.5.2). Notice that while the choice of rewrite rule in Equation (36) is deterministic, the outcome of the rewrite is still nondeterministic because the particular substitution $\xi$ used is unspecified. This is why $\mathcal{A}$, though a total relation from $\Lambda \times \mathcal{R}$ to $\Lambda$, isn't a single-valued function.

Applications of $\mathcal{A}$ with some specimen rewrite tuples from $\mathcal{R}$ are given next, where $w$, $x$, $y$, and $z$ are shape variables in $\mathcal{S}$, and $\widehat{u}$ and $\widehat{v}$ are shape variables in $\widehat{\mathcal{S}}$:

$$\begin{aligned} &\mathcal{A}((x \dot{+} x) \dot{+} (x \dot{+} x), (z \dot{+} z \rightarrow z, z \dot{+} \bar{\iota} \rightarrow z, \bar{\iota} \dot{+} w \rightarrow w)) = x \dot{+} x, \\ &\mathcal{A}\big(\overline{\langle 2, 3\rangle} \dot{*} \overline{\langle 3, 4\rangle}, \big(\widehat{u} \dot{*} \widehat{v} \rightarrow \Psi(\psi(\widehat{u}) * \psi(\widehat{v}))\big)\big) = \overline{\langle 2, 4\rangle}, \\ &\mathcal{A}(x \dot{+} (y \dot{+} y), (z \dot{+} z \rightarrow z, z \dot{+} \bar{\iota} \rightarrow z, \bar{\iota} \dot{+} z \rightarrow z)) = x \dot{+} (y \dot{+} y). \end{aligned}$$

In the first case, the rule $z \dot{+} z \rightarrow z$ is used with the matching substitution $\{z \hookrightarrow x \dot{+} x\}$. In the second case, the single rule applies with the matching substitution

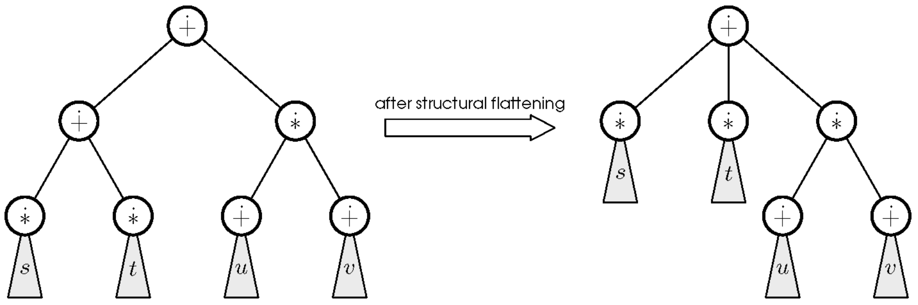

Fig. 7. Structural flattening of a shape term's rooted ordered tree.

$\{\widehat{u} \hookrightarrow \overline{\langle 2,3\rangle}, \widehat{v} \hookrightarrow \overline{\langle 3,4\rangle}\}$. In the third case, no rule matches $x \dot{+} (y \dot{+} y)$. Observe that in the first two cases, the images are unique.

## 6.4 Commutativity and Associativity

The properties of commutativity and associativity are treated specially in our approach and aren't modeled as rules in a rewrite tuple. They are dealt with external to the rules applicator as part of a symbolic evaluator of shape terms (see Section 6.5). This is done so as to achieve a terminating evaluator, which might not have been possible if a rule such as $x \dot{+} y \rightarrow y \dot{+} x$ had been included in the rules applicator's rewrite tuple (in that situation, rewrites may oscillate back and forth between shape terms, resulting in an endlessly applicable rules applicator[18]). In our approach, commutativity is handled by sorting shape terms according to a total order $\sqsubseteq$ on the elements in $\Lambda$, and associativity is managed by a pair of transformations, called SQUASH and UNSQUASH, that essentially "flatten" and "unflatten" shape terms.

6.4.1 *Squashing Shape Terms.* The motivation for SQUASH is that the parentheses in an associative operation are irrelevant. Its workings on $\dot{f}_n(s_1, s_2, \ldots, s_n)$ is a two-step process: (1) *Immediately* nested occurrences of $\dot{f}_n$ in the rooted ordered tree representation of $\dot{f}_n(s_1, s_2, \ldots, s_n)$ are flattened, and (2) the ordered children of the root in the flattened structure are returned as a tuple. An occurrence of $\dot{f}_n$ is considered to be "immediately nested" if its parent is $\dot{f}_n$ and is the root of $s$, or if its parent is itself an immediately nested occurrence of $\dot{f}_n$ in $s$. For instance,

$$\begin{aligned}
&\textsc{squash}(x \dot{+} (y \dot{+} z)) = (x, y, z),\\
&\textsc{squash}(x \dot{*} (y \dot{+} z)) = (x, y \dot{+} z),\\
&\textsc{squash}((x \dot{*} y) \dot{+} (\overline{\langle 2,5\rangle} \dot{+} \overline{\iota})) = (x \dot{*} y, \overline{\langle 2,5\rangle}, \overline{\iota}).
\end{aligned}$$

Why does SQUASH return a tuple of the subterms in the flattened structure, rather than the flattened structure itself? This is because, as shown in Figure 7,

[18]In traditional rewriting, ad hoc extensions are usually used to cope with the associativity and commutativity axioms; the result is often termed *AC rewriting* [Dershowitz and Plaisted 2001].

```
SQUASH($s$)
1   if $root(s) = s$
        ☛ This means $s \in \Lambda^* \cup \mathcal{S}$.
2       return $(s)$
    ☛ Beyond this point, $s$ is of
    ☛ the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$.
3   $\dot{f}_n \leftarrow root(s)$
4   $n \leftarrow arity(\dot{f}_n)$
5   $tuple \leftarrow children(s)$
6   for $i \leftarrow 1$ to $n$ do
7       $s_i \leftarrow component(tuple, i)$
8       if $root(s_i) = \dot{f}_n$
9           $tuple_i \leftarrow$ SQUASH($s_i$)
        else
10          $tuple_i \leftarrow (s_i)$
11  return $concat(tuple_1, tuple_2, \ldots, tuple_n)$
```

```
UNSQUASH($\dot{f}_n$, $tuple$)
    ☛ $tuple$ is of the form
    ☛ $(s_1, s_2, \ldots, s_k)$, where $k \geq 1$
    ☛ and $s_j \in \Lambda$ for all $1 \leq j \leq k$.
1   $k \leftarrow length(tuple)$
2   $n \leftarrow arity(\dot{f}_n)$
3   if $k < n$
4       return $\perp$
5   else if $k = n$
6       $t \leftarrow build(\dot{f}_n, tuple)$
        ☛ $t$ is of the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$.
7       return $t$
8   else if $n > 1$
9       $t \leftarrow build(\dot{f}_n, take(tuple, n))$
        ☛ $t$ is of the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$.
10      $tuple' \leftarrow concat((t), drop(tuple, n))$
11      return UNSQUASH($\dot{f}_n$, $tuple'$)
    else
12      return $\perp$
```

Fig. 8. The SQUASH and UNSQUASH procedures.

such a structure may not be well-formed under the syntactical rules of $\Lambda$ (the flattened structure isn't well-formed because $+$ isn't a ternary operator).

Pseudocode given in Figure 8 provides the algorithmic definition of SQUASH. Lines prefixed by the ☛ character are comments, and procedures such as *root* denote primitives that are documented in Table III. The SQUASH procedure begins by testing whether its argument $s$ is "atomic." If so, the one-component tuple $(s)$ is returned. If the test fails, then $s$ will have to be of the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$. Its root $\dot{f}_n$ and arity $n$ are then determined in Lines 3 and 4. Line 5 assigns the root's children to *tuple*. As indicated in Table III, the *children* primitive returns the ordered children of the root as a tuple, or $\perp$ if the root has no children. Lines 6 to 10 then iteratively extract each child $s_i$ in *tuple*, either recursively squashing it and assigning the result to $tuple_i$ if its root is $\dot{f}_n$, or assigning $(s_i)$ to $tuple_i$ if not. The $n$ tuples thus obtained are concatenated and returned on Line 11.

Because the recursion on Line 9 is always on a smaller shape term, it is easy to see that SQUASH always terminates when invoked on any shape term $s$. Moreover, if $\mathcal{T} = \bigcup_{k \geq 1} \Lambda^k$ is the set of all shape term tuples, it is easy to verify that SQUASH constitutes a function from $\Lambda$ to $\mathcal{T}$. Lemma 2 to follow shows the connection between the arity of a shape term's root and the length of its tuple image under SQUASH.

LEMMA 2. *If $s = \dot{f}_n(s_1, s_2, \ldots, s_n)$, then*

$$length(\text{SQUASH}(s)) = n + m(n-1), \tag{37}$$

*where $m$ is the number of immediately nested occurrences of $\dot{f}_n$ in $s$.*

6.4.2 *"Unsquashing" Shape Terms.* UNSQUASH does the opposite of SQUASH. It rematerializes a shape term from $\dot{f}_n$ and *tuple* by recursively left-associating

Table III. Primitive Procedures

| Invocation Syntax | Description | Complexity |
|---|---|---|
| *arity*$(f)$ | arity of $f$ | $O(1)$ |
| *root*$(t)$ | $f$ if $t = f(t_1, t_2, \ldots, t_n)$; $t$ otherwise | $O(1)$ |
| *iscommutative*$(f)$ | *true* if $f$ is commutative; *false* otherwise | $O(1)$ |
| *isassociative*$(f)$ | *true* if $f$ is associative; *false* otherwise | $O(1)$ |
| *children*$(t)$ | $(t_1, t_2, \ldots, t_n)$ if $t = f(t_1, t_2, \ldots, t_n)$; $\perp$ otherwise | $O(1)$ |
| *build*$(f, (t_1, t_2, \ldots, t_n))$ | $f(t_1, t_2, \ldots, t_n)$ if $f$ is of arity $n$; $\perp$ otherwise | $O(1)$ |
| *length*$(T)$ | $k$, where $T = (t_1, t_2, \ldots, t_k)$ | $O(k)$ |
| *component*$(T, k)$ | $k$th component in $T$ if $1 \leq k \leq$ *length*$(T)$; $\perp$ otherwise | $O(k)$ |
| *take*$(T, k)$ | subtuple of first $k$ components in $T$ if $1 \leq k \leq$ *length*$(T)$; $\perp$ otherwise | $O(k)$ |
| *drop*$(T, k)$ | subtuple after dropping first $k$ components in $T$ if $0 \leq k <$ *length*$(T)$; $\perp$ otherwise | $O(k)$ |
| *concat*$(T_1, T_2, \ldots, T_k)$ | concatenate $T_1$, $T_2$, and so on until $T_k$ $(k \geq 1)$ into a single tuple | $O(k)$ |
| *sort*$(\sqsubseteq, (t_1, t_2, \ldots, t_k))$ | a sorted version of $(t_1, t_2, \ldots, t_k)$, sorted according to the total order $\sqsubseteq$ | $O(\vartheta k \log k)$ |
| *lookup*$(s, subs)$ | lookup the binding $s \hookrightarrow t$ in $subs$ and return $t$ if it exists; $\perp$ otherwise | $O(B)$ |
| *update*$(s \hookrightarrow t, subs)$ | update the current binding of $s$ in $subs$ to $s \hookrightarrow t$ if it exists or insert $s \hookrightarrow t$ if it doesn't | $O(B)$ |
| *substitute*$(t, subs)$ | the result of substituting symbols in $t$ using well-defined bindings available in $subs$ | $O(B\|t\|)$ |

- ❑ $t$, $t_i$ $(i \geq 1)$ stand for rooted ordered trees.
- ❑ $T$, $T_i$ $(i \geq 1)$ stand for ordered tuples.
- ❑ Ordered tuples are stored as linked lists.
- ❑ Tree nodes store their children as linked lists.
- ❑ The value of $\vartheta$ in the asymptotic upper bound for *sort* depends on the time taken to compare a pair of shape terms according to the $\sqsubseteq$ notation.
- ❑ $subs$ is a substitution represented as a linked list of $B$ bindings of variables to terms.
- ❑ $\|t\|$ stands for the number of nodes in $t$.

the shape terms in *tuple* with the $n$-ary shape operator $\dot{f}_n$, or returning $\perp$ if such an association isn't possible. Some examples of UNSQUASH at work are:

$$\text{UNSQUASH}(\dot{+}, (x, y, z)) = (x \dot{+} y) \dot{+} z,$$
$$\text{UNSQUASH}(\dot{*}, (x)) = \perp,$$
$$\text{UNSQUASH}(\dot{+}, (x \dot{*} y, \overline{\langle 2, 5 \rangle}, \bar{\iota})) = ((x \dot{*} y) \dot{+} \overline{\langle 2, 5 \rangle}) \dot{+} \bar{\iota}.$$

Figure 8 also presents the pseudocode for UNSQUASH. Four possibilities can arise during its execution, depending on the number of shape terms $k$ in *tuple* and the arity $n$ of $\dot{f}_n$: (1) If $k < n$, then $\dot{f}_n$ can't be directly applied on the shape terms in *tuple*, so $\perp$ is returned; (2) if $k = n$, the only possible left-association $\dot{f}_n(s_1, s_2, \ldots, s_n)$ is returned, where $tuple = (s_1, s_2, \ldots, s_n)$; (3) if $k > n$, and $n > 1$, UNSQUASH($\dot{f}_n, (\dot{f}_n(s_1, s_2, \ldots, s_n), s_{n+1}, s_{n+2}, \ldots, s_k)$) is returned after associating $\dot{f}_n$ with the first $n$ shape terms; and finally, (4) if $k > n$, but $n = 1$, $\perp$ is returned because an association isn't possible. The third case corresponds to the recursive invocation of UNSQUASH on Line 11. Because the length of *tuple*$'$ is $k - n + 1$, where $n$ is strictly larger than 1, *tuple*$'$ is strictly smaller than *tuple*. Thus, the recursive call is guaranteed to terminate, though it may still return $\perp$. Lemma 3 next presents a necessary and sufficient condition for the third case to always yield a shape term.

LEMMA 3. *UNSQUASH($\dot{f}_n$, tuple) returns a shape term iff*

$$\textit{length}(tuple) = n + m(n-1), \tag{38}$$

*where $\dot{f}_n$ is an n-ary shape operator and m is some nonnegative integer.*

6.4.3 *An Invariant under Squashing and Unsquashing.* From Lemmas 2 and 3, UNSQUASH($\dot{f}_n$, SQUASH($s$)) is always a shape term if $s = \dot{f}_n(s_1, s_2, \ldots, s_n)$. Obviously, UNSQUASH($\dot{f}_n$, SQUASH($s$)) may not be the same as $s$, for instance,

$$\begin{aligned}
&\text{UNSQUASH}\big(\dot{+}, \text{SQUASH}\big(x\dot{+}(y\dot{+}z)\big)\big) = (x\dot{+}y)\dot{+}z,\\
&\text{UNSQUASH}\big(\dot{+}, \text{SQUASH}\big((x\dot{*}y)\dot{+}(z\dot{+}\overline{\langle 3,1,7\rangle})\big)\big) = ((x\dot{*}y)\dot{+}z)\dot{+}\overline{\langle 3,1,7\rangle},\\
&\text{UNSQUASH}\big(\dot{+}, \text{SQUASH}\big((x\dot{*}(x\dot{+}x))\dot{+}(x\dot{+}x)\big)\big) = ((x\dot{*}(x\dot{+}x))\dot{+}x)\dot{+}x.
\end{aligned}$$

In fact, it follows from Lemma 4 that when $n > 1$, UNSQUASH($\dot{f}_n$, SQUASH($s$)) and $s$ will have the same number of immediately nested $\dot{f}_n$ tree nodes. Since SQUASH($s$) only knocks off immediately nested occurrences of $\dot{f}_n$, this also means that both $s$ and UNSQUASH($\dot{f}_n$, SQUASH($s$)) will then have exactly the same nodes, although not necessarily in the same positions, as the previous examples have shown.

LEMMA 4. *If UNSQUASH($\dot{f}_n$, tuple) returns a shape term s, then*

$$(n-1)d = \textit{length}(tuple) - n, \tag{39}$$

*where d is the number of nested occurrences of $\dot{f}_n$ in s that is added by* UNSQUASH.

