NORTHWESTERN UNIVERSITY

A Type Inference System for MATLAB with Applications to Code Optimization

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of ELECTRICAL AND COMPUTER ENGINEERING

By

Pramod G. Joisha

EVANSTON, ILLINOIS

December 2003

Abstract

This thesis presents techniques for deducing valuable array shape information in typeless array-based languages like MATLAB, even when the shapes may be statically unknown. The approach is based on an algebraical modelling of the language's array shape semantics.

The dissertation makes five primary contributions: (1) it shows how to model shape semantics so as to enable precise characterizations that are independent of an array's dimensionality; (2) it presents a rewriting methodology that uses the characterizations to symbolically determine an array expression's shape in $O(R\|s\|^3 \log \|s\|)$ time, where $\|s\|$ is the number of nodes in a rooted ordered tree representation of a shape term s, and R is an upper bound on an array's canonical rank; (3) it presents a shape-tracking optimization that detects equivalent shapes, permitting a translator to eliminate redundant shape checks and to generate specialized code even when the shapes may be compile-time unknowns; (4) it shows a preallocation optimization that determines the set of all possible shapes that an expression may assume during program execution; and (5) it presents an interference graph-based algorithm that uses type and control-flow information to coalesce array storage. The dissertation also substantiates the techniques by presenting performance numbers of an implementation on a benchmark suite.

Acknowledgements

At the outset of this thesis, I would like to express my gratitude and thanks to my advisor, Prithviraj Banerjee, for making this work possible. Prith has been a tremendous source of support and encouragement, and working under him has been both a rewarding and pleasurable experience for me.

I would also like to thank the other members of my committee—David Padua, Alok Choudhary, Hai Zhou and Lawrence J. Henschen—for their valuable comments and suggestions. Special thanks to David Padua for taking the time to come to my proposal defense all the way from the University of Illinois at Urbana-Champaign, and to once again attend my final defense by teleconferencing from the IBM T. J. Watson Research Center at Yorktown Heights, New York.

I would like to thank Nagaraj Shenoy for initial discussions on this thesis topic. Thanks also to Siddhartha Chatterjee, Manish Gupta, James R. Larus, Bjarne Steensgaard, David Tarditi and Benjamin G. Zorn for valuable comments, suggestions and feedback that have greatly improved this work.

Thanks to Christopher Bachmann, Predrag S. Bundalo and other members of the ECE computing staff for always responding swiftly to support requests, and for ensuring that our systems were always up and running. Thanks also to Nancy Singer, Deneen Marie Bryce, Judith Stein, Carol Surma and others members of the ECE office staff for their help throughout my time in this department.

Thanks to my friends in Evanston for making my stay here memorable. Special thanks to Bernice and Charles Klostermann for their help, kindness and warmth; I shall dearly miss them. Special thanks also to Kaveri and Girish for

the wonderful weekend getaways at Milwaukee. I shall also dearly miss them.

Finally, I wish to thank my family for their support throughout this endeavor. I am deeply grateful to my parents for their love, encouragement and support, and for the umpteen sacrifices that they have made for me. I regard myself as being extremely fortunate to have them; it is to them that I dedicate this thesis.

To My Parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Productivity increased only when we were able to raise the level of abstraction.
— ALBERTO L. SANGIOVANNI-VINCENTELLI
"The Tides of EDA,"
Design Automation Conference Keynote Address, June 2003.

Problem abstraction has always been an important facilitator of scientific and technological progress. The need for higher levels of abstraction has become even more acute in modern times as science and technology take on new and more ambitious problems, considerably more complicated both in scale and scope. Tools embodying more efficacious methodologies to problem expression, analyses and solution are being perpetually sought after to ease time-to-market pressures, improve productivity and to enhance the creative process by relieving it from more mundane tasks. Spurred by such demands, 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 very high-level proprietary languages like MATLAB and IDL.[1] This phenomenon is interesting when viewed against the backdrop of a language like FOR-

[1]MATLAB® is a registered trademark of The MathWorks, Inc. IDL® is a registered trademark of Research Systems, Inc.

TRAN 90 [4] that has provided a similar—yet not exactly the same—paradigm of programming, but that hasn't as steadily grown from strength to strength, even evidencing perhaps a slowing rate of pervasion in niche domains such as scientific computing [63, page 2]. What have been the chief inducers of MATLAB's ascent? Arguably, an important factor has been a programming philosophy founded on the array as the central data structure, coupled with a tacit typing policy for data.

1.1 An Evolution in Array Semantics

Though arrays have been around as integral offerings since the time of FORTRAN, ALGOL, Pascal and other third generation languages, their sophistication and role has greatly varied. Classic Pascal [48], for instance, fixed array sizes at compile time, and more importantly, by treating the size as part of an array's type, precluded the writing of *polyshape* procedures and functions.[2] FORTRAN 77 also statically fixed array sizes, though it did permit polyshape user-defined procedures through devices such as *adjustable array* dummy parameters [59]. ALGOL 60 allowed greater flexibility by supporting the allocation and parameter-passing of arrays with dynamic bounds. Still, *general* array manipulations in ALGOL 60, as in FORTRAN 77 and Pascal, had to be expressed at the elemental level, perhaps through the use of program loops, because the language's operators were only defined on scalars.[3] This changed in FORTRAN 90, which besides supporting dynamically sized arrays, provided intrinsic operators that could also work on array objects. Nevertheless, the expressive power of arrays in FORTRAN 90 is somewhat tempered because most intrinsic operators have their scalar semantics extended in a common, *element-by-*

[2] The latter has been cited as one of the most serious drawbacks of the language [60, 40]. It has been addressed in Level 1 of the ISO 7185:1983 Pascal Standard by the so-called *conformant array* [75]; conformant arrays have in turn been subsumed by the more general *type schemata* in the ISO 10206:1993 Pascal Standard.

[3] Excluded are special-case supports like the handling of strings, or the reading and writing of whole arrays through the `READ` and `WRITE` statements in FORTRAN 77.

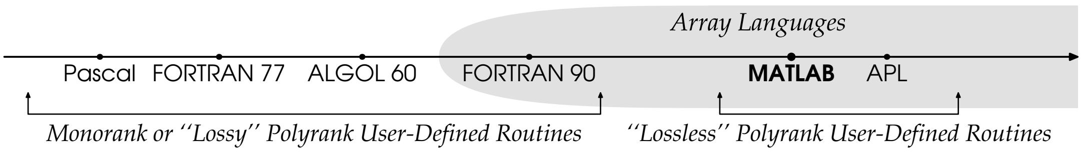

Fig. 1.1 Array Sophistication and Role

element way, to arrays [4].[4] Rather than use a scalar view of data as a basis and then extend it to arrays, MATLAB and APL took alternate routes—MATLAB began with a matrix view of data that was later extended to arrays,[5] while APL began by considering all expressions as first-class arrays. In both, scalars correspond to degenerate cases—1×1 matrices in MATLAB and zero-dimensional arrays in APL. Consequently, even though FORTRAN 90 also considers scalars as zero-dimensional arrays, language operators in APL and MATLAB tend to have a higher, and richer, vector semantics. For example, even subscript expressions in MATLAB and APL can be arbitrary arrays. And all this is apart from the fact that in both MATLAB and APL, array sizes, *as well as dimensionalities*, can be dynamically set!

Figure 1.1 shows our take on the sophistication and role of arrays in MATLAB, in relation to some other languages. The ordering criteria have primarily been the ability to write polyshape and truly polydimensional code (see § 1.4), operator semantics and a user-friendly tacit array typing policy (see § 1.3). Figure 1.1 is intended to be only suggestive; if one were to consider Pascal with conformant arrays, or the fact that Pascal always allowed for the dynamic allocation of arrays, the ordering could quite possibly be different.