## 6.5 A Symbolic Evaluator

Figure 9 shows the algorithm called ONE-PASS-EVAL, inspired by the standard evaluation sequence in the Mathematica programming language [Wolfram 1999, 1034], that performs a single-step symbolic evaluation of a shape term. It does this bottom-up, rewriting recursively evaluated subterms using a combination of SQUASH, *sort*, UNSQUASH, and the rules applicator $\mathcal{A}$. It consists of four parts:

(1) Lines 1 and 2 that take care of atomic shape terms. Beyond Line 2, the argument $s$ will be of the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$.
(2) Lines 3 to 9 that recursively apply ONE-PASS-EVAL to each of the subterms $s_i$ and concatenate their single-step evaluated versions $s_i'$ into a single tuple.

```
ONE-PASS-EVAL($s$)
1   if $root(s) = s$
2       return $s$
3   $\dot{f}_n \leftarrow root(s)$
4   $n \leftarrow arity(\dot{f}_n)$
5   $tuple \leftarrow children(s)$
6   for $i \leftarrow 1$ to $n$ do
7           $s_i \leftarrow component(tuple, i)$
8           $s'_i \leftarrow$ ONE-PASS-EVAL($s_i$)
9   $tuple' \leftarrow (s'_1, s'_2, \ldots, s'_n)$
10  if $isassociative(\dot{f}_n)$
11      $tuple' \leftarrow$ SQUASH($build(\dot{f}_n, tuple')$)
12  if $iscommutative(\dot{f}_n)$
13      $tuple' \leftarrow sort(\sqsubseteq, tuple')$
14  if $isassociative(\dot{f}_n)$
15      $t \leftarrow$ UNSQUASH($\dot{f}_n, tuple'$)
    else
16      $t \leftarrow build(\dot{f}_n, tuple')$
17  return $\mathcal{A}(t, \varrho)$
    ☛ $\varrho \in \mathcal{R}$ is a fixed rewrite tuple of the form
    ☛ $(l_1 \rightarrow r_1, l_2 \rightarrow r_2, \ldots, l_\eta \rightarrow r_\eta)$.
```

Fig. 9. A one-pass depth-first evaluator.

```
MULTI-PASS-EVAL($s$)
1   $s' \leftarrow s$
2   do
3           $s'' \leftarrow s'$
4           $s' \leftarrow$ ONE-PASS-EVAL($s'$)
5   until $s' = s''$
6   return $s'$
```

Fig. 10. A multipass evaluator.

(3) Lines 10 to 16 that perform a kind of rewriting on $\dot{f}_n(s'_1, s'_2, \ldots, s'_n)$ using the commutativity and associativity axioms. The outcome $t$ after Line 16 can be thought of as a canonicalized version of $\dot{f}_n(s'_1, s'_2, \ldots, s'_n)$.

(4) Line 17 that rewrites the preceding result by using $\mathcal{A}$ with a fixed rewrite tuple $\varrho$.

The $t$ computed on Lines 15 and 16 is always a shape term. This is because $build(\dot{f}_n, tuple')$ on Line 11 returns the shape term $\dot{f}_n(s'_1, s'_2, \ldots, s'_n)$. Therefore, from Lemma 2, the output of SQUASH on Line 11 will always be a tuple of length $n + m(n-1)$ for some nonnegative integer $m$. Since *sort*, if invoked, will return a tuple having the same length as its argument, $t$ on Line 15 is guaranteed to be a shape term by Lemma 3. Similarly, $t$ on Line 16 is guaranteed to be a shape term because the length of $tuple'$ on that line will always be $n$.

Figure 10 shows MULTIPASS-EVAL, a general-purpose evaluator of shape terms that uses the services of ONE-PASS-EVAL. MULTIPASS-EVAL works by repeatedly applying ONE-PASS-EVAL until two successive applications produce the same result.

Examples of ONE-PASS-EVAL and MULTIPASS-EVAL in action, assuming $\varrho$ to be $(x \dot{+} x \to x, x \dot{*} \overline{\iota} \to x, \overline{\iota} \dot{*} x \to x)$, are:

$$
\begin{aligned}
&\text{ONE-PASS-EVAL}((\overline{\langle 3, 2\rangle} \dot{*} \overline{\langle 1, 1\rangle}) \dot{+} \overline{\langle 3, 2\rangle}) = \overline{\langle 3, 2\rangle} \dot{+} \overline{\langle 3, 2\rangle},\\
&\text{ONE-PASS-EVAL}(\overline{\langle 3, 2\rangle} \dot{+} \overline{\langle 3, 2\rangle}) = \overline{\langle 3, 2\rangle},\\
&\text{ONE-PASS-EVAL}(\overline{\langle 3, 2\rangle}) = \overline{\langle 3, 2\rangle},\\
&\text{MULTIPASS-EVAL}((\overline{\langle 3, 2\rangle} \dot{*} \overline{\langle 1, 1\rangle}) \dot{+} \overline{\langle 3, 2\rangle}) = \overline{\langle 3, 2\rangle},\\
&\text{MULTIPASS-EVAL}(z \dot{+} (\overline{\iota} \dot{*} z)) = z,\\
&\text{MULTIPASS-EVAL}(u \dot{*} v) = u \dot{*} v.
\end{aligned}
$$

6.5.1 *Strong Normalization: A Termination Condition.* A standard method for proving termination of a rewriting process is to show that every rewrite step decreases a term in some sense and that such a decrease can't be indefinite. Formally, this translates to showing the existence of a *well-founded* relation $\rhd$ on the set of terms such that $s \rhd t$ whenever a rewrite step produces a term $t$ from a term $s$ [Dershowitz and Plaisted 2001]. A binary relation $\rhd$ is defined to be well-founded if there are no infinitely decreasing chains of the form $w_0 \rhd w_1 \rhd w_2 \rhd \ldots$ [Mitchell 1996]. Termination follows from the fact that a term can't be rewritten into a different term beyond the end of a chain.

Using $x \unrhd y$ to mean either $x \rhd y$ or $x = y$, Theorem 5 to follow shows that MULTIPASS-EVAL will always terminate if a well-founded $\rhd$ could be fabricated on $\Lambda$ such that $s \unrhd$ ONE-PASS-EVAL$(s)$ for every $s$ in $\Lambda$.[19] By considering the steps in the code for ONE-PASS-EVAL in Figure 9, Lemma 5 presents a set of conditions that if honored by $\rhd$, ensures $s \unrhd$ ONE-PASS-EVAL$(s)$ for every $s$ in $\Lambda$.

THEOREM 5. *If $\rhd$ is a well-founded relation such that $s \unrhd$ ONE-PASS-EVAL$(s)$ for any shape term s, then* MULTIPASS-EVAL *will always terminate.*

PROOF. Since $\rhd$ is well-founded, there can't be an infinitely decreasing chain of the form $s \rhd$ ONE-PASS-EVAL$(s) \rhd$ ONE-PASS-EVAL$^2(s) \rhd \ldots$, where the meaning of ONE-PASS-EVAL$^i(s)$ for nonnegative integers $i$ is

$$
\text{ONE-PASS-EVAL}^i(s) = \begin{cases} \text{ONE-PASS-EVAL}(\text{ONE-PASS-EVAL}^{i-1}(s)) & \text{if } i > 0,\\ s & \text{if } i = 0. \end{cases} \tag{40}
$$

Therefore, ONE-PASS-EVAL$^i(s)$ would have to equal ONE-PASS-EVAL$^{i-1}(s)$ at some point $i > 0$, causing the **do** loop test on Line 5 of MULTIPASS-EVAL to succeed. □

[19]A technicality is that since ONE-PASS-EVAL is based on the rules applicator $\mathcal{A}$, it could be a many-valued function due to the many possible results in $\Lambda$ for a given argument. Therefore, strictly speaking, $s \unrhd$ ONE-PASS-EVAL$(s)$ means that $s \unrhd t$ for *any* image $t$ of ONE-PASS-EVAL$(s)$. Despite the many-valued nature of ONE-PASS-EVAL$(s)$, the termination arguments remain the same.

LEMMA 5. *If $\rhd$ is a transitive binary relation on $\Lambda$ defined such that*

$$\dot{f}_n(s_1, s_2, \ldots, s_n) \trianglerighteq \dot{f}_n(s'_1, s'_2, \ldots, s'_n) \quad \text{if } s_i \trianglerighteq s'_i \text{ for all } 1 \le i \le n, \tag{41}$$

$$\dot{f}_n(s_1, s_2, \ldots, s_n) \trianglerighteq \mathit{build}\big(\dot{f}_n, \mathit{sort}(\sqsubseteq, (s_1, s_2, \ldots, s_n))\big) \quad \text{if } \mathit{isassociative}(\dot{f}_n) = \mathit{false} \wedge \mathit{iscommutative}(\dot{f}_n) = \mathit{true}, \tag{42}$$

$$\dot{f}_n(s_1, s_2, \ldots, s_n) \trianglerighteq \text{UNSQUASH}\big(\dot{f}_n, \text{SQUASH}(\dot{f}_n(s_1, s_2, \ldots, s_n))\big) \quad \text{if } \mathit{isassociative}(\dot{f}_n) = \mathit{true} \wedge \mathit{iscommutative}(\dot{f}_n) = \mathit{false}, \tag{43}$$

$$\dot{f}_n(s_1, s_2, \ldots, s_n) \trianglerighteq \text{UNSQUASH}\big(\dot{f}_n, \mathit{sort}(\sqsubseteq, \text{SQUASH}(\dot{f}_n(s_1, s_2, \ldots, s_n)))\big) \quad \text{if } \mathit{isassociative}(\dot{f}_n) = \mathit{true} \wedge \mathit{iscommutative}(\dot{f}_n) = \mathit{true}, \tag{44}$$

$$\xi(l) \trianglerighteq \xi(r) \text{ for all } \xi \in \Xi \text{ and all } l \to r \text{ in } \varrho, \tag{45}$$

*then $s \trianglerighteq$ ONE-PASS-EVAL$(s)$ for any shape term $s$.*

6.5.1.1 *Fulfilling the Termination Condition.* Does there exist an actual relation $\rhd$ on $\Lambda$ that satisfies the conditions in Lemma 5 and is well-founded at the same time? Clearly, the existence of such a $\rhd$ would also be dependent on which rules are included in $\varrho$. We shall now show that even if every rule $l \to r$ in $\varrho$ models an identity $l = r$ (and *not* $r = l$) in Table II, excluding of course the commutativity and associativity identities, there exists a well-founded $\rhd$ that complies with the requirements of Lemma 5. Our technique for constructing such a $\rhd$ will be to assign "weights" to shape terms according to a function $\varpi : \Lambda \mapsto \mathbb{N}$, where $\mathbb{N}$ is the set of positive integers. We will use the particular weight function

$$\varpi(s) = \begin{cases} c & \text{if } s \in \Lambda^* \cup \mathcal{S}, \\ \varpi(s_1)\varpi(s_2) & \text{if } s = s_1 \dot{*} s_2, \\ c^{\varpi(s_1)} & \text{if } s = \dot{\neg} s_1, \\ 1 + c\sum_{i=1}^{n} \varpi(s_i) & \text{if } s = \mathit{build}\big(\dot{f}_n, \mathit{sort}(\sqsubseteq, \text{SQUASH}(s))\big) \\ & \quad \wedge\ \mathit{isassociative}(\dot{f}_n) = \mathit{false} \\ & \quad \wedge\ \mathit{iscommutative}(\dot{f}_n) = \mathit{true}, \\ 1 + c\sum_{i=1}^{n} \varpi(s_i) & \text{if } s = \text{UNSQUASH}(\dot{f}_n, \text{SQUASH}(s)) \\ & \quad \wedge\ \mathit{isassociative}(\dot{f}_n) = \mathit{true} \\ & \quad \wedge\ \mathit{iscommutative}(\dot{f}_n) = \mathit{false}, \\ 1 + c\sum_{i=1}^{n} \varpi(s_i) & \text{if } s = \text{UNSQUASH}\big(\dot{f}_n, \mathit{sort}(\sqsubseteq, \text{SQUASH}(s))\big) \\ & \quad \wedge\ \mathit{isassociative}(\dot{f}_n) = \mathit{true} \\ & \quad \wedge\ \mathit{iscommutative}(\dot{f}_n) = \mathit{true}, \\ c + c\sum_{i=1}^{n} \varpi(s_i) & \text{else if } s = \dot{f}_n(s_1, s_2, \ldots, s_n), \end{cases} \tag{46}$$

where $c$ is some constant integer greater than 1, to construct $\rhd$ as follows:

$$s \rhd t \Leftrightarrow \varpi(s) > \varpi(t). \tag{47}$$

*Transitivity and Proof of Satisfaction of Equation* (41). To prove that the preceding relation honors Lemma 5, we first need to show that it is transitive.

This is obvious from Equation (47). We next need to show that it satisfies Equation (41). Suppose $s = \dot{f}_n(s_1, s_2, \ldots, s_n)$ and $s' = \dot{f}_n(s'_1, s'_2, \ldots, s'_n)$ and let $s_i \trianglerighteq s'_i$ for all $1 \leq i \leq n$. Since Equation (41) is obvious if $s_i = s'_i$ for all $1 \leq i \leq n$, let there be at least one $i$, say $j$, such that $s_j \rhd s'_j$. If $\dot{f}_n \neq \dot{*}, \dot{\neg}$, then

$$\varpi(s) = \begin{cases} 1 + c\sum_{i=1}^{n} \varpi(s_i), \\ c + c\sum_{i=1}^{n} \varpi(s_i), \end{cases}$$

$$\varpi(s') = \begin{cases} 1 + c\sum_{i=1}^{n} \varpi(s'_i), \\ c + c\sum_{i=1}^{n} \varpi(s'_i). \end{cases}$$

The difference $\varpi(s) - \varpi(s')$ can then only be one of the following three possibilities:

$$\varpi(s) - \varpi(s') = \begin{cases} c\sum_{i=1}^{n}(\varpi(s_i) - \varpi(s'_i)), \\ (c-1) + c\sum_{i=1}^{n}(\varpi(s_i) - \varpi(s'_i)), \\ -(c-1) + c\sum_{i=1}^{n}(\varpi(s_i) - \varpi(s'_i)). \end{cases} \tag{48}$$

Since $s_j \rhd s'_j$, we have $\varpi(s_j) > \varpi(s'_j)$ from Equation (47). This implies

$$c(\varpi(s_j) - \varpi(s'_j)) \geq c.$$

Since we also know that $s_i \trianglerighteq s'_i$ for all $1 \leq i \leq n$, we can therefore conclude that

$$c\sum_{i=1}^{n}(\varpi(s_i) - \varpi(s'_i)) \geq c.$$

This means $\varpi(s) - \varpi(s')$ in Equation (48) is always positive. The remaining case is when $\dot{f}_n$ is either $\dot{*}$ or $\dot{\neg}$. From Equation (46), we then have

$$\varpi(s) = \varpi(s_1)\varpi(s_2) > \varpi(s'_1)\varpi(s'_2) = \varpi(s'),$$
$$\varpi(s) = c^{\varpi(s_1)} > c^{\varpi(s'_1)} = \varpi(s'),$$

whenever $\varpi(s_i) \geq \varpi(s'_i)$ for all $1 \leq i \leq n$ and $\varpi(s_j) > \varpi(s'_j)$.

*Proof of Satisfaction of Equations* (42) *to* (44). It is straightforward to show that $\rhd$ fulfills each of the Equations (42) to (44). For example, let $\dot{f}_n$ be a nonassociative and commutative shape operator and let

$$\dot{f}_n(s_1, s_2, \ldots, s_n) \neq \mathit{build}\big(\dot{f}_n, \mathit{sort}(\sqsubseteq, (s_1, s_2, \ldots, s_n))\big).$$

This gives us the following weights from Equation (46):

$$\varpi(\dot{f}_n(s_1, s_2, \ldots, s_n)) = c + c\sum_{i=1}^{n} \varpi(s_i),$$
$$\varpi(\mathit{build}(\dot{f}_n, \mathit{sort}(\sqsubseteq, (s_1, s_2, \ldots, s_n)))) = 1 + c\sum_{i=1}^{n} \varpi(s_i).$$

Thus from the previous equations, $\varpi(\dot{f}_n(s_1, s_2, \ldots, s_n)) > \varpi(\mathit{build}(\dot{f}_n, \mathit{sort}(\sqsubseteq, (s_1, s_2, \ldots, s_n))))$.