1.2 Semantic Contrasts in APL and MATLAB

Array semantics in APL conduce to even more higher levels of abstraction than in MATLAB. A case in point is the matrix-multiplication operation that is defined in

[4]Out of the 113 intrinsic procedures in FORTRAN 90, 65 are elemental.

[5]Multidimensional array support was introduced in version series 5 of MATLAB.

the standard linear algebra sense in MATLAB, becoming an elementwise multiplication when one of the operands is a scalar. In APL however, matrix multiply is supplanted by an inner tensor product [80] that *seamlessly* degenerates to a traditional matrix multiply when the operands are appropriately conforming matrices. In fact, APL even leaves the specification of the "addition" and "multiplication" operations to the user, thereby permitting the natural expression of the operation in other algebraic domains. Despite the greater array sophistication, MATLAB has proven to be much more popular than APL. Some of the reasons for this may be the language's structured control flow,[6] its simpler, more intuitive, syntax,[7] and its strong linear algebra support, including facilities for sparse matrices.

1.3 Tacit Array Typing

Another ingredient that may have contributed to MATLAB's popularity is its lack of explicit type declarations. The language uses a typing policy that is implicit from the semantics of its operators (called *built-in functions*) and the execution context; nowhere is it expressly given in the program. Furthermore, an expression's type is allowed to dynamically change. Although such features—particularly implicit and dynamic *array rank* and *array shape* typing, and array polymorphism—may have greatly promoted the language's level of abstraction, ease of use and success, they have also severely burdened its execution support, forcing implementations to usually cope through interpretation. Prior knowledge of an array's type attributes, such as rank, shape and intrinsic type,[8] is thus desirable because of its potential to improve execution performance and efficiency. For instance, knowing an array's shape

[6]Normal flow of control in APL is altered through a *computed branch* [80]; though vendor versions of APL exist that have high-level conditional or looping constructs, the computed branch is the only control-flow construct provided in the ISO 8485:1989 APL Standard.

[7]While APL has been criticized on the grounds of a cryptic syntax and a non-standard character set, it has also had vigorous proponents of its notation [46, 36].

[8]Intrinsic type determination is also an issue in MATLAB and APL because their arrays have an orthogonal intrinsic type attribute that defines the arithmetic possible on the contents.

```
1   d ← max(b*b, abs(a)*b);
2   a ← atan2(b, b');
3   d ← (b+d)*fix(b);
4   while p1,
5      c ← a*b;
6      a ← c+a.^b;
7      b ← c-a./c;
8      while p2,
9         e ← e*d;
       end;
    end;
10  h ← b.^b-a;
```

Fig. 1.2 Tacit Array Typing in Action

in advance could eliminate redundant shape conformance checks, permit compile-time verification of shape correctness and allow for the preallocation of array storage.

To understand how array typing works in MATLAB, consider the synthetic code fragment in Figure 1.2. 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.[9] If either `a` or `b` is a scalar, the other variable can be an arbitrary array and the result is produced by multiplying the scalar with the elements of the array. On the other hand, if both `a` and `b` are nonscalar, they have to be conforming matrices for the operation to be valid, i.e., their shapes have to be $p \times q$ and $q \times r$ respectively [66]. In current static shape inference systems for MATLAB [81, 21, 32], the shape of `c`, at least on the first iteration, can be inferred if the initial shapes of `a` and `b` are known at compile time. However, if either of these initial shapes are unknown, even partially (some array extents known, others unknown), the shape of `c` will also be considered unknown and conservative code to perform run-time resolution will be generated. Unknown shapes have a propagative effect in that one unknown shape could lead to a whole set of shapes becoming unknown. This would be the case for the code in Figure 1.2 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 1.2, we shall see that:

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

INFERENCE 1.
The shapes of `d` on both Lines 1 and 3 are identical for *any* initial `a` and `b`. This means that a translator could replace shape conformance checks against Lines 1 and 3 by a single one 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 `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.

1.4 "Lossless" Polyrank User-Defined Routines

A unique feature that distinguishes MATLAB and APL from most other languages is the ability to write polyrank routines (single routines that can be invoked on arrays of various ranks) such that the formal parameter's rank always reflects the actual parameter's rank. The term rank in this thesis has the FORTRAN 90 and APL connotation—it means the dimensionality of an array. While languages like 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 is left unspecified as may happen in C if pointer parameters are used. The only way to communicate an actual parameter's rank into a user-defined routine in such languages is either by passing an additional parameter or by other mechanisms such as globally visible memory locations.

To concretely illustrate this distinction, consider the problem of adding to an array of rank n a generalization of the Hilbert matrix [96] to n dimensions. Specifically, we define a *Hilbert array* of n dimensions as having the elements

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

```
   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. 1.3 Lossy Polyrank Hilbert Array Accumulation in FORTRAN 90

Figure 1.3 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* array dummy parameter [4]. However, this polyrank trait is lossy because the dummy parameter is statically fixed to be a one-dimensional array. In Figure 1.4 is a lossless polyrank MATLAB version that relies on a reflective call to `size` to access the actual's rank.

Although the ability to write lossless polyrank code permits the 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

```
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. 1.4 Lossless Polyrank Hilbert Array Accumulation in MATLAB

in MATLAB because these efforts practically considered a monorank MATLAB by restricting array dimensionalities to 2 [81, 21, 32, 7]. This thesis will later show how automated inference techniques could be used to deduce that:

> Inference 4.
> Irrespective of their ranks, the shapes of `a` and `b` in Figure 1.4 are always identical under every successful execution of the function.

1.5 Dissertation Outline

This thesis presents an approach to inferring shapes in languages that exhibit a tacit array typing policy of the kind discussed in § 1.3. The approach is based on a calculus that models array shape semantics algebraically and a rewriting methodology that evaluates expressions in this calculus. The thesis later shows how the approach can be used to automatically optimize out shape checks, produce specialized code even when shapes are compile-time unknowns, preallocate arrays and coalesce storage.

Chapter 2 discusses background material for the problem. It begins by discussing where the bottlenecks in executing an array expression lie, and the broad approaches that have been used in the past to overcome them. The chapter elaborates on the approaches, covering related prior work. The problem of array shape determination is then discussed along with limitations of past approaches.

Chapter 3 describes a key property in MATLAB called selective rank demotion. The chapter shows how this property may be used to devise a shape calculus that needn't wrestle with the possibility of different arrays having different ranks.

Chapter 4 presents a shape calculus for MATLAB. The calculus is based on a modelling of the language's shape semantics through shape-tuple equations. The chapter shows how these equations can be used to uncover an algebraic structure on shapes. Shape algebras are desirable because they allow the characterization of general properties pertaining to shape that a code translator could take advantage of even in the absence of explicit shape information at translation time.

Chapter 5 presents a symbolic evaluator to automate the shape calculus of the previous chapter. At the heart of the evaluator is a rewriting process that represents algebraic identities using rules and that handles the properties of commutativity and associativity by special structural transformations. The chapter gives sufficient conditions for the evaluator to be strongly and uniquely normalizing. It finishes by proving the soundness of the evaluator and deriving its computational complexity.

Two important applications of the evaluator for shape analysis are discussed in Chapter 6. The first, referred to as shape tracking, establishes the shape equivalence of two array expressions along a control-flow path. The second, called preallocation, determines an "upper bound" on an array's shape during program execution.

Chapter 7 discusses the applicability of the inference methods of the previous chapters to a large class of language operators in MATLAB. It presents a proactive classification scheme and shows how the inference methods can be extended to cover an important group of operators in the language. By identifying a class of operators that are amenable to shape analyses, the taxonomy also serves as a useful guide to language designers contemplating extensions to the language.

Chapter 8 briefly discusses an extensible inference engine called MAGICA that realizes the techniques described in the thesis. It explains how to use MAGICA in an interactive mode and shows its operation on a program that finds the transitive closure of a graph and a code fragment that computes the nth Fibonacci number.