Table IV. Rules from Table II with Differences

| $l \to r$ | Lower Bound on $\varpi(\xi(l)) - \varpi(\xi(r))$ |
|---|---|
| $\overline{\pi} \dotplus s \to \overline{\pi}$ | $c(X + c - 1) + 1$ |
| $s \dotplus \overline{\pi} \to \overline{\pi}$ | $c(X + c - 1) + 1$ |
| $\overline{\iota} \dotplus s \to s$ | $(c - 1)X + c^2 + 1$ |
| $s \dotplus \overline{\iota} \to s$ | $(c - 1)X + c^2 + 1$ |
| $s \dotplus s \to s$ | $(2c - 1)X + 1$ |
| $\overline{\pi} \dot{*} s \to \overline{\pi}$ | $c(X - 1)$ |
| $s \dot{*} \overline{\pi} \to \overline{\pi}$ | $c(X - 1)$ |
| $\overline{\iota} \dot{*} s \to s$ | $(c - 1)X$ |
| $s \dot{*} \overline{\iota} \to s$ | $(c - 1)X$ |
| $(s \dot{*} s) \dot{*} (s \dot{*} s) \to s \dot{*} s$ | $X^2(X^2 - 1)$ |
| $s \dot{*} (s \dot{*} s) \to s \dot{*} s$ | $X^2(X - 1)$ |
| $(s \dot{*} s) \dot{*} s \to s \dot{*} s$ | $X^2(X - 1)$ |
| $s \dot{*} (s \dot{*} (s \dot{*} (s \dot{*} t))) \to s \dot{*} (s \dot{*} (s \dot{*} t))$ | $X^3Y(X - 1)$ |
| $(((s \dot{*} t) \dot{*} t) \dot{*} t) \dot{*} t \to ((s \dot{*} t) \dot{*} t) \dot{*} t$ | $XY^3(Y - 1)$ |
| $s \dotplus s \dot{*} s \to s \dot{*} s$ | $(c - 1)X^2 + cX + 1$ |
| $(s \dotplus t) \dot{*} (s \dotplus t) \to s \dot{*} s \dotplus t \dot{*} t$ | $(c - 1)(cX^2 + cY^2 + 1) + 2c(X + Y + cXY)$ |
| $(s \dotplus t) \dot{*} t \to s \dotplus t \dot{*} t$ | $cX(Y - 1) + Y - c$ |
| $s \dot{*} (s \dotplus t) \to s \dot{*} s \dotplus t$ | $cY(X - 1) + X - c$ |
| $s \dot{*} s \dotplus s \dot{*} t \to s \dot{*} s \dotplus t$ | $c(Y(X - 1) - 1) + 1$ |
| $s \dot{*} t \dotplus t \dot{*} t \to s \dotplus t \dot{*} t$ | $c(X(Y - 1) - 1) + 1$ |
| $(s \dot{*} s \dotplus t) \dot{*} (t \dot{*} t \dotplus s) \to s \dot{*} s \dotplus t \dot{*} t$ | $1 + c(X + Y + cXY + cX^2Y^2 + cX^3 + cY^3 - 1)$ |
| $(s \dotplus t \dot{*} t) \dot{*} t \to s \dotplus t \dot{*} t$ | $c(X + Y^2)(Y - 1) + Y - c$ |
| $s \dot{*} (s \dot{*} s \dotplus t) \to s \dot{*} s \dotplus t$ | $c(X^2 + Y)(X - 1) + X - c$ |
| $\dot{\neg}(s \dot{*} t) \to \dot{\neg} t \dot{*} \dot{\neg} s$ | $c^{XY} - c^{X+Y}$ |
| $s \dotplus \dot{\neg} s \to s \dot{*} s$ | $1 + cX + c^{X+1} - X^2$ |
| $\widehat{u} \dot{*} \widehat{v} \to \Psi(\psi(\widehat{u}) \dot{*} \psi(\widehat{v}))$ | $c^2 - c$ |
| $\widehat{u} \dotplus \widehat{v} \to \Psi(\psi(\widehat{u}) \dotplus \psi(\widehat{v}))$ | $1 + 2c^2 - c$ |
| $\dot{\neg} \widehat{u} \dot{*} \widehat{v} \to \Psi(\dot{\neg} \psi(\widehat{u}))$ | $c^c - c$ |

❑ $X$ and $Y$ stand for $\varpi(\xi(s))$ and $\varpi(\xi(t))$ in the tabulated differences.

*Proof of Satisfaction of Equation* (45). Consider the rules $l \to r$ in Table IV that model every identity $l = r$ from Table II, as well as some more. The additional rules, shown as a bottom section in the table, describe how operations on explicit shape terms map to explicit shape terms. To validate that $\rhd$ honors Equation (45) even when all of these rules are included in the $\varrho$ used by ONE-PASS-EVAL, we shall compute the weights of $l$ and $r$ and show that their difference $\varpi(\xi(l)) - \varpi(\xi(r))$ is positive under any substitution $\xi \in \Xi$. The second column in Table IV gives lower bounds on these computed differences. For instance, for the heterogenous identity $(s \dotplus t) \dot{*} (s \dotplus t) = s \dot{*} s \dotplus t \dot{*} t$ in Table II,

$$\varpi(\xi(l)) = \varpi((\xi(s) \dotplus \xi(t)) \dot{*} (\xi(s) \dotplus \xi(t))) = \varpi((x \dotplus y) \dot{*} (x \dotplus y)),$$
$$\varpi(\xi(r)) = \varpi(\xi(s) \dot{*} \xi(s) \dotplus \xi(t) \dot{*} \xi(t)) = \varpi(x \dot{*} x \dotplus y \dot{*} y),$$

where $x$ and $y$ are shorthands for $\xi(s)$ and $\xi(t)$. But from Equation (46),

$$\varpi((x \dot{+} y) * (x \dot{+} y)) = \begin{cases} (1 + c\varpi(x) + c\varpi(y))^2, \\ (c + c\varpi(x) + c\varpi(y))^2, \end{cases}$$

$$\varpi(x \dot{*} x \dot{+} x \dot{*} x) = \begin{cases} 1 + c\varpi(x)^2 + c\varpi(y)^2, \\ c + c\varpi(x)^2 + c\varpi(y)^2. \end{cases}$$

From this, and using $X$ and $Y$ as shorthands for $\varpi(x)$ and $\varpi(y)$, we obtain

$$\begin{aligned} \varpi(\xi(l)) - \varpi(\xi(r)) &= \varpi((x \dot{+} y) * (x \dot{+} y)) - \varpi(x \dot{*} x \dot{+} x \dot{*} x) \\ &\geq (1 + cX + cY)^2 - (c + cX^2 + cY^2) \\ &= (c - 1)(cX^2 + cY^2 - 1) + 2c(X + Y + cXY). \end{aligned}$$

However, both $X$ and $Y$ are at least $c$ by Equation (46). Therefore, since $c$ is greater than 1, the last aforementioned expression is clearly positive. It can be similarly verified that all of the other weight-difference lower bounds in Table IV are positive.[20] For the last three rules, the lower bounds are obtained by noting that $\xi(\widehat{u})$, $\xi(\widehat{v})$, and the rules' righthand sides, are shape terms in $\Lambda^*$ for any substitution $\xi \in \Xi$.

*Proof of Satisfaction of the Termination Condition.* For the fulfillment of Theorem 5, all that remains is to show that $\rhd$ is well-founded. This is obvious, since from Equation (46), $\varpi(s) \geq c$ for any shape term $s$ and because on positive integers, the arithmetic ordering $<$ is known to be well-founded.

6.5.1.2 *Offline Verification of Termination.* Coming up with a binary relation $\rhd$ that is both well-founded and in agreement with the requirements in Lemma 5 is a nontrivial task, as exemplified by the exercise in Section 6.5.1.1. It is easy to arrive at a well-founded $\rhd$ that meets *some* of the requirements in Lemma 5, but from our experience (which involved a couple of iterations on the weight function's design), attempting to meet all of them simultaneously is a challenge.

[20]As an example, how can we show that $1 + cX + c^{X+1} - X^2$ (the lower bound on $\varpi(\xi(s \dot{+} \dot{\neg} s)) - \varpi(\xi(s * s))$, where $X = \varpi(\xi(s))$) is positive when $c$ is an integer greater than 1? This can be done by first finding the derivative of $Z = 1 + cX + c^{X+1} - X^2$ with respect to $X$:

$$\frac{dZ}{dX} = c + c^{X+1} \ln c - 2X. \tag{49}$$

By differentiating the preceding equation once more with respect to $X$, we have

$$\frac{d^2Z}{dX} = c^{X+1}(\ln c)^2 - 2. \tag{50}$$

Because $X \geq c$, we get $c^{X+1}(\ln c)^2 \geq c^{c+1}(\ln c)^2 \geq 3.84362$. Therefore, $\frac{d^2Z}{dX}$ in Equation (50) is always positive. This means that $\frac{dZ}{dX}$ is an increasing function of $X$. Also, $\frac{dZ}{dX}$ at $X = c$ is

$$c + c^{c+1} \ln c - 2c \geq 3.54518. \tag{51}$$

Hence, $\frac{dZ}{dX}$ is always positive. Thus $Z$ is also an increasing function of $X$. Because $Z$ at $X = c$ is

$$1 + c^2 + c^{c+1} - c^2 = 1 + c^{c+1}, \tag{52}$$

we can therefore conclude that $1 + cX + c^{X+1} - X^2$ is always positive.

Luckily, the weight function in Equation (46) is proactive in the sense that it is defined for all shape operators in $\dot{\mathcal{F}}$. Thus, as more rules characterizing other shape properties are added to $\varrho$, the only condition that we risk breaking is Equation (45). This breakage can be tackled in two ways: (1) Come up with a new weight function or perhaps even better, a new formulation for $\rhd$; or (2) accept only those characterizations $l \rightarrow r$ that pass Equation (45) and reject others. Either way, observe that the testing of all these requirements can be done once, and offline, before the evaluator is actually used in any interpretation or compilation run.

Another point is that it isn't necessary to know the particular total order $\sqsubseteq$ used in *sort*, or the way UNSQUASH associates shape terms, for us to come up with a $\rhd$ that ensures MULTIPASS-EVAL's termination. Thus, rather than returning a left-associated tree, UNSQUASH can choose any other association order as its canonical form and Equation (47) will still define a well-founded relation that honors Theorem 5. This affords some flexibility to an implementation.

6.5.2 *Unique Normalization: The Church-Rosser Property.* The next question is whether MULTIPASS-EVAL(s) always produces the same result for a given shape term $s$ upon termination. The answer is "no" because while all the primitives used in ONE-PASS-EVAL are functions in the mathematical sense (as are the SQUASH and UNSQUASH transformations), the rules applicator used in Line 17 of Figure 9 isn't, as discussed in Section 6.3. How can $\mathcal{A}$ be made a function? One way of achieving this, shown by Lemma 6 to follow, is to restrict $\mathcal{A}$ to a certain subset of $\mathcal{R}$.

LEMMA 6. *Let $\acute{\mathcal{R}}$ be a subset of $\mathcal{R}$ such that*

$$\acute{\mathcal{R}} = \{(l_1 \rightarrow r_1, l_2 \rightarrow r_2, \ldots, l_k \rightarrow r_k) \mid (l_1 \rightarrow r_1, l_2 \rightarrow r_2, \ldots, l_k \rightarrow r_k) \in \mathcal{R} \wedge \mathcal{V}(l_i) \supseteq \mathcal{V}(r_i) \textit{ for all } 1 \leq i \leq k\}. \quad (53)$$

*Then the restriction of $\mathcal{A}$ to $\acute{\mathcal{R}}$, denoted as $\acute{\mathcal{A}}$, is a function.*

Since $\mathcal{V}(l) \supseteq \mathcal{V}(r)$ for every rule $l \rightarrow r$ in Table IV, it follows that even if all the rules in Table IV are included in the fixed rewrite tuple $\varrho$, the outcome of MULTIPASS-EVAL($s$) is still unique upon termination.

6.5.3 *Soundness.* A transformational system is *sound* if it is always meaning-preserving under conversions. In other words, if such a system converts a representation $A$ with meaning $M$ to a representation $B$, then $B$ will also have the same meaning $M$. Hence, to prove that MULTIPASS-EVAL is sound, we have to demonstrate that MULTIPASS-EVAL($s$) and $s$ are algebraically equivalent with respect to the rules in $\varrho$. This can be shown by considering the subset $\Xi_g$ of substitutions that are valid mappings from $\mathcal{S}$ to the set $\Lambda_g$ of ground shape terms, and then proving that the meanings of $\xi_g$(ONE-PASS-EVAL($s$)) and $\xi_g(s)$, where $\xi_g$ is any substitution in $\Xi_g$, are always the same for any shape term $s$ [Joisha 2003].

6.5.4 *Complexity Analysis.* If the termination condition in Section 6.5.1 is satisfied, then for a certain total order $\sqsubseteq$, MULTIPASS-EVAL($s$) can be shown to run in $O(R\|s\|^3 \log \|s\|)$ time, where $\|s\|$ is the number of nodes in the rooted ordered tree representing the shape term $s$, and $R$ is an upper bound on an array's canonical rank [Joisha 2003].[21] This figure can be derived by choosing linked lists as the data structure for ordered tuples, tree-node children, and substitutions, although as shown in Joisha [2003], it isn't determined so much by this choice of data structure than by the fact that a comparison sort of $k$ elements takes $O(\vartheta k \log k)$ time, where $\vartheta$ is an upper bound on the time taken to compare a pair of elements.

## 6.6 Mapping MATLAB Expressions to Their Shapes

Let $\varphi$ be a *shape map* that associates MATLAB variables with their shape terms in $\Lambda$ at a given point in a program. Like substitutions (see Section 6.2), $\varphi$ can be homomorphically extended to a function from $\mathbb{M}$ to $\Lambda$, where $\mathbb{M}$ is the syntactic set of MATLAB expressions. That is, if $f_n$ is an $n$-ary Type I MATLAB function, then

$$\varphi(f_n(e_1, e_2, \ldots, e_n)) = \dot{f}_n(\varphi(e_1), \varphi(e_2), \ldots, \varphi(e_n)), \tag{54}$$

where $\dot{f}_n$ is the shape operator in $\dot{\mathcal{F}}$ corresponding to $f_n$. For instance, $\varphi(\mathtt{a} + \mathtt{b})$ would be $\varphi(\mathtt{a})\dot{+}\varphi(\mathtt{b})$, and $\varphi(\mathtt{a} + \mathtt{b} * \mathtt{c})$ would be $\varphi(\mathtt{a})\dot{+}\varphi(\mathtt{b})\dot{*}\varphi(\mathtt{c})$.

What should $\varphi(e)$ be when $e$ is a $\phi$-function instance of the form $\phi(e_1, e_2, \ldots, e_m)$? To answer this, we observe that when $\mathtt{c} \leftarrow \phi(e_1, e_2, \ldots, e_m)$ is executed, the variable $\mathtt{c}$ will be set to one of the $e_i$ expressions. Here, $m$ is the number of immediate predecessor nodes to the join node housing the $\phi$-function assignment [Cytron et al. 1991]. Exactly which $e_i$ is assigned to $\mathtt{c}$ depends on the edge chosen to reach the join node at runtime. Thus, while a $\phi$-function of the form $\phi(e_1, e_2, \ldots, e_m)$ isn't a mathematical function because different instances may not yield the same $e_i$, it can be made into one by *gating* it with an integer-valued runtime selector $P$:

$$\mathtt{c} \leftarrow \phi(P, e_1, e_2, \ldots, e_m). \tag{55}$$

Every join node in a CFG can be considered to have its own selector. If it has $m$ predecessors, the selector assumes a value from 1 to $m$ signifying the particular predecessor from which control flows into it at runtime. This value may change in the course of execution as join nodes get revisited from different predecessors. Therefore, while a gated $\phi$-function can be viewed as an imaginary MATLAB operation, it isn't of the Type I category because merely knowing the shapes of $P$ and all the $e_i$ in $\phi(P, e_1, e_2, \ldots, e_m)$ isn't sufficient for establishing the shape of the outcome. The value assumed by $P$ would also have to be known in order to make that determination. However, if we were to *curry* [Mitchell 1996] the $\phi$-function on $P$, we would then obtain a pseudo language operator that has Type I characteristics:

$$\Phi(P)(e_1, e_2, \ldots, e_m) = \phi(P, e_1, e_2, \ldots, e_m). \tag{56}$$

[21]From the discussions in Section 6.5.1 and Section 6.5.2, we know that a $\sqsubseteq$ can be chosen without affecting either the termination or the Church-Rosser traits of MULTIPASS-EVAL.

Since each $e_i$ in Equation (56) is a syntactic MATLAB expression in $\mathbb{M}$, $\Phi(P)$ constitutes a Type I MATLAB operation from $\mathbb{M}^m$ to $\mathbb{M}$, albeit a virtual one. The $\Phi$ itself is a *higher-order function* [Mitchell 1996] from a subset of the set of all positive integers $\mathbb{N}$ to the set of all possible mappings from $\mathbb{M}^m$ to $\mathbb{M}$. Therefore, there exists an $m$-ary shape operator $\dot{\Phi}(P)$ that corresponds to $\Phi(P)$, which gives us the following characterization for the shape of c in $\mathtt{c} \leftarrow \phi(P, e_1, e_2, \ldots, e_m)$:

$$\varphi(\mathtt{c}) = \dot{\Phi}(P)(\varphi(e_1), \varphi(e_2), \ldots, \varphi(e_m)). \tag{57}$$

## 7. APPLYING SYMBOLIC EVALUATION TO SHAPE ANALYSIS

This section presents two applications of MULTIPASS-EVAL to shape analysis. The first, which we refer to as *shape tracking*, determines whether an expression's shape follows the shape of another expression. The second, called *preallocation*, determines the set of all possible shapes that an expression might assume at runtime.

### 7.1 Shape Tracking

Let p and q be variables defined in a MATLAB program at the assignment statements $X$ and $Y$. We say that the shape of q *tracks* the shape of p if:

(1) $Y$ is dominated by $X$ and $\varphi(\mathrm{q}) = \varphi(\mathrm{p})$; or
(2) $Y$ lies on the dominance frontier of $X$, is of the form $\mathtt{c} \leftarrow \Phi(P)(\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_m)$, and all the variables $\mathrm{v}_i$ $(1 \leq i \leq m)$ have the same shape $\varphi(\mathrm{p})$.

A point $x$ in a program is said to *dominate* a point $y$ if every possible path in the program's control-flow graph (CFG) from the entry node to $y$ passes through $x$ [Muchnick 1997]. Note that the aforementioned definition of shape tracking doesn't mandate the constancy of tracked shapes with time. Also, for the definition to be applicable, it isn't necessary for the CFG to be in the SSA form because Condition 1 can hold in any CFG. However, when given a CFG in the SSA form, Condition 2 encompasses additional cases not covered by the first. This is because when $Y$ lies on the dominance frontier[22] of $X$, it won't be strictly dominated by $X$, even though the q defined at $Y$ might always have the same shape as the p defined at $X$.

7.1.1 *Optimizations Empowered by Tracked Shapes.* At the source-level, the execution of an array expression could be thought of as a two-phase process: a *preprocessing phase* in which it is examined and prepared for evaluation, and an *evaluation phase* in which all the elements of its array result are computed [Wiedmann 1979]. The main characteristic of the preprocessing phase is that it is composed of subtasks that can be generally performed in a time that is independent of the number of elements in the array arguments to the

[22]The dominance frontier of a CFG node $x$ is the set of all CFG nodes $y$ such that $x$ dominates an immediate predecessor of $y$, but doesn't strictly dominate $y$ [Muchnick 1997].

expression. It normally spans four subtasks:

(1) Verifiing the conformance of the expression's operands (if any) to the requirements of the expression's operators.
(2) Determining the nature, that is, the type, of the result (if any).
(3) Allocating, or reallocating storage for the result.
(4) Dispatching control to one among various subroutines, each of which calculates the result for some specific type configuration of the arguments.

Tracked shapes can render some or all of these subtasks unnecessary, even when shapes aren't explicitly known. Furthermore, knowledge that certain operands have the same shapes may allow the use of specialized routines in the operation's evaluation phase. For instance, consider the subtask of verifying whether the expression that computes q is shape-conforming. Since this amounts to testing whether $\varphi$(q) is $\overline{\pi}$, the verification step becomes redundant if q tracks the shape of p *and* p has already been tested for shape conformance. Similarly, the subtask of determining q's shape may be skipped because it is known once p's shape is determined. Also, if p is dead by the time control reaches $Y$, then storage allocation for q could be sidestepped, since p's storage is available for reuse at $Y$ [Joisha and Banerjee 2003a]. Finally, it may even be possible to simplify or eliminate the dispatch logic in the preprocessing phase. As an example, suppose that the assignment statement $Y$ is

```
q ← a+b
```

and that the shapes tracked by a and b are identical. Then, the dispatch logic doesn't have to test whether a and b are scalars. Control can be directly transferred to a simple evaluation loop that computes elements in q by stepping through each element in the array a and adding the corresponding elements in array b.

7.1.2 *The Shape-Tracking Algorithm.* Because every use in the SSA translation of a program is either dominated by its definition or lies in a $\phi$-function whose ancestor is dominated by the definition, tracked shapes could be searched for by first projecting assignments in the SSA form onto the shape domain using $\varphi$, and then testing the obtained shape expressions for equality after canonicalizing them with MULTIPASS-EVAL. SHAPE-TRACKING, shown in Figure 11, does this by making use of seven new primitives, explained in the following, in addition to those listed in Table III:

—*SSA*($G$) returns an SSA form of the CFG $G$.
—*assignments*($G'$) is the set of all assignment statements in $G'$.
—*lhs*(*astmt*) returns the lefthand side of the assignment *astmt*.
—*DF*($a_1$) is the set of statements that form the dominance frontier of $a_1$.
—*SDOM*($a_1$) is the set of statements strictly dominated by $a_1$.
—*DFS*($N, G, F$) performs a depth-first traversal starting at node $N$ in a graph $G$, invoking $F(v, G)$ on each node $v$ encountered along the way.

```
SHAPE-TRACKING($G$)
    $G' \leftarrow SSA(G)$
    FORM-SHAPE-MAP($G'$)
    ☛ After this point, a map of variables
    ☛ to shapes is available in $\varphi$.
    foreach $a_2$ in $assignments(G')$ do
        $\mathtt{q} \leftarrow lhs(a_2)$
        $trackedby(\mathtt{q}) \leftarrow \mathtt{q}$
        foreach $a_1$ in $assignments(G')$ do
            $\mathtt{p} \leftarrow lhs(a_1)$
            if $(a_2 \in DF(a_1)) \wedge \varphi(\mathtt{q}) = \varphi(\mathtt{p})$
                $trackedby(\mathtt{q}) \leftarrow \mathtt{p}$
                break
            if $(a_2 \in SDOM(a_1)) \wedge \varphi(\mathtt{q}) = \varphi(\mathtt{p})$
                $trackedby(\mathtt{q}) \leftarrow \mathtt{p}$
                break
```

Fig. 11. The SHAPE-TRACKING procedure.

```
FORM-SHAPE-MAP($G'$)
    ☛ $G'$ is a CFG in the SSA form.
    $G'' \leftarrow DFS(\mathsf{entry}, G', forward\_substitute)$
    foreach $astmt$ in $assignments(G'')$ do
        $\mathtt{c} \leftarrow lhs(astmt)$
        $e \leftarrow rhs(astmt)$
        $\varphi(\mathtt{c}) \leftarrow$ MULTIPASS-EVAL($\varphi(e)$)
```

Fig. 12. The FORM-SHAPE-MAP procedure.

—forward_substitute($v, G'$) visits definitions in a basic block $v$ in their lexical order, and substitutes every *forward use*, both in $v$ and in the rest of the SSA-conforming CFG $G'$, by the definition's righthand side.

The algorithm consists of three stages: (1) An initial conversion of the input CFG $G$ into a new CFG $G'$ that is in the SSA form; (2) an invocation of FORM-SHAPE-MAP on $G'$ to build the map $\varphi$ of variables to shapes; and (3) the formation of a map *trackedby* from variables to variables such that if *trackedby*(q) is p, then the shape of q tracks the shape of p. The FORM-SHAPE-MAP procedure begins by forward-substituting all assignments in $G'$ to obtain a new CFG $G''$. Forward substitution is performed with the hope of determining the "ultimate" shape of a variable. Informally, this means determining the variable's shape after taking into consideration the shapes of all variables both directly and indirectly used in its defining MATLAB expression, and evaluating the resulting shape expression into some simplified canonical form.[23] Figure 12 does the forward substitution in depth-first order, even though a breadth-first order could also have been used. FORM-SHAPE-MAP then loops through each assignment *astmt* in $G''$, extracting its lefthand side c and righthand side $e$. The loop updates the shape map $\varphi$ at c by determining the shape expression of $e$ (i.e., $\varphi(e)$), evaluating it (i.e., MULTIPASS-EVAL($\varphi(e)$)), and assigning the result to $\varphi$(c).

[23]Clearly, how close FORM-SHAPE-MAP comes to achieving this ideal depends on two factors: (1) The degree to which the shape propagation is carried out, that is, the length of the forward-substitution chain; and (2) the evaluator's ability to detect a more simple underlying canonical shape.

$$\begin{aligned}
&\mathtt{d}_1 &&\leftarrow \max(\mathrm{b}_0{*}\mathrm{b}_0,\ \mathrm{abs}(\mathrm{a}_0){*}\mathrm{b}_0);\\
&\mathtt{a}_1 &&\leftarrow \mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0\text{'});\\
&\mathtt{d}_2 &&\leftarrow (\mathrm{b}_0{+}\max(\mathrm{b}_0{*}\mathrm{b}_0,\ \mathrm{abs}(\mathrm{a}_0){*}\mathrm{b}_0)){*}\mathrm{fix}(\mathrm{b}_0);\\
&\mathtt{a}_2 &&\leftarrow \Phi(P)(\mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0\text{'}),\ \mathtt{a}_3);\\
&\mathtt{b}_1 &&\leftarrow \Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2);\\
&\mathtt{e}_1 &&\leftarrow \Phi(P)(\mathrm{e}_0,\ \mathrm{e}_2);\\
&\mathtt{c} &&\leftarrow \Phi(P)(\mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0\text{'}),\ \mathtt{a}_3){*}\Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2);\\
&\mathtt{a}_3 &&\leftarrow \Phi(P)(\mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0\text{'}),\ \mathtt{a}_3){*}\Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2)\\
& &&\qquad +\Phi(P)(\mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0\text{'}),\ \mathtt{a}_3).\hat{}\,\Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2);\\
&\mathtt{b}_2 &&\leftarrow \Phi(P)(\mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0\text{'}),\ \mathtt{a}_3){*}\Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2)\text{-}(((\Phi(P)(\mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0\text{'}),\ \mathtt{a}_3){*}\Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2))\\
& &&\qquad +(\Phi(P)(\mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0\text{'}),\ \mathtt{a}_3).\hat{}\,\Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2)))\\
& &&\qquad\quad ./(\Phi(P)(\mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0\text{'}),\ \mathtt{a}_3){*}\Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2)));\\
&\mathtt{e}_2 &&\leftarrow \Phi(Q)(\Phi(P)(\mathrm{e}_0,\ \mathrm{e}_2),\ \mathrm{e}_3);\\
&\mathtt{e}_3 &&\leftarrow \Phi(Q)(\Phi(P)(\mathrm{e}_0,\ \mathrm{e}_2),\ \mathrm{e}_3){*}((\mathrm{b}_0{+}\max(\mathrm{b}_0{*}\mathrm{b}_0,\ \mathrm{abs}(\mathrm{a}_0){*}\mathrm{b}_0)){*}\mathrm{fix}(\mathrm{b}_0));\\
&\mathtt{h} &&\leftarrow \Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2).\hat{}\,\Phi(P)(\mathrm{b}_0,\ \mathrm{b}_2)\text{-}\Phi(P)(\mathrm{atan2}(\mathrm{b}_0,\ \mathrm{b}_0'),\ \mathtt{a}_3);
\end{aligned}$$

Fig. 13. After forward substitution on the pruned SSA code in Figure 6.

$$\begin{aligned}
\varphi(\mathtt{d}_1) &= \varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)\dot{*}\varphi(\mathtt{b}_0),\\
\varphi(\mathtt{a}_1) &= \varphi(\mathtt{b}_0)\dot{+}\dot{\neg}\varphi(\mathtt{b}_0),\\
\varphi(\mathtt{d}_2) &= \big(\varphi(\mathtt{b}_0)\dot{+}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)\dot{*}\varphi(\mathtt{b}_0))\big)\dot{*}\varphi(\mathtt{b}_0),\\
\varphi(\mathtt{a}_2) &= \dot{\Phi}(P)(\varphi(\mathtt{b}_0)\dot{+}\dot{\neg}\varphi(\mathtt{b}_0),\varphi(\mathtt{a}_3)),\\
\varphi(\mathtt{b}_1) &= \dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2)),\\
\varphi(\mathtt{e}_1) &= \dot{\Phi}(P)(\varphi(\mathtt{e}_0),\varphi(\mathtt{e}_2)),\\
\varphi(\mathtt{c}) &= \dot{\Phi}(P)(\varphi(\mathtt{b}_0)\dot{+}\dot{\neg}\varphi(\mathtt{b}_0),\varphi(\mathtt{a}_3))\dot{*}\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2)),\\
\varphi(\mathtt{a}_3) &= \dot{\Phi}(P)(\varphi(\mathtt{b}_0)\dot{+}\dot{\neg}\varphi(\mathtt{b}_0),\varphi(\mathtt{a}_3))\dot{*}\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2))\\
&\qquad \dot{+}\big(\dot{\Phi}(P)(\varphi(\mathtt{b}_0)\dot{+}\dot{\neg}\varphi(\mathtt{b}_0),\varphi(\mathtt{a}_3))\dot{+}\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2))\big),\\
\varphi(\mathtt{b}_2) &= \dot{\Phi}(P)(\varphi(\mathtt{b}_0)\dot{+}\dot{\neg}\varphi(\mathtt{b}_0),\varphi(\mathtt{a}_3))\dot{*}\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2))\\
&\qquad \dot{+}\big(\big(\dot{\Phi}(P)(\varphi(\mathtt{b}_0)\dot{+}\dot{\neg}\varphi(\mathtt{b}_0),\varphi(\mathtt{a}_3))\dot{*}\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2))\\
&\qquad\quad \dot{+}\big(\dot{\Phi}(P)(\varphi(\mathtt{b}_0)\dot{+}\dot{\neg}\varphi(\mathtt{b}_0),\varphi(\mathtt{a}_3))\dot{+}\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2))\big)\big)\\
&\qquad\qquad \dot{+}\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\dot{\neg}\varphi(\mathtt{b}_0),\varphi(\mathtt{a}_3))\dot{*}\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2))\big),\\
\varphi(\mathtt{e}_2) &= \dot{\Phi}(Q)(\dot{\Phi}(P)(\varphi(\mathtt{e}_0),\varphi(\mathtt{e}_2)),\varphi(\mathtt{e}_3)),\\
\varphi(\mathtt{e}_3) &= \dot{\Phi}(Q)(\dot{\Phi}(P)(\varphi(\mathtt{e}_0),\varphi(\mathtt{e}_2)),\varphi(\mathtt{e}_3))\dot{*}\big(\big(\varphi(\mathtt{b}_0)\dot{+}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)\dot{*}\varphi(\mathtt{b}_0))\big)\dot{*}\varphi(\mathtt{b}_0)\big),\\
\varphi(\mathtt{h}) &= \big(\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2))\dot{+}\dot{\Phi}(P)(\varphi(\mathtt{b}_0),\varphi(\mathtt{b}_2))\big)\dot{+}\dot{\Phi}(P)(\varphi(\mathtt{b}_0)\dot{+}\dot{\neg}\varphi(\mathtt{b}_0),\varphi(\mathtt{a}_3)).
\end{aligned}$$

Fig. 14. After applying $\varphi$ to the left and righthand sides of the assignments in Figure 13.