Chapter 9 presents an algorithm that uses inferred type information to coalesce array storage. The algorithm operates in two phases. In the first phase, an interference graph is created to determine classes of variables that don't contest storage. The graph is constructed using type information and results from a live-range analysis. In the second phase, the classes are partitioned using control flow and type information into groups of arrays that will ultimately share the same storage.

Chapter 10 reports measurements on the performance of our implementation on a benchmark suite consisting of mainly publicly available MATLAB programs. The techniques were assayed by collecting numbers on a variety of metrics such as the inferred shape compositions, inference times, static storage reductions, memory footprints (both dynamic and average), and the run times of the compiled codes.

Finally, chapter 11 concludes the dissertation by summarizing its contributions, and suggesting possible directions for future research.

Chapter 2

Background

An attempt is made to demonstrate that the use of APL is growing in an epidemic fashion. The theory of epidemic processes is applied in an approximate manner by means of data provided by a nearly-exhaustive bibliography given as an appendix. APL is proved to be undoubtedly epidemic.
— J. C. RAULT and G. DEMARS
"Is APL epidemic? or a Study of its Growth through an Extended Bibliography,"
Proceedings of the ACM SIGAPL International Conference on APL, 1972. Page 1.

While the aforementioned features promote MATLAB's prowess as a rapid prototyping tool, they also impede, and indeed challenge, the efficient execution of its programs. Factoring in all those high-level semantics in an implementation entails an extensive ancillary processing effort at the operation level, an overhead that becomes extravagant for small-sized arrays. Thus the issue of transparently boosting the performance of languages that offer programming models coincident with that of MATLAB's without sacrificing their high-levelness is an important problem, one that has actively engaged the interest of researchers since the heyday of APL.

This chapter reviews some of the earlier approaches to improving performance, and shows why the inference of types is really the ticket to its effective solution. The attributes of type that need to be inferred depend on the target language chosen by the translator. If it is a low-level language like C, at least the attributes of intrinsic type and shape have to be inferred to bring about an efficient translation. Although capable solutions have been presented for the problem of intrinsic type determination, there appears to be an opportunity for improvement as far as shape inference is concerned. This chapter later discusses previous efforts on the automatic determination of shapes in typeless array-based languages along with their limitations. It is at least some of these limitations that we seek to surmount in our work. Finally, the chapter briefly discusses the bottom-up design of tacitly typed array languages that are more amenable to the compile-time analysis of shape.

2.1 The Execution Overhead

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 [98]. Thus, the total execution time for an array expression could be considered as comprising two parts: (1) a fixed *set-up* time, which is the time expended on the preprocessing phase; and (2) a variable *cumulative per-element* time, which is the time required for computing all the elements in the array result [98].

The main characteristic of the preprocessing phase is that it is composed of subtasks that can be performed in constant time, independent of the number of elements in the array arguments to the expression. It normally spans four subtasks:

1. verification of conformance of the expression's operands (if any) to the requirements of the expression's operators,
2. determination of the nature, i.e., the type, of the result (if any),
3. allocation, or reallocation, of storage for the result, and
4. dispatch of control to one among various code fragments, each of which calculates the result for some specific type configuration of the arguments.

The evaluation phase consists of subtasks whose running times are dependent on the number of elements N in the result. These would include initializing the output, or copying data from an old location in case the result is a resized array.

The cumulative per-element time is thus of the form $p \times f(N)$, where the *per-element time* p is the time outlay on a single element of the result. From a user's perspective, a fair share of the cumulative per-element time is time spent doing "useful" work.

The constant set-up times thus represent overhead that are usually much larger than the per-element times. (On an old CYBER 175C Series 100 system, the set-up and per-element times for release 13 of Control Data's APL Version 2 were reported as being 129 microseconds and 1.6 microseconds respectively [98].) Hence, the larger an expression's array operands, the better the chance that its set-up time doesn't overwhelm its cumulative per-element time. The aim of the game is therefore to figure out how automated techniques can be used to reduce the overall (i.e., across a program) set-up and cumulative per-element times, and in particular, how the overall set-up times can be made substantially lower than the overall cumulative per-element times, especially when the arrays manipulated are small in size.