### 7.1.3 *Confirming Inferences* 1 *and* 2 *in Section* 1.

To demonstrate the workings of the SHAPE-TRACKING algorithm, we revisit the example in Figure 6. Figure 13 shows the assignments after running a forward-substitution pass, that is, after Line 1 in FORM-SHAPE-MAP. Figure 14 displays the unevaluated shape expressions after applying $\varphi$ on both sides of the assignments in Figure 13, in accordance with Equation (54). Figure 15 shows the results of applying MULTIPASS-EVAL on these unevaluated shape expressions.[24] We observe that the results are mostly more simple than those of the shape expressions in Figure 14, the exceptions being $\varphi(\mathtt{b}_1)$, $\varphi(\mathtt{e}_1)$, and $\varphi(\mathtt{e}_2)$, which remain the same. Moreover, the evaluated shape expressions for two pairs of variables—$\varphi(\mathtt{d}_1)$, $\varphi(\mathtt{d}_2)$, and $\varphi(\mathtt{a}_3)$, $\varphi(\mathtt{b}_2)$—are identical (in boldface in Figure 15). Because

[24]The evaluation was done with an implementation of MULTIPASS-EVAL that used Mathematica's canonical expression ordering [Wolfram 1999, 1031] as the total order $\sqsubseteq$. Employing a similar MATLAB code fragment as the running example, Joisha and Banerjee [2002] discuss how Mathematica can be programmed to perform this evaluation.

$$
\begin{aligned}
&\boldsymbol{\varphi(\mathtt{d}_1) = \varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0) \dotplus \varphi(\mathtt{a}_0)},\\
&\varphi(\mathtt{a}_1) = \varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0),\\
&\boldsymbol{\varphi(\mathtt{d}_2) = \varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0) \dotplus \varphi(\mathtt{a}_0)},\\
&\varphi(\mathtt{a}_2) = \dot{\Phi}(P)(\varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0), \varphi(\mathtt{a}_3)),\\
&\varphi(\mathtt{b}_1) = \dot{\Phi}(P)(\varphi(\mathtt{b}_0), \varphi(\mathtt{b}_2)),\\
&\varphi(\mathtt{e}_1) = \dot{\Phi}(P)(\varphi(\mathtt{e}_0), \varphi(\mathtt{e}_2)),\\
&\varphi(\mathtt{c}) \ \ = \dot{\Phi}(P)(\varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0), \varphi(\mathtt{a}_3)) * \dot{\Phi}(P)(\varphi(\mathtt{b}_0), \varphi(\mathtt{b}_2)),\\
&\boldsymbol{\varphi(\mathtt{a}_3) = \dot{\Phi}(P)(\varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0), \varphi(\mathtt{a}_3)) * \dot{\Phi}(P)(\varphi(\mathtt{b}_0), \varphi(\mathtt{b}_2))}\\
&\qquad\quad \boldsymbol{\dotplus \dot{\Phi}(P)(\varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0), \varphi(\mathtt{a}_3)) \dotplus \dot{\Phi}(P)(\varphi(\mathtt{b}_0), \varphi(\mathtt{b}_2)))},\\
&\boldsymbol{\varphi(\mathtt{b}_2) = \dot{\Phi}(P)(\varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0), \varphi(\mathtt{a}_3)) * \dot{\Phi}(P)(\varphi(\mathtt{b}_0), \varphi(\mathtt{b}_2))}\\
&\qquad\quad \boldsymbol{\dotplus \dot{\Phi}(P)(\varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0), \varphi(\mathtt{a}_3)) \dotplus \dot{\Phi}(P)(\varphi(\mathtt{b}_0), \varphi(\mathtt{b}_2)))},\\
&\varphi(\mathtt{e}_2) = \dot{\Phi}(Q)(\dot{\Phi}(P)(\varphi(\mathtt{e}_0), \varphi(\mathtt{e}_2)), \varphi(\mathtt{e}_3)),\\
&\varphi(\mathtt{e}_3) = \dot{\Phi}(Q)(\dot{\Phi}(P)(\varphi(\mathtt{e}_0), \varphi(\mathtt{e}_2)), \varphi(\mathtt{e}_3)) * (\varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0) \dotplus \varphi(\mathtt{a}_0)),\\
&\varphi(\mathtt{h}) \ \ = \dot{\Phi}(P)(\varphi(\mathtt{b}_0) * \varphi(\mathtt{b}_0), \varphi(\mathtt{a}_3)) \dotplus \dot{\Phi}(P)(\varphi(\mathtt{b}_0), \varphi(\mathtt{b}_2)).
\end{aligned}
$$

Fig. 15. After MULTIPASS-EVAL on the shape expressions in Figure 14.

$\mathtt{d}_1$ and $\mathtt{a}_3$ strictly dominate $\mathtt{d}_2$ and $\mathtt{b}_2$, respectively, Line 12 in SHAPE-TRACKING (Figure 11) sets *trackedby*($\mathtt{d}_2$) to $\mathtt{d}_1$, and *trackedby*($\mathtt{b}_2$) to $\mathtt{a}_3$, thus substantiating Inferences 1 and 2 in Section 1.

7.1.4 *Confirming Inferences* 5 *and* 6 *in Section* 2.1.1. The `quadrature` function in Figure 4 is already in the SSA form. Only `a` and `b` are live at its entry. Thus, by following the same sequence of steps described in Section 7.1.3, we can apply SHAPE-TRACKING on `quadrature`'s CFG to obtain the following shape map:

$$
\begin{aligned}
&\varphi(\mathtt{h}) = \varphi(\mathtt{a}) \dotplus \varphi(\mathtt{b}),\\
&\varphi(\mathtt{mid}) = \varphi(\mathtt{a}) \dotplus \varphi(\mathtt{b}),\\
&\varphi(\mathtt{Fa}) = \varphi(\mathtt{a}) \dotplus \varphi(\mathtt{b}),\\
&\varphi(\mathtt{Fmid}) = \varphi(\mathtt{a}) \dotplus \varphi(\mathtt{b}),\\
&\varphi(\mathtt{Fb}) = \varphi(\mathtt{a}) \dotplus \varphi(\mathtt{b}),\\
&\varphi(\mathtt{S}) = (\varphi(\mathtt{a}) \dotplus \varphi(\mathtt{b})) * (\varphi(\mathtt{a}) \dotplus \varphi(\mathtt{b})).
\end{aligned}
$$

Since the images $\varphi(\mathtt{h})$, $\varphi(\mathtt{mid})$, $\varphi(\mathtt{Fa})$, $\varphi(\mathtt{Fmid})$, and $\varphi(\mathtt{Fb})$ are all equal and because the body of `quadrature` is straight-line code, we get the claim in Inference 5. In particular, if the computation in Line 2 of `quadrature` is well-defined, the computations up to Line 6 will also be well-defined. However, because $s*s$ is $\overline{\pi}$ only when $s$ isn't a square shape, `S` on Line 7 is well-defined only when `a` and `b` are square matrices. Hence the claim in Inference 6.

## 7.2 Preallocation

By resorting to forward substitution, FORM-SHAPE-MAP effectively ignores the propagation of shape information along CFG back edges. We now present a method that accounts for shape information along CFG back edges so as to give a fuller picture of how a variable's shape evolves in the course of program execution.

```
SET-OF-SHAPES-EVAL($\{s_1, s_2, \ldots, s_m\}$)
    ☛ Each $s_j$ $(1 \le j \le m)$ is a shape term in $\Lambda$.
1   $S' \leftarrow \emptyset$
2   for $j \leftarrow 1$ to $m$ do
3       $s'_j \leftarrow$ MULTIPASS-EVAL($s_j$)
4       $S' \leftarrow S' \cup \{s'_j\}$
5   do
6       $S'' \leftarrow S'$
7       $S' \leftarrow \mathcal{A}^\dagger(S', \varrho^\dagger)$
        ☛ $\varrho^\dagger$ is a fixed rewrite tuple of the form
        ☛ $(L_1 \to R_1, L_2 \to R_2, \ldots, L_\gamma \to R_\gamma)$,
        ☛ where each $L_i \to R_i$ is in $2^\Lambda \times 2^\Lambda$.
8   until $S' = S''$
9   return $S'$
```

Fig. 16. The SET-OF-SHAPES-EVAL procedure.

Consider the power set $2^\Lambda$ of $\Lambda$. Our approach to preallocation (i.e., determining the set of all possible shapes assumed by an array expression) is based on the definitions of a lattice $\mathcal{L}_{2^\Lambda}$ on $2^\Lambda$ and a function space $\breve{\mathcal{F}}$ that consists of operators which are monotonic on $\mathcal{L}_{2^\Lambda}$. Members of $\breve{\mathcal{F}}$ are systematically formulated by using a procedure called SET-OF-SHAPES-EVAL. SET-OF-SHAPES-EVAL is an extension of MULTIPASS-EVAL to *sets* of shape terms. To phrase it another way, SET-OF-SHAPES-EVAL($\{s_1, s_2, \ldots, s_m\}$) produces a "simplified" version of $\{s_1, s_2, \ldots, s_m\}$, where each $s_j$ $(1 \le j \le m)$ is a shape term in $\Lambda$. The extension operates in two steps:

(1) It first simplifies each $s_j$ in $\{s_1, s_2, \ldots, s_m\}$ by using MULTIPASS-EVAL to obtain the new set $\{s'_1, s'_2, \ldots, s'_m\}$; and
(2) it then tries to simplify $\{s'_1, s'_2, \ldots, s'_m\}$ in its entirety by using certain *set-of-shapes* identities, that is, identities that exist between entire sets of shape terms.

7.2.1 *The Set-of-Shapes Evaluator.* Like the identities in Section 5.7 that describe properties in the semantic domain $\mathbb{S}_\wp$, properties in $2^{\mathbb{S}_\wp}$ can also be characterized and captured by identities. An example is

$$\{\bar{\boldsymbol{s}}{*}\bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}{*}\bar{\boldsymbol{t}}, \bar{\boldsymbol{s}}{*}\bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}, \bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}{*}\bar{\boldsymbol{t}}\} = \{\bar{\boldsymbol{s}}{*}\bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}, \bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}{*}\bar{\boldsymbol{t}}\}, \tag{58}$$

which follows from the fact that for a given pair of shape tuple class expressions, $\bar{\boldsymbol{s}}$ and $\bar{\boldsymbol{t}}$, in $\mathbb{S}_\wp$, $\bar{\boldsymbol{s}}{*}\bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}{*}\bar{\boldsymbol{t}}$ will always be either $\bar{\boldsymbol{s}}{*}\bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}$ or $\bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}{*}\bar{\boldsymbol{t}}$ [Joisha 2003]. Another set-of-shapes identity, proved in Joisha [2003], is

$$\{\bar{\boldsymbol{s}}{*}\bar{\boldsymbol{t}}{+}\bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}, \bar{\boldsymbol{s}}{*}\bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}, \bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}{*}\bar{\boldsymbol{t}}\} = \{\bar{\boldsymbol{s}}{*}\bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}, \bar{\boldsymbol{s}}{+}\bar{\boldsymbol{t}}{*}\bar{\boldsymbol{t}}\}. \tag{59}$$

Figure 16 shows how the SET-OF-SHAPES-EVAL procedure uses these set-of-shapes identities. The transformer $\mathcal{A}^\dagger$ on Line 7 is defined as

$$\mathcal{A}^\dagger(S, (L_1 \to R_1, L_2 \to R_2, \ldots, L_k \to R_k)) = \begin{cases} (S - \xi(L_1)) \cup \xi(R_1) & \text{if } \exists \xi \in \Xi \text{ such that } \xi(L_1) \subseteq S, \\ S & \text{else if } k = 1, \\ \mathcal{A}^\dagger(S, (L_2 \to R_2, \ldots, L_k \to R_k)) & \text{otherwise.} \end{cases} \tag{60}$$

Each rule $L_i \to R_i$ in Equation (60) is an ordered pair in $2^{\Lambda} \times 2^{\Lambda}$ representing an identity on the $2^{\mathbb{S}_\wp}$ semantic domain. Substitutions are used in Equation (60) after being homomorphically extended to mappings from $2^{\Lambda}$ to $2^{\Lambda}$:

$$\xi(\{s_1, s_2, \ldots, s_m\}) = \{\xi(s_1), \xi(s_2), \ldots, \xi(s_m)\}. \tag{61}$$

Thus, the transformer $\mathcal{A}^{\dagger}$ is similar to the rules applicator in Equation (36), except that subsets are matched in Equation (60) rather than individual shape terms as in Equation (36). It is used with a fixed tuple of rules $\varrho^{\dagger}$ in SET-OF-SHAPES-EVAL.

Since $\mathcal{A}^{\dagger}$ is a many-valued function like the rules applicator in Equation (36), we shall write $T \in \mathcal{A}^{\dagger}(S, \tau)$ if $T$ is an image of $\mathcal{A}^{\dagger}(S, \tau)$. Thus, if

$$\mathcal{B}(\{S_1, S_2, \ldots\}, \tau) = \bigcup_{i \geq 1} \mathcal{A}^{\dagger}(S_i, \tau), \tag{62}$$

where each $S_i$ is in $2^{\Lambda}$, then $\mathcal{B}$ is a single-valued function based on $\mathcal{A}^{\dagger}$.

7.2.1.1 *Strong Normalization.* Consider a binary relation $\rhd$ on $2^{\Lambda}$ defined such that $S \rhd T$ if and only if $T$ is distinct from $S$, but can be obtained from it after one or more applications of $\mathcal{A}^{\dagger}$ with $\varrho^{\dagger}$. That is,

$$S \rhd T \Leftrightarrow (S \neq T) \wedge (\exists j > 0 \text{ such that } T \in \mathcal{B}^j(\{\{S\}, \varrho^{\dagger})), \tag{63}$$

where the meaning of $\mathcal{B}^i(\mathbf{S}, \varrho^{\dagger})$ for any $\mathbf{S} \in 2^{2^{\Lambda}}$ and nonnegative integer $i$ is

$$\mathcal{B}^i(\mathbf{S}, \varrho^{\dagger}) = \begin{cases} \mathcal{B}(\mathcal{B}^{i-1}(\mathbf{S}, \varrho^{\dagger}), \varrho^{\dagger}) & \text{if } i > 0, \\ \mathbf{S} & \text{if } i = 0. \end{cases} \tag{64}$$

If $\rhd$ is well-founded, then SET-OF-SHAPES-EVAL can also be shown to terminate by arguments similar to the proof for Theorem 5. But under what conditions is $\rhd$ well-founded? One such condition on $\varrho^{\dagger}$ is given next in Lemma 7.

LEMMA 7. *If every rule $L \to R$ in $\varrho^{\dagger}$ is such that either $L = R$ or $|L| > |R|$, then $\rhd$ will be well-founded.*[25]

As an example, if the rules that formed $\varrho^{\dagger}$ were

$$\{s{*}s{+}t{*}t, s{*}s{+}t, s{+}t{*}t\} \to \{s{*}s{+}t, s{+}t{*}t\}, \tag{65}$$
$$\{(s{*}t{+}s){+}t, s{*}s{+}t, s{+}t{*}t\} \to \{s{*}s{+}t, s{+}t{*}t\}, \tag{66}$$
$$\{s{*}t{+}(s{+}t), s{*}s{+}t, s{+}t{*}t\} \to \{s{*}s{+}t, s{+}t{*}t\}, \tag{67}$$

which correspond to the identities in Equations (58) and (59),[26] then SET-OF-SHAPES-EVAL would have been strongly normalizing as a consequence of Lemma 7. Note that it would have also been strongly normalizing if $\varrho^{\dagger}$ were simply defined as

$$\varrho^{\dagger} = (\{s\} \to \{s\}). \tag{68}$$

[25]Vertical bars are used to denote the cardinalities of sets. Recollect from Section 5.1 that they also express determinants, for instance, $|\boldsymbol{s}|$. The different usages will be clear from the context.
[26]Two rules are needed for the identity in Equation (59) due to the possible associations of $+$.

7.2.1.2 *Unique Normalization.* Although SET-OF-SHAPES-EVAL($S$) as defined in Figure 16 isn't guaranteed to produce the same result for a given argument $S$ (on account of the many-valued nature of $\mathcal{A}^\dagger$), it can be made uniquely normalizing either by: (1) restricting $\varrho^\dagger$ to certain kinds of tuples so that the rewriting with $\mathcal{A}^\dagger$ becomes a *confluent* process [Dershowitz and Plaisted 2001]; or by (2) using techniques that are independent of the rewrite tuples used, but that give the impression of uniqueness by doing away with the nondeterminism in Equation (60). An example of the latter is imposing a total order on $\Xi$ and always choosing the "least" among candidate substitutions that achieve a subset match. Of course, while such techniques are limited in that they could inadvertenty result in useful solutions being overlooked, they won't impact situations where only one normal form exists for a solution.

For instance, suppose $\varrho^\dagger$ consists of the rules in Equations (65) to (67). Then given $S = \{(u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v\dot{*}v, (u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v, u\dot{*}u\dot{+}v\dot{*}v, u\dot{*}u\dot{+}v, u\dot{+}v\dot{*}v\}$, there are two solutions to $\mathcal{A}^\dagger(S, \varrho^\dagger)$ by Equation (60):

$$\mathcal{A}^\dagger(S, \varrho^\dagger) = \begin{cases} \{(u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v, u\dot{*}u\dot{+}v\dot{*}v, u\dot{*}u\dot{+}v, u\dot{+}v\dot{*}v\}, \\ \{(u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v\dot{*}v, (u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v, u\dot{*}u\dot{+}v, u\dot{+}v\dot{*}v\}. \end{cases}$$

Both solutions use the rewrite rule $\{(s\dot{*}s)\dot{*}(s\dot{*}s)\dot{+}t, s\dot{*}s\dot{+}t, s\dot{+}t\dot{*}t\} \to \{s\dot{*}s\dot{+}t, s\dot{+}t\dot{*}t\}$, but apply it with the substitutions $\{s \hookrightarrow u\dot{*}u, t \hookrightarrow v\}$ and $\{s \hookrightarrow u, t \hookrightarrow v\}$. Hence, SET-OF-SHAPES-EVAL($S$) would ultimately produce either of two solutions:

$$\begin{tikzcd}
 & \{(u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v, u\dot{*}u\dot{+}v\dot{*}v, u\dot{*}u\dot{+}v, u\dot{+}v\dot{*}v\} \arrow[dr] & \\
\text{SET-OF-SHAPES-EVAL}(S) \arrow[ur] \arrow[dr] & & \{(u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v, u\dot{*}u\dot{+}v, u\dot{+}v\dot{*}v\} \\
 & \{(u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v\dot{*}v, (u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v, u\dot{*}u\dot{+}v, u\dot{+}v\dot{*}v\} &
\end{tikzcd}$$

However, if the $\varrho^\dagger$ defined in Equation (68) is used, then SET-OF-SHAPES-EVAL would be trivially uniquely normalizing because $\mathcal{A}^\dagger$ would then be single-valued. Later, in Section 7.2.4, we shall see that this value for $\varrho^\dagger$ suffices to arrive at Inference 3.

7.2.2 *A Set-of-Shapes Algebra.* If the SET-OF-SHAPES-EVAL procedure is uniquely normalizing, then for any $n$-ary shape operator $\dot{f}_n$ of a Type I MATLAB function $f_n$, we can define a *set-of-shapes* operator $\check{f}_n : (2^\Lambda)^n \mapsto 2^\Lambda$ as follows:

$$\check{f}_n(S_1, S_2, \ldots, S_n) =$$
$$\begin{cases} \text{SET-OF-SHAPES-EVAL}\big(\bigcup_{i=1}^{n} S_i\big) & \text{if } \dot{f}_n = \dot{\Phi}(P), \\ \text{SET-OF-SHAPES-EVAL}(\{t \mid \exists s_i \in S_i \text{ for all} & \text{if } \dot{f}_n \neq \dot{\Phi}(P). \\ 1 \le i \le n \text{ such that } t = \dot{f}_n(s_1, s_2, \ldots, s_n)\}) & \end{cases} \tag{69}$$

Let $\check{\mathcal{F}}$ be the set of all $\check{f}_n$ operators defined by Equation (69) for various $n$. Then, $\check{\mathcal{F}}$ forms a *set-of-shapes* algebraic structure $[2^\Lambda, \check{\mathcal{F}}]$ on the power set of shape terms. A set-of-shapes algebra is useful because it provides us with a way to determine the set of all possible shapes of a MATLAB expression. Therefore, $[2^{\Lambda}, \breve{\mathcal{F}}]$ is the "broadest" of the shape-related algebras because it operates on *sets* of shape terms rather than individual shape terms, as is the case for $[\Lambda, \dot{\mathcal{F}}]$, discussed in Section 6.

A few set-of-shapes operators are $\breve{+}$, $\breve{*}$, and $\breve{\Phi}(P)$, which respectively correspond to $\dot{+}$, $\dot{*}$, and $\dot{\Phi}(P)$. Examples of their workings when $\varrho^{\dagger} = (\{s\} \rightarrow \{s\})$ follow:

$$\{\overline{\langle 4, 2, 2\rangle}, \overline{\langle 5, 3\rangle}\}\breve{+}\{\overline{\langle 4, 2, 2\rangle}, \overline{\langle 10, 11\rangle}\} = \{\overline{\langle 4, 2, 2\rangle}, \overline{\pi}\},$$
$$\{\overline{\langle 1, 2\rangle}, \overline{\langle 5, 3\rangle}\}\breve{*}\{\overline{\langle 3, 2\rangle}, \overline{\langle 2, 4\rangle}\} = \{\overline{\pi}, \overline{\langle 1, 4\rangle}, \overline{\langle 5, 2\rangle}\},$$
$$\breve{\Phi}(P)(\{s\}, \{s, t\}) = \{s, t\}.$$

7.2.2.1 *Mitigating Exponential Blowup.* From Equation (69), the number of shape terms in $\breve{f}_n(S_1, S_2, \ldots, S_n)$ can, at most, be $\prod_{i=1}^{n} |S_i|$. However, this exponential upper bound may not be reached for two reasons: (1) Calls to MULTIPASS-EVAL in Line 3 of SET-OF-SHAPES-EVAL may produce the same outcome on distinct shape terms; and (2) the set-of-shapes identities exercised in Line 7 of the same may simplify the resulting sets in their entirety. An example of a situation where a set-of-shapes identity causes a reduction is

$$\{s, s\dot{*}s\}\breve{+}\{t, t\dot{*}t\} = \{s\dot{+}t, s\dot{+}t\dot{*}t, s\dot{*}s\dot{+}t\}.$$

This will happen when the $\varrho^{\dagger}$ used contains a rule for the identity in Equation (58).

7.2.2.2 *Monotone Set-of-Shapes Operators.* On the set $2^{\Lambda}$, a lattice $\mathcal{L}_{2^{\Lambda}}$ can be defined in which the subset relation $\subseteq$ is the partial order, the empty set $\emptyset$ and $\Lambda$ are the least and greatest elements, and set union $\cup$ and set intersection $\cap$ are the join and meet operations. From Equation (69), it is easy to see that the set-of-shapes operator $\breve{f}_n$ is monotonic on $\mathcal{L}_{2^{\Lambda}}$ *if* SET-OF-SHAPES-EVAL is monotonic on $\subseteq$.

Let $X$ and $Y$ be the values of $S'$ after the **for** loop in Lines 2 to 4 of Figure 16 when SET-OF-SHAPES-EVAL is invoked on $S$ and $T$, respectively. Then clearly, $X \subseteq Y$ if $S \subseteq T$. Thus, it is $\varrho^{\dagger}$ that really determines the monotonicity of SET-OF-SHAPES-EVAL. An obvious choice that ensures this is given in Equation (68) because $U = \mathcal{A}^{\dagger}(U, (\{s\} \rightarrow \{s\}))$ for all $U \in 2^{\Lambda}$. This will be the $\varrho^{\dagger}$ for the rest of this article.

Since the function space $\breve{\mathcal{F}}$ on $\mathcal{L}_{2^{\Lambda}}$ will then comprise monotone members, a fixed-point solution will always exist for a system of equations that is formed on the $2^{\Lambda}$ domain using the set-of-shapes operators in $\breve{\mathcal{F}}$. While monotonicity guarantees convergence to a solution, lattice-theoretic algorithms that seek such a solution aren't assured to terminate because $\mathcal{L}_{2^{\Lambda}}$ has an infinite height; an indefinite number of iterations may therefore be necessary before quiescence is reached.

7.2.3 *An Iterative Forward Dataflow Algorithm.* Consider a function $\varsigma$ such that $\varsigma(\mathtt{c})$ is an element in $2^{\Lambda}$ which denotes the set of all possible shapes taken on by a MATLAB variable c across an entire program. Like $\varphi$, $\varsigma$ can be

```
FORM-SET-OF-SHAPES-MAP($G'$)
    ☛ $G'$ is a CFG in the SSA form.
1   foreach $v$ in nodes($G'$)
2       foreach c in used_vars($v$)
3           $\varsigma(c) \leftarrow \{\varphi(c)\}$
4   foreach $v$ in nodes($G'$)
5       foreach c in defined_vars($v$)
6           $\varsigma(c) \leftarrow \emptyset$
    ☛ At this point, $\varsigma$ is initialized so that if a
    ☛ variable c is only used in the CFG but
    ☛ defined elsewhere, then $\varsigma(c)$ is set to
    ☛ $\{\varphi(c)\}$. If c is however defined in the
    ☛ CFG, then $\varsigma(c)$ is set to the empty set $\emptyset$.
7   iters $\leftarrow 0$
8   do
9       fixedpoint $\leftarrow$ true
10      foreach astmt in assignments($G'$) do
11          c $\leftarrow$ lhs(astmt)
12          $e \leftarrow$ rhs(astmt)
13          if $\varsigma(c) \neq \varsigma(e)$
14              fixedpoint $\leftarrow$ false
15          $\varsigma(c) \leftarrow \varsigma(e)$
16      iters $\leftarrow$ iters $+ 1$
17  while (fixedpoint $=$ false $\wedge$ iters $\leq$ maxiters)
    ☛ maxiters caps the number of iterations
    ☛ because $\mathfrak{L}_{2^\Lambda}$ is an infinite height lattice.
18  if fixedpoint $=$ false
        ☛ Fixed-point not attained; force solution
        ☛ to the topmost element of the lattice.
19      foreach $v$ in nodes($G'$)
20          foreach c in defined_vars($v$)
21              $\varsigma(c) \leftarrow \Lambda$
```

Fig. 17. The FORM-SET-OF-SHAPES-MAP procedure.

homomorphically extended to a function from $\mathbb{M}$ to $2^\Lambda$ so that

$$\varsigma(f_n(e_1, e_2, \ldots, e_n)) = \check{f}_n(\varsigma(e_1), \varsigma(e_2), \ldots, \varsigma(e_n)), \tag{70}$$

where $\check{f}_n$ is the $n$-ary set-of-shapes operator in $\check{\mathcal{F}}$ corresponding to $f_n$. For example, $\varsigma(\mathsf{a} * \mathsf{b}. * \mathsf{c})$ is $\varsigma(\mathsf{a})\check{*}\varsigma(\mathsf{b})\check{+}\varsigma(\mathsf{c})$, and $\varsigma(\Phi(P)(e_1, e_2))$ is $\check{\Phi}(P)(\varsigma(e_1), \varsigma(e_2))$.[27]

The FORM-SET-OF-SHAPES-MAP algorithm, shown in Figure 17, takes a CFG $G'$ in SSA form and builds the $\varsigma$ map for the MATLAB variables in it. It effectively constructs a set-of-shapes equation $\varsigma(\mathsf{c}) = \varsigma(e)$ for every assignment $\mathsf{c} \leftarrow e$ and iteratively solves the system up to *maxiters* times, until a stationary solution is achieved. It initially sets $\varsigma(\mathsf{c})$ of every variable c defined within $G'$ to $\emptyset$ using the *used_vars* and *defined_vars* primitives; *used_vars* and *defined_vars*, respectively, return the variables used and defined in a CFG node. If, however, c is live on entry, then $\varsigma(\mathsf{c})$ is set to $\{\varphi(\mathsf{c})\}$. The solution process then iteratively forward-propagates the set-of-shapes information, capping the number of iterations at

[27]Our precedence convention for the set-of-shapes operators reflects our precedence convention for the shape operators that underlie them. Hence $\check{*}$, which corresponds to $*$, has a higher precedence than $\check{+}$, which corresponds to $+$.

$$
\begin{aligned}
\varsigma(\mathtt{d}_1) &= \{\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)\},\\
\varsigma(\mathtt{a}_1) &= \{\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\},\\
\varsigma(\mathtt{d}_2) &= \{\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)\},\\
\varsigma(\mathtt{a}_2) &= \{\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\},\\
\varsigma(\mathtt{b}_1) &= \{\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0), \varphi(\mathtt{b}_0)\},\\
\varsigma(\mathtt{e}_1) &= \{\Big(\big(\varphi(\mathtt{e}_0)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\big)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\Big)\\
&\qquad \dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)),\\
&\qquad\quad \big(\varphi(\mathtt{e}_0)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\big)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)),\\
&\qquad\quad \varphi(\mathtt{e}_0)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)),\\
&\qquad\quad \varphi(\mathtt{e}_0)\},\\
\varsigma(\mathtt{c}) &= \{(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0))\dot{*}\varphi(\mathtt{b}_0), \varphi(\mathtt{b}_0))\dot{*}\varphi(\mathtt{b}_0)\},\\
\varsigma(\mathtt{a}_3) &= \{\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\},\\
\varsigma(\mathtt{b}_2) &= \{\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\},\\
\varsigma(\mathtt{e}_2) &= \{\Big(\big(\varphi(\mathtt{e}_0)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\big)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\Big)\\
&\qquad \dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)),\\
&\qquad\quad \big(\varphi(\mathtt{e}_0)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\big)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)),\\
&\qquad\quad \varphi(\mathtt{e}_0)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)),\\
&\qquad\quad \varphi(\mathtt{e}_0)\},\\
\varsigma(\mathtt{e}_3) &= \{\Big(\big(\varphi(\mathtt{e}_0)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\big)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\Big)\\
&\qquad \dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)),\\
&\qquad\quad \big(\varphi(\mathtt{e}_0)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\big)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0)),\\
&\qquad\quad \varphi(\mathtt{e}_0)\dot{*}(\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\dot{+}\varphi(\mathtt{a}_0))\},\\
\varsigma(\mathtt{h}) &= \{\varphi(\mathtt{b}_0)\dot{*}\varphi(\mathtt{b}_0)\}.
\end{aligned}
$$

Fig. 18. Set-of-Shapes map for the pruned SSA code in Figure 6.

the predefined constant *maxiters* to overcome the problem of the lattice's infinite height.

7.2.4 *Confirming Inference* 3 *in Section* 1. The pruned SSA form for the code in Figure 1 was shown in Figure 6. Since $\mathtt{a}_0$, $\mathtt{b}_0$, and $\mathtt{e}_0$ are live on entry, FORM-SET-OF-SHAPES-MAP initializes their images under $\varsigma$ to $\{\varphi(\mathtt{a}_0)\}$, $\{\varphi(\mathtt{b}_0)\}$, and $\{\varphi(\mathtt{e}_0)\}$, respectively. For every other variable, the image under $\varsigma$ is initially set to $\emptyset$. At the conclusion of FORM-SET-OF-SHAPES-MAP, executed with *maxiters* equal to $\infty$ and $\varrho^\dagger$ set according to Equation (68), the set-of-shapes map shown in Figure 18 is obtained as the fixedpoint.[28] From this resulting map, we see that there can only be, at most, two shapes for $\mathtt{c}$, one for $\mathtt{a}_3$ and $\mathtt{b}_2$, and three for $\mathtt{e}_3$ during the entire execution lifetime of the loops. This corroborates Inference 3 in Section 1.

## 7.3 On the Issue of Approximations

Unlike the preallocation analysis, the shape-tracking analysis isn't based on a lattice, so there's no approximation or loss of precision in a lattice-theoretic sense. If the analysis can, at best, determine the shape of a MATLAB expression $e$ to be a symbolic shape term, then there's no loss of precision because that symbolic expression will evaluate to the exact shape of $e$ when the shape term's symbols are substituted at runtime. A translator based on the framework will

[28]See Joisha and Banerjee [2002] for a discussion on how Mathematica can be used for this. Joisha and Banerjee [2002] use a nonconfluent $\varrho^\dagger$ for rewriting the set-of-shapes terms.

simply emit code against a symbolic shape term so as to resolve it at runtime. This resolution code may use a data structure such as a vector of integers to access shape tuples.

Approximations (but no inaccuracies) may happen in the preallocation analysis due to its lattice-theoretic formulation. Two kinds of approximations can occur: (1) when the cap *maxiters* on the number of iterations in FORM-SET-OF-SHAPES-MAP is reached, and (2) when a set consisting of one or more shape terms is returned as the fixed-point solution. In the former case, the symbol $\Lambda$ is returned. $\Lambda$ signifies the set of all possible shape terms, so all this means is that in the course of program execution, $e$ may assume any shape, both legal and illegal. In the latter case, each shape term in the set-of-shapes expression indicates a shape that $e$ *might* take on during program execution. In both cases, a translator based on the framework will still have to rely on runtime resolution to allocate storage for $e$. However, the emitted code in the second case will be able to allocate storage immediately after the component shape terms in the set-of-shapes expression have been resolved. This resolution may occur much sooner than the definition of $e$, which is unlike the first case, where it may have to wait until $e$ is about to be defined. The difference is that the preallocation code in the second case may manifest outside a loop, thus avoiding repeated allocations, expansions, contractions, or deallocations. Ultimately, there is a tension between how far ahead of time some information is to be inferred (and the difficulty of obtaining such an inference) versus the specificity of the inference and its potential for inference-based optimizations.

## 8. THE SHAPE INFERENCE FRAMEWORK

From a shape inference perspective, it suffices to focus on only those operators that are part of the MATLAB language definition. These built-in functions are similar to the primitives in APL, and ultimately comprise all MATLAB programs. Once we know how to infer shapes for each of these functions, the shapes of arbitrary MATLAB expressions can be determined by applying program-wide techniques.

### 8.1 Taxonomy

Built-in functions that return a value can be classified into one of three categories on the basis of how the shape of the outputs depend on the shapes of the inputs:

—*Type I*. These built-ins produce values whose shapes are completely known once the shapes of the arguments, if any, are known. Examples are the matrix-multiplication operator, and elementwise operators such as array addition.