2.2 Performance Improving Strategies

Broadly speaking, three strategies have been used to reduce the execution time in typeless array-based languages: (1) type inference, (2) source-to-source transformations, and (3) specialized libraries and parallelization. Of these, only the first two have the capacity to decrease both the set-up and cumulative per-element times. The third strategy targets at only reducing the cumulative per-element times. Hence, it has mostly been used in enterprise-scale applications where the arrays processed are so large that the set-up times are sufficiently negligible in relation to the cumulative per-element times for them to be overlooked [71, 14, 87, 45, 73, 23]. Table 2.1, on pages 14 to 18, summarizes previous work on the basis of these strategies.

Historically, researchers have usually pursued the three strategies separately, because for the most part, each of them can be realized with little to no support from the others. For instance, classic compiler optimizations such as global common-subexpression elimination and copy propagation [5] are examples of source-to-source

	Effort	MAGICA MAT2C [55, 54]	FALCON [31, 32, 30]	AMF [70, 69]	MaJIC [7, 6]	Menhir [21]
Type Inference	Static Inference	exp/sym	exp	AVAIL	none	dirs+exp
	Dynamic Inference	NPA	NPA		APA	NPA
	Solution Method	UF+IT	IT		IT	IT
	Array Dimensionality	$d \geq 2$	$d = 2$		$d = 2$	$d = 2$
Source-to-Source Transformations	Preallocation	✓	✓	✓	✓	✗
	Vectorization	✗	✗	✓	✗	✗
	Algebraic Simplifications	✗	✗	✓	✗	✗
	Common-Subexpression Elimination	✓	✗	✗	✗	✗
	Copy Propagation	✓	✗	✗	✗	✗
	Constant Propagation	✓	✗	✗	✓	✗
	Constant Folding	✓	✗	✗	✓	✗
	Dead-Code Elimination	✓	✗	✗	✗	✓
	Parallelization	✗	✗	✗	✗	✓

Legend: exp ≡ explicit types | dirs ≡ directives/declarations | sym ≡ symbolic types
NPA ≡ near point of action | APA ≡ away from point of action
UF ≡ unification-based solver | IT ≡ iterative solver
AVAIL ≡ prior availability of inference information assumed | d ≡ array dimensionality
■ ■ MATLAB effort | ■ APL effort | ■ Other language (e.g., IDL) effort

Table 2.1 Related Efforts at Speeding Up Typeless Array-Based Languages (contd. on next page)

	Effort	MATCH [11, 77]	Otter [81, 65]	MVM [43]	mcc (version 2.2) [67]	telescoping languages [20]
Type Inference	Static Inference	dirs	exp	none	exp	AVAIL
	Dynamic Inference	NPA	NPA	NPA	NPA	
	Solution Method	none	?	none	none	
	Array Dimensionality	$d = 2$	$d = 2$	$d = 2$	$d = 2$	
Source-to-Source Transformations	Preallocation	✗	✗	✗	✓	✓
	Vectorization	✗	✗	✗	✗	✓
	Algebraic Simplifications	✗	?	✗	✗	✗
	Common-Subexpression Elimination	✗	?	✗	✗	✓
	Copy Propagation	✗	?	✗	✗	✗
	Constant Propagation	✗	?	✗	✗	✓
	Constant Folding	✗	?	✗	✓	✗
	Dead-Code Elimination	✗	?	✗	✗	✗
	Parallelization	✓	✓	✓	✗	✗

Legend: exp ≡ explicit types | dirs ≡ directives/declarations | sym ≡ symbolic types
NPA ≡ near point of action | APA ≡ away from point of action
UF ≡ unification-based solver | IT ≡ iterative solver
AVAIL ≡ prior availability of inference information assumed | d ≡ array dimensionality
■ ■ MATLAB effort | ■ APL effort | ■ Other language (e.g., IDL) effort

Table 2.1 Related Efforts at Speeding Up Typeless Array-Based Languages (continued)