—*Type II*. These are built-ins that don't belong to the Type I category, and that produce values whose shapes are completely known only when the elemental values of one or more of their arguments is known. An example is the colon operator [The MathWorks, Inc. 1997]. For instance, in the assignment `c` $\leftarrow$ `a:b`, the result `c` will be a row vector consisting of $\lfloor b' - a' \rfloor + 1$ elements when $a' \leq b'$, where $a'$ and $b'$ represent the runtime *real values* of `a` and `b`,

Table V. Shape-Based Classification of MATLAB's Built-In Functions

| Type I | Type II | Type III |
|---|---|---|
| a*b | a:b | dbstack |
| a+b | a($e$) | eval(a) |
| a-b | permute(a, $e$) | evalin(a, b) |
| a.*b | cat($e$, a, b) | feval(a, b) |
| a.^b | a($e_1, e_2, \ldots, e_n$) | |
| a./b | rand($e_1, e_2, \ldots, e_n$) | |
| a.\b | randn($e_1, e_2, \ldots, e_n$) | |
| a==b | zeros($e_1, e_2, \ldots, e_n$) | |
| a~=b | ones($e_1, e_2, \ldots, e_n$) | |
| a<b | eye($e_1, e_2, \ldots, e_n$) | |
| a&b | c($e$) $\leftarrow$ a | |
| a\|b | c($e_1, e_2, \ldots, e_n$) $\leftarrow$ a | |
| a/b | | |
| a\b | | |
| [a, b] | | |
| [a; b] | | |
| +a | | |
| -a | | |
| ~a | | |
| c(:) $\leftarrow$ a | | |
| a^b | | |
| a(:) | | |
| a' | | |
| a.' | | |
| rand | | |
| length(a) | | |

respectively. When $a' > b'$, `c` is the empty row vector. Thus, the shape of `c` can be determined only when the elemental values of `a` and `b` are known.

—*Type III*. These are built-ins that are neither Type I nor Type II. For them, even full knowledge of the arguments doesn't suffice to determine the shapes of the results. For example, there exists a built-in called `dbstack` that returns the stack trace as a column vector [The MathWorks, Inc. 1997]. A complete execution history may be necessary to determine the shape of the result.

Table V shows some of MATLAB's built-ins grouped by the aforementioned classification system. Members of the Type I class appear to be the majority in the language; in fact, so far we have been able to uncover nine quotient algebras to which the shape semantics of over 50 Type I built-ins are isomorphic [Joisha et al. 2000]. Notice that certain built-ins, like `rand`, can be considered as being either Type I or Type II, depending on which overloaded version is invoked. For instance, when invoked without arguments, `rand` always returns a scalar-shaped result and therefore behaves as a Type I built-in. When invoked with arguments, say `2`, `3`, and `4`, `rand` produces an array of size $2 \times 3 \times 4$ and thus behaves as a Type II operator. The `eval`, `evalin`, and `feval` built-ins shown in the third column execute arbitrary strings. However, merely knowing what these argument strings are won't always be sufficient for determining the shape of the result. In the most general setting, a complete execution history may be necessary to determine the outcome's shape.

Note that this is a proactive taxonomy; if a new built-in were to be introduced into the MATLAB language, it would have to fall into one of the preceding categories.

## 8.2 Extensions to Type II Built-In Functions

Though the framework described in this article directly addresses only the Type I function group, it can be extended to handle Type II operators by using the same technique of function currying used in Section 6.6. By this method, a shape operator of a Type II built-in is itself treated as a function of the nonshape arguments.

Consider $\Delta = \bigcup_{k \geq 0} \mathbb{C}^k$, the set of all tuples composed of complex numbers. If $g$ is an $n$-ary Type II operation in MATLAB, then by definition of the Type II category, there exists a mapping $\grave{g} : (\Delta \times \mathbb{S}_{\wp})^n \mapsto \mathbb{S}_{\wp}$ that describes $g$'s shape semantics. By currying $\grave{g}$ on $\Delta^n$, we can obtain a shape operator with Type I characteristics:

$$\grave{g}(\varepsilon_1, \overline{\boldsymbol{s_1}}, \varepsilon_2, \overline{\boldsymbol{s_2}}, \ldots, \varepsilon_n, \overline{\boldsymbol{s_n}}) = \vec{g}(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n)(\overline{\boldsymbol{s_1}}, \overline{\boldsymbol{s_2}}, \ldots, \overline{\boldsymbol{s_n}}). \tag{71}$$

The codomain of $\vec{g}$ in the previous equation is the set $\dot{\mathcal{F}}$ of shape operators for Type I built-ins.

For instance, consider the righthand side array indexing operation in

$$\texttt{c} \leftarrow \texttt{a}(i_1,\ i_2,\ \ldots\ ,\ i_m).$$

A subscript $i_k$ ($1 \leq k \leq m$) can be any array of integers, as long as each integer in $i_k$ lies within the bounds of a in the $k$th dimension, that is, 1 and the extent of a in the $k$th dimension. Hence, the righthand side array indexing operation can't be a Type I built-in. However, it qualifies as a Type II built-in.

The indexing works conceptually in two steps: (1) Each subscript array is viewed as a set of integers and a Cartesian product is formed across all the subscripts; and (2) the array a is indexed using each tuple in the Cartesian product. The shape of the indexing result is the "shape" of the Cartesian product. Phrased differently, if $j_k$ is the number of elements in the subscript $i_k$, then the shape of the result (assuming there are no out-of-bounds errors) is $j_1 \times j_2 \times \cdots \times j_k$ [The MathWorks, Inc. 1997]. Note that the shape of the output is only affected by the number of elements in a subscript; the subscript's actual shape doesn't affect the result's shape.

As an example, suppose a, x, and y are the following arrays:

$$\texttt{a} = \begin{pmatrix} -99 & 21 & -0.99 \\ 10 & 15 & 101 \end{pmatrix} \qquad \texttt{x} = (1\ \ 2), \qquad \texttt{y} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}.$$

Then, after the assignment c $\leftarrow$ a(x, y, 1), we will have

$$\texttt{c} = \begin{pmatrix} 21 & -0.99 \\ 15 & 101 \end{pmatrix}.$$

In functional form, the array indexing operation could be written as c $\leftarrow \hbar_m$(a, $i_1$, $i_2, \ldots, i_m$), where $\hbar_m$ represents the righthand side array indexing operation with $m$ subscripts. If $\chi$ is a function that associates a MATLAB array with a tuple in $\Delta$ by linearizing the array in some fixed order, then the Type I shape operator $\dot{\hbar}_m$ corresponding to $\hbar_m$ will be

$$\dot{\hbar}_m = \vec{\hbar}_m(\chi(\texttt{a}), \chi(i_1), \chi(i_2), \ldots, \chi(i_m)). \tag{72}$$

If the fixed order is column major, then for the previous example,

$$\begin{aligned}
\chi(\texttt{a}) &= (-99, 10, 21, 15, -0.99, 101),\\
\chi(\texttt{x}) &= (1, 2),\\
\chi(\texttt{y}) &= (2, 3),\\
\chi(\texttt{c}) &= (21, 15, -0.99, 101).
\end{aligned}$$

From Equations (71) and (72), the shape of `a(x, y, 1)` will then be

$$\vec{\hbar}_3((-99, 10, 21, 15, -0.99, 101), (1, 2), (2, 3), (1))(\overline{\langle 2, 3\rangle}, \overline{\langle 1, 2\rangle}, \overline{\langle 2, 1\rangle}, \overline{\iota}) = \overline{\langle 2, 2\rangle}.$$

Of course, an illegal shape will be returned if an integer in an array's subscript exceeds the bounds for that dimension. For instance,

$$\vec{\hbar}_2((-1, 7, 2.3, 0, 11), (2), (2))(\overline{\langle 1, 5\rangle}, \overline{\iota}, \overline{\iota}) = \overline{\pi}.$$

8.2.1 *Value Ranges with Curried Shape Operators.* Since keeping track of the entire contents of an array by means of a tuple in $\Delta$ may be prohibitive, an approximation may be to track only its value range. Consider the shape

$$\vec{\hbar}_m(\chi(t_0), \chi(t_1), \chi(t_2), \ldots, \chi(t_m))(s_0, s_1, s_2, \ldots, s_m),$$

where each $s_i$ is a shape term, and each $t_i$ represents a MATLAB expression. Thus, $\chi(t_i)$ is a placeholder for an element in $\Delta$. If we had no more information about any $\chi(t_i)$, the aforementioned may be the most simplified form of the shape.[29] However, if we were to know the value ranges of the $t_i$, then it might be possible to simplify the shape. For instance, let the value ranges of $\chi(t_0)$, $\chi(t_1)$, and $\chi(t_2)$ be $[\![-\infty, \infty]\!]$, $[\![1.1, 4.78]\!]$, and $[\![2, 4]\!]$, respectively. Then, we would have the following:

$$\begin{aligned}
&\vec{\hbar}_2(\chi(t_0), \chi(t_1), \chi(t_2))(\overline{\langle 10, 5\rangle}, \overline{\langle 2, 2\rangle}, \overline{\langle 3, 5\rangle}) = \overline{\langle 4, 15\rangle},\\
&\vec{\hbar}_2(\chi(t_0), \chi(t_1), \chi(t_2))(\overline{\langle 1, 5\rangle}, \overline{\langle 2, 2\rangle}, \overline{\langle 3, 5\rangle}) = \overline{\pi},\\
&\vec{\hbar}_2(\chi(t_0), \chi(t_1), \chi(t_2))(\overline{\langle 5, 7\rangle}, \overline{\iota}, \overline{\iota}) = \overline{\iota},\\
&\vec{\hbar}_2(\chi(t_0), \chi(t_1), \chi(t_2))(\overline{\langle 3, 7\rangle}, \overline{\iota}, \overline{\iota}) = \vec{\hbar}_2(\chi(t_0), \chi(t_1), \chi(t_2))(\overline{\langle 3, 7\rangle}, \overline{\iota}, \overline{\iota}).
\end{aligned}$$

In the last example, the shape remains unsimplified, since $[\![1.1, 4.78]\!]$ straddles 4.[30]

8.2.2 *Type I Compositions of Type II Built-In Functions.* The composition of certain Type II built-ins with Type I built-ins exhibits overall Type I characteristics. For instance, an idiom seen in many MATLAB programs is

$$\texttt{b} \leftarrow F(\texttt{size(a)}),$$

[29]To ensure that shape errors propagate through the rest of the program from the point of origination, the preceding shape will always simplify to $\overline{\pi}$ if any of the $s_i$ are $\overline{\pi}$.

[30]Whether the indices are actually integers would be established by an intrinsic type analysis.

where $F$ could be zeros, ones, or rand, among others. This construction is used to create an array with the same shape as an existing array, but initialized to some other value (see Figure 3). The size built-in is Type I because knowing the shape of a is enough to establish the shape of size(a). As an example, if the shape of a is $\langle 2, 4, 5\rangle$, then the shape of size(a) is $\langle 1, 3\rangle$. On the other hand, zeros is a Type II built-in because the actual values of the arguments need to be known to determine the shape of the result. For instance, if x is a row vector with the elements 5, 3, and 7, zeros(x) would create an array having the shape $\langle 5, 3, 7\rangle$. Thus, the composition zeros(size(a)) will always return an array with the same shape as a; it is therefore Type I in character.

8.2.3 *Confirming Inference* 4 *in Section* 1. The invocation length(x) always returns a scalar for any MATLAB array x. If x is the empty array, the value returned is 0; otherwise, it equals max(size(x)) [The MathWorks, Inc. 1997]. Hence, the array rank in the hilarray_acc user-defined function of Figure 3 is always a scalar. From the discussion in Section 8.2.2, both indices and sindices will always have the same shape as a. Now for any MATLAB array x, the call cumsum(x, *e*) returns an array of the same shape as x, as long as *e* is a positive scalar integer [The MathWorks, Inc. 1997]. However, if the loop in Figure 3 gets executed, k is guaranteed to be a positive scalar integer. Thus, the outcome of cumsum(indices, k) will have the same shape as indices, and transitively, the same shape as a. Since we know that sindices initially also has the shape of a, the shape of sindices remains unchanged in every iteration of the accumulation loop. Because rank is a scalar, we can thus conclude that the b computed in the last line of hilarray_acc will always have the same shape as a.

## 8.3 Limitations: Treatment of Type III Built-In Functions

Type III operators appear to be few in MATLAB and although the framework presented in this article lacks in its ability to expressly model their shape semantics, their presence doesn't thwart its applicability or usefulness. The framework can always resort to viewing the shape of an expression involving such operators in an opaque manner by using a symbolic variable to represent the expression's shape. Thus, while an implementation based on the framework may have to bank on runtime resolution to calculate the shape outcomes of Type III built-ins, it may still benefit from the framework because of the way these outcomes are subsequently used.

To demonstrate, consider code that has calls to the Type III built-in evalin:

```
a ← evalin('ws', s1);
b ← evalin('ws', s2);
c ← (a*b+a+b)*(a*b+a+b);
```

evalin allows the execution of a string containing any valid MATLAB expression in the context of a specified workspace of variables. In the given code fragment, two strings are executed in the workspace ws and their results are assigned to a and b. The framework would effectively view the code in the following way:

$$\begin{array}{l}
\blacktriangleright\ \varphi(\texttt{a}) \leftarrow \varphi(\texttt{evalin('ws', s1)}) \\
\texttt{a} \leftarrow \texttt{evalin('ws', s1);} \\
\blacktriangleright\ \varphi(\texttt{b}) \leftarrow \varphi(\texttt{evalin('ws', s2)}) \\
\texttt{b} \leftarrow \texttt{evalin('ws', s2);} \\
\blacktriangleright\ \varphi(\texttt{c}) \leftarrow (\varphi(\texttt{a}) \dot{*} \varphi(\texttt{b}) \dot{+} \varphi(\texttt{a}) \dot{+} \varphi(\texttt{b})) \dot{*} (\varphi(\texttt{a}) \dot{*} \varphi(\texttt{b}) \dot{+} \varphi(\texttt{a}) \dot{+} \varphi(\texttt{b})) \\
\texttt{c} \leftarrow \texttt{(a*b+a+b)*(a*b+a+b),}
\end{array}$$

where $\varphi(\texttt{a})$, $\varphi(\texttt{b})$, and $\varphi(\texttt{c})$ denote the shapes of `a`, `b`, and `c`, respectively, and $\varphi(\texttt{evalin('ws', s1)})$, $\varphi(\texttt{evalin('ws', s2)})$ stand for the shapes of `evalin`'s outcomes. At compile-time, the framework would then be able to simplify the computation of $\varphi(\texttt{c})$, using one of the heterogeneous identities in Table II, to:

$$\begin{array}{l}
\blacktriangleright\ \varphi(\texttt{a}) \leftarrow \varphi(\texttt{evalin('ws', s1)}) \\
\texttt{a} \leftarrow \texttt{evalin('ws', s1);} \\
\blacktriangleright\ \varphi(\texttt{b}) \leftarrow \varphi(\texttt{evalin('ws', s2)}) \\
\texttt{b} \leftarrow \texttt{evalin('ws', s2);} \\
\blacktriangleright\ \varphi(\texttt{c}) \leftarrow \varphi(\texttt{a}) \dot{*} \varphi(\texttt{a}) \dot{+} \varphi(\texttt{b}) \dot{*} \varphi(\texttt{b}) \\
\texttt{c} \leftarrow \texttt{(a*b+a+b)*(a*b+a+b).}
\end{array}$$

Thus, while $\varphi(\texttt{c})$ is still ultimately resolved at runtime using the values of $\varphi(\texttt{a})$ and $\varphi(\texttt{b})$, the framework diminishes the overhead of its runtime computation.

## 9. MEASUREMENTS

This section reports measurements from MAGICA [Joisha and Banerjee 2003b], an implementation that realizes the shape inference techniques described in this article. Numbers pertaining to inferred shape information were collected over 17 programs obtained from a variety of sources, including the test suites of recent research compilers for MATLAB. These programs are organized as input files called *M-files*, and are hence directly executable by a MATLAB interpreter. Each benchmark consists of a separate driver routine from which is invoked an entry point to the benchmark. This organization was taken from the FALCON research compiler [De Rose and Padua 1999]. Table VI lists these benchmarks with brief descriptions, their sources and sizes in terms of the number of program files, and the total number of lines of program text.

The current version of MAGICA supports built-in functions, such as `disp` and `fprintf`, that enable writing output to an external file. However, no support presently exists for reading data from an external file. The original versions of some of the benchmarks did read in external data, though; these were modified to include the loaded data in the driver routine. This wasn't found to be a problem because the externally loaded data was only a couple of scalars.