	Effort	MultiMATLAB [71]	Matpar [87]	MITMATLAB [45]	ParAL [73]	RTExpress [14]
Type Inference	Static Inference	none	none	none	exp (shapes fixed)	none
	Dynamic Inference	NPA	NPA	NPA	NPA	NPA
	Solution Method	none	none	none	none	none
	Array Dimensionality	$d = 2$	$d = 2$	$d = 2$	$d = 2$	$d = 2$
Source-to-Source Transformations	Preallocation	✗	✗	✗	✗	✗
	Vectorization	✗	✗	✗	✗	✗
	Algebraic Simplifications	✗	✗	✗	✗	✗
	Common-Subexpression Elimination	✗	✗	✗	?	✗
	Copy Propagation	✗	✗	✗	?	✗
	Constant Propagation	✗	✗	✗	?	✗
	Constant Folding	✗	✗	✗	?	✗
	Dead-Code Elimination	✗	✗	✗	?	✗
	Parallelization	✓	✓	✓	✓	✓

Legend: exp ≡ explicit types | dirs ≡ directives/declarations | sym ≡ symbolic types
NPA ≡ near point of action | APA ≡ away from point of action
UF ≡ unification-based solver | IT ≡ iterative solver
AVAIL ≡ prior availability of inference information assumed | d ≡ array dimensionality
■ ■ MATLAB effort | ■ APL effort | ■ Other language (e.g., IDL) effort

Table 2.1 Related Efforts at Speeding Up Typeless Array-Based Languages (continued)

	Effort	High-Performance APL Interpretation [84]	APL\3000 [49]	STSC [95]	APL/370 [22]	APLc [17]
Type Inference	Static Inference	none	none	dirs+exp	exp	exp
	Dynamic Inference	NPA	APA	NPA	NPA	NPA
	Solution Method	none	?	IT	IT	IT
	Array Dimensionality	$d \geq 0$	$d \geq 0$	$d \geq 0$	$0 \leq d \leq 7$	$d \geq 0$
Source-to-Source Transformations	Preallocation	✗	✗	✗	✗	✗
	Vectorization	✗	✗	✗	✗	✗
	Algebraic Simplifications	✗	✗	✗	✗	✗
	Common-Subexpression Elimination	✗	✗	✓	✗	✗
	Copy Propagation	✗	✗	✓	✗	✗
	Constant Propagation	✗	✗	?	✗	✗
	Constant Folding	✗	✗	✓	✗	✗
	Dead-Code Elimination	✗	✗	✓	✗	✗
	Parallelization	✗	✗	✗	✗	✗

Legend: exp ≡ explicit types | dirs ≡ directives/declarations | sym ≡ symbolic types
NPA ≡ near point of action | APA ≡ away from point of action
UF ≡ unification-based solver | IT ≡ iterative solver
AVAIL ≡ prior availability of inference information assumed | d ≡ array dimensionality
■ ■ MATLAB effort | ■ APL effort | ■ Other language (e.g., IDL) effort

Table 2.1 Related Efforts at Speeding Up Typeless Array-Based Languages (continued)

	Effort	APL Compilation for Vector Processing [18]	COMPC [23]	IDL [82]
Type Inference	Static Inference	dirs	exp (ranks and intrinsic types fixed?)	none
	Dynamic Inference	NPA	NPA	NPA
	Solution Method	none	IT	none
	Array Dimensionality	$d \geq 0$	$0 \leq d \leq 7$	scalar and $1 \leq d \leq 8$
Source-to-Source Transformations	Preallocation	✗	✗	?
	Vectorization	✗	✗	?
	Algebraic Simplifications	✗	✗	?
	Common-Subexpression Elimination	✗	✗	?
	Copy Propagation	✗	✗	?
	Constant Propagation	✗	✗	?
	Constant Folding	✗	✗	?
	Dead-Code Elimination	✗	✗	?
	Parallelization	✓	✓	✗

Legend: exp ≡ explicit types | dirs ≡ directives/declarations | sym ≡ symbolic types
NPA ≡