Our benchmark suite includes two programs that manipulate three-dimensional arrays. This is unlike the test suites of previous research compilers for MATLAB because existing MATLAB programs mainly confine themselves to manipulating scalars, vectors, and matrices—a state of affairs due to early versions of MATLAB supporting only matrices. With the introduction of

Table VI. Benchmark Suite Description

| Benchmark | Synopsis | Origin | M-Files | Lines |
|---|---|---|---|---|
| `adpt` | Adaptive Quadrature by Simpson's Rule | † | 2 | 79 |
| `capr` | Transmission Line Capacitance | ❋ | 5 | 68 |
| `clos` | Transitive Closure | ✞ | 2 | 30 |
| `crni` | Crank-Nicholson Heat Equation Solver | † | 3 | 48 |
| `diff` | Young's Two-Slit Diffraction Experiment | ✶ | 2 | 40 |
| `dich` | Dirichlet Solution to Laplace's Equation | † | 2 | 49 |
| `edit` | Edit Distance | ✶ | 2 | 34 |
| ❑ `fdtd` | Finite Difference Time Domain (FDTD) Technique | ❋ | 2 | 47 |
| `fiff` | Finite Difference Solution to the Wave Equation | † | 2 | 32 |
| `nb1d` | One-Dimensional $N$-Body Simulation | ✞ | 2 | 53 |
| ❑ `nb3d` | Three-Dimensional $N$-Body Simulation | Modified `nb1d` | 2 | 46 |
| `bari` | Elastic Bar Displacement | ✶ | 2 | 21 |
| `baye` | Probabilities by Bayes' Rule | ✣ | 3 | 29 |
| `brt1` | Mandelbrot | ✾ | 2 | 23 |
| `brt2` | Mandelbrot Optimized | Modified `brt1` | 2 | 30 |
| `sunl` | Sunlight Illumination in Lux | ✡ | 2 | 60 |
| `vfin` | Vector Comparison with Loop | ✣ | 3 | 49 |

**Legend**

❑ Benchmarks involve three-dimensional arrays.
† FALCON MATLAB Compiler Test Suite [De Rose and Padua 1999].
✞ OTTER Parallel MATLAB Compiler Test Suite [Malishevsky 1998].
❋ Chalmers University of Technology, Sweden (`www.elmagn.chalmers.se/courses/CEM/`).
✶ The MATLAB Central File Exchange (`www.mathworks.com/matlabcentral/fileexchange`).
✣ "Accelerating MATLAB: The MATLAB JIT-Accelerator" (`www.mathworks.com/mld_accel`).
✾ "Picking up the Pace with the MATLAB Profiler", *MATLAB News & Notes*, May 2003.
✡ Tel Aviv University, Israel (`http://wise-obs.tau.ac.il/~eran/matlab.html`).

multidimensional array support in version series 5 (the current series is 7), this is likely to change.

## 9.1 Platform Specifications

The measurements were taken on a Dell Latitude C840, equipped with a 1.60 GHz Intel Pentium 4 Mobile processor, having 512MB of RAM, and running Red Hat Linux 8.0 (kernel version 2.4.18–14). The version of MAGICA used was 1.1, which was executed on version 4.2 of the Mathematica kernel.[31]

## 9.2 Inferred Shapes' Composition

Table VII presents results due to the shape-tracking analysis alone. The "Shapes" column shows the total number of shapes that were inferred for each benchmark. These include the shapes of the original program variables as well as the shapes of temporaries that were introduced in the steps that led up to

[31]For measurements with slightly older versions of MAGICA and Mathematica, see Joisha [2003]. The platform there was a 440 MHz UltraSPARC-IIi, with 128MB of RAM, and ran Solaris 7.

Table VII. Composition of Inferred Shapes

| Benchmark | Shapes | Explicit Shapes % | Explicit Shapes Scalar | Explicit Shapes Nonscalar | Symbolic Copies (%) | Type Inference Timings (secs) Kernel | Type Inference Timings (secs) MathLink |
|---|---|---|---|---|---|---|---|
| adpt | 188 | 42.55 | 70 | 10 | 62.96 | 9.63 | 1.37 (12%) |
| capr | 245 | 32.65 | 72 | 8 | 61.82 | 12.40 | 1.50 (11%) |
| clos | 64 | 100 | 21 | 43 | 0.00 | 1.88 | 1.02 (35%) |
| crni | 139 | 100 | 64 | 75 | 0.00 | 3.62 | 1.15 (24%) |
| diff | 92 | 91.30 | 70 | 14 | 62.50 | 9.85 | 1.65 (14%) |
| dich | 143 | 100 | 94 | 49 | 0.00 | 12.20 | 1.40 (10%) |
| edit | 83 | 28.92 | 19 | 5 | 62.71 | 1.59 | 0.99 (38%) |
| fdtd | 192 | 100 | 38 | 154 | 0.00 | 1.88 | 1.08 (36%) |
| fiff | 88 | 100 | 58 | 30 | 0.00 | 1.92 | 1.10 (36%) |
| nb1d | 163 | 10.43 | 12 | 5 | 64.38 | 3.56 | 1.12 (24%) |
| nb3d | 118 | 16.10 | 14 | 5 | 37.37 | 2.03 | 1.02 (33%) |
| bari | 73 | 100 | 32 | 41 | 0.00 | 0.20 | 1.09 (84%) |
| baye | 60 | 100 | 35 | 25 | 0.00 | 1.97 | 0.92 (32%) |
| brt1 | 45 | 80.00 | 27 | 9 | 66.67 | 5.25 | 1.12 (18%) |
| brt2 | 61 | 100 | 37 | 24 | 0.00 | 39.30 | 3.30 ( 8%) |
| sunl | 143 | 100 | 123 | 20 | 0.00 | 2.96 | 1.05 (26%) |
| vfin | 94 | 70.21 | 46 | 20 | 57.14 | 0.93 | 0.93 (50%) |

type determination.[32] For instance, temporaries were introduced during the SSA conversion step that preceded the type inference phase. It additionally includes variables that were introduced by the effective splitting of $\dot{\Phi}(P)(s, t)$ into copies during the SSA inversion phase. Array copy propagation and dead-code elimination were performed prior to type determination, so trivial copies didn't contribute to these numbers.

The "Explicit Shapes" column indicates the percentage of inferred shapes that were 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 can 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 resort to a general array addition routine.

All nonexplicit shapes are symbolic. The "Symbolic Copies" column displays the percentage of symbolic shapes that were inferred to be identical. This deduction happened as a result of the shape-tracking analysis described in Section 7.1. This was a metric on which we significantly improved over all past efforts simply because past efforts didn't look at shapes at a symbolic level. Establishing that one symbolic shape is exactly the same as another allows a translator not only to avoid unnecessary shape checks, but to also reduce the runtime shape computation overhead. Table VII shows that when shapes were symbolic, a large percentage was usually determined to be identical, even in programs that only manipulated two-dimensional arrays. This indicates the potential for such inferences to considerably lower shape-related

[32]MAGICA also infers intrinsic type and value range, in addition to shape [Joisha and Banerjee 2003b].

runtime overheads. As far as we know, no previous approach to shape determination for the MATLAB/APL class of languages can make this kind of an inference.

## 9.3 Inference Timings

The total time to obtain all type attributes—that is, shape, intrinsic type, and value range—for each benchmark is presented in the last column of Table VII. This was the sum of two parts: a "kernel" time, which is the time taken by the Mathematica kernel to infer the type attributes, and a "MathLink" time, which is the time for transferring an intermediate representation of the input program to the Mathematica kernel and to transfer the inferred types back. This data movement occurs over a special interprocess communication interface called MathLink [Wolfram 1999] and can be expensive. As Table VII shows, in 8 out of 17 benchmarks, the MathLink timings accounted for over 30% of the total type inference times. The MathLink channel has been improving with newer releases of Mathematica and it is possible that the total type inference timings will be closer to that shown in the "Kernel" subcolumn in future versions. Still, considering the quality of the inferences made, we believe that the current timings are acceptable, given that a typical deployment scenario would involve the interpreter during prototyping and code development, and an inference-capable compiler during production code generation.

## 9.4 Execution Timings and Memory Footprints of Compiled MATLAB Code

Ultimately, the information inferred by the techniques in this article has the potential to impact the code produced by a translator. We have implemented a source-to-source translator system called $\mathrm{M}^{\mathrm{AT}}{}_{2}\mathcal{C}$ that uses the inferences produced by MAGICA to efficiently compile MATLAB programs into stand-alone C code [Joisha 2003].

The speedups on the generated C codes for the first 11 benchmarks in Table VI, with respect to code produced by a commercial compiler for MATLAB called `mcc` (`mcc` is from the makers of MATLAB and relies solely on runtime resolution), ranged from 10% to over two orders of magnitude on a Solaris platform [Joisha and Banerjee 2003a; Joisha 2003]. Since the work of Joisha and Banerjee [2003a] and Joisha [2003], the benchmark suite has grown to 17 (the new entries are the last six in Table VI) and the testbed has migrated to the Linux platform mentioned in Section 9.1. Execution times on the current suite under the new setup against a more recent version of `mcc` have ranged between a slowdown of 0–30% in 4 programs to speedups in the remaining 13, of at least an order of magnitude in 6 of them. Execution times have also been measured against a recent version of the MATLAB interpreter enhanced with a dynamic JIT (just-in-time) compiler [The MathWorks, Inc. 2002a, 2002b]; the performance in this case ranged between a slowdown of 40–90% in 3 programs to speedups in the remaining benchmarks, speedups that were at least an order of magnitude in 2 of them.

Substantial savings in memory footprints have also been achieved due to storage optimizations permitted by the inferred shapes [Joisha and Banerjee

2003a]. For instance, on the 11 benchmarks in Table VI that were profiled on the Solaris testbed, average virtual memory size savings of between 0.7% and 139% were observed against code produced by the commercial MATLAB compiler; these savings ranged from 123KB to over 9MB in absolute terms [Joisha and Banerjee 2003a; Joisha 2003]. Measurements on the entire benchmark suite done on the Linux platform and against the more recent version of `mcc` show average virtual memory savings that range from losses of 0.6–13.2% in 5 programs to gains in the remaining 12, with savings over 46% in 5 of the latter.

While this article shows how the framework could be used to realize a systematic preallocation analysis, the analysis isn't reflected in the numbers reported in Table VII. This is because despite MAGICA's ability to perform the analysis, our MATLAB-to-C compiler currently doesn't handle its preallocation inferences. Thus, all performance improvements have been due to information gathered by the shape-tracking analysis alone. In fact, MAGICA is yet to fulfill the full potential of this framework, since it would entail more developmental work. For instance, only a subset of the identities in Table II have been currently coded into the system.

## 10. CONCLUSIONS

This article has presented an approach based on symbolic evaluation for inferring array shapes in array-based languages such as MATLAB. Unlike past 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.

Of course, the quality of the inferences is crucially dependent on how fully the various shape-related algebraic properties have been 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 achieved. However, the shape semantics of most of the language operators have simple algebraic properties that can be both easily identified and codified and from which shape inference benefits immediately accrue. Moreover, since not coding some algebraic identity doesn't mean incorrectness, but only a missed opportunity at simplification and a useful inference, systems based on these methods can be built incrementally, eventually trading quality of inference for runtime efficiency. We believe that our implementation in MAGICA confirms this point by demonstrating that a system capable of high-quality inferences can be both practically built and used.

## ACKNOWLEDGMENTS

The article has benefited from feedback on the manuscript by Christopher W. Fraser, and from discussions with Erik Ruf and David R. Tarditi.

## REFERENCES

Adams, J. C., Brainerd, W. S., Martin, J. T., Smith, B. T., and Wagener, J. L. 1992. *FORTRAN 90 Handbook*. McGraw-Hill, New York.

Almási, G. 2001. MaJIC: A MATLAB Just-In-Time Compiler. Ph.D. thesis, University of Illinois at Urbana-Champaign.

Almási, G. and Padua, D. A. 2002. MAJIC: Compiling MATLAB for speed and responsiveness. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 294–303.

Ancourt, C. and Nguyen, T. V. N. 2001. Array resizing for scientific code debugging, maintenance and reuse. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, New York, 32–37.

Banerjee, U. 1993. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic, Norwell, MA.

Budd, T. 1988. *An APL Compiler*. Springer Verlag, New York City.

Chauveau, S. and Bodin, F. 1998. Menhir: An Environment for high performance MATLAB. In *Proceedings of the 4th International Workshop on Languages, Compilers, and Runtime Systems*. LNCS, vol. 1511. Springer Verlag, 27–40.

Ching, W.-M. 1986. Program analysis and code generation in an APL/370 compiler. *IBM J. Res. Dev. 30*, 6 (Nov.), 594–602.

Cytron, R., Ferrante, J., Rosen, B. K., and Wegman, M. N. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct.), 451–490.

De Rose, L. A. 1996. Compiler techniques for MATLAB programs. Ph.D. thesis, University of Illinois at Urbana-Champaign.

De Rose, L. A. and Padua, D. A. 1999. Techniques for the translation of MATLAB programs into FORTRAN 90. *ACM Trans. Program. Lang. Syst. 21*, 2 (Mar.), 286–323.

Dershowitz, N. and Plaisted, D. A. 2001. Rewriting. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, eds. vol. 1. Elsevier, Amsterdam, The Netherlands.

Gupta, R. 1993. Optimizing array bounds checks using flow analysis. *ACM Lett. Program. Lang. Syst. 2*, 1–4, 135–150.

Hindley, J. R. 1969. The principal type-scheme of an object in combinatory logic. *Trans. American Math. Society 146*, 29–60.

Jay, B. C. and Steckler, P. A. 1998. The functional imperative: Shape! In *Proceedings of the 7th European Symposium On Programming*. LNCS, vol. 1381. Springer Verlag, 139–153.

Joisha, P. G. 2003. A type inference system for MATLAB with applications to code optimization. Ph.D. thesis, Northwestern University.

Joisha, P. G. and Banerjee, P. 2001a. Computing array shapes in MATLAB. In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 2624. Springer Verlag.

Joisha, P. G. and Banerjee, P. 2001b. Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In *Proceedings of the ACM SIGAPL Conference on Array Processing Languages*. ACM, New York, 6–21.

Joisha, P. G. and Banerjee, P. 2002. Implementing an array shape inference system for MATLAB using Mathematica. Tech. Rep. CPDC–TR–2002–10–003, Northwestern University.

Joisha, P. G. and Banerjee, P. 2003a. Static array storage optimization in MATLAB. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 294–303.

Joisha, P. G. and Banerjee, P. 2003b. The MAGICA type inference engine for MATLAB. In *Proceedings of the 12th International Conference on Compiler Construction*. Lecture Notes in Computer Science, vol. 2622. Springer Verlag, 121–125.

Joisha, P. G., Shenoy, U. N., and Banerjee, P. 2000. An approach to array shape determination in MATLAB. Tech. Rep. CPDC–TR–2000–10–010, Northwestern University.

Kaplan, M. A. and Ullman, J. D. 1978. A general scheme for the automatic inference of variable types. In *Proceedings of the 5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 60–75.

Knight, K. 1989. Unification: A multidisciplinary survey. *ACM Comput. Surv. 21*, 1, 93–124.

Malishevsky, A. 1998. Implementing a runtime library for a parallel MATLAB compiler. M.S. thesis, Oregon State University.

McCosh, C. 2003. Type-Based specialization in a telescoping compiler for MATLAB. Tech. Rep. TR03–412, Rice University.

Milner, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 3 (Dec.), 348–375.

Mitchell, J. C. 1996. *Foundations for Programming Languages*. The MIT Press, Cambridge, MA.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

Quinn, M. J., Malishevsky, A., Seelam, N., and Zhao, Y. 1998. Preliminary results from a parallel MATLAB compiler. In *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, S. Sahni, Ed. IEEE Computer Society Press, 81–87.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. Association Comput. Mach. 12*, 1 (Jan.), 23–41.

Tenenbaum, A. M. 1974. Type determination in very high-level languages. Ph.D. thesis, Rep. NSO-3, New York University.

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

The MathWorks, Inc. 2002a. Accelerating MATLAB: The MATLAB JIT-Accelerator. At `http://www.mathworks.com/company/newsletters/digest/sept02/accel_matlab.pdf`.

The MathWorks, Inc. 2002b. The MathWorks announces release 13 with major new versions of MATLAB and Simulink. At `http://www.mathworks.com/company/pressroom/index.shtml/article/332`.

Tremblay, J. P. and Manohar, R. 1975. *Discrete Mathematical Structures with Applications to Computer Science*. Computer Science Series. McGraw-Hill, New York.

Walther, C. 1988. Many-Sorted unification. *J. Assoc. Comput. Mach. 35*, 1, 1–17.

Weisstein, E. W. 2005. Hilbert matrix; From *MathWorld*—A Wolfram web resource. At `http://mathworld.wolfram.com/HilbertMatrix.html`.

Wiedmann, C. 1979. Steps toward an APL compiler. In *Proceedings of the ACM SIGAPL Conference on Array Processing Languages*, A. Anger, Ed. ACM, New York, 321–328.

Wolfram, S. 1999. *The Mathematica Book*, 4th ed. Wolfram Media, Champaign, IL.

Xi, H. and Pfenning, F. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language, Design, and Implementation*. ACM, New York, 249–257.

Received May 2004; revised January 2005; accepted May 2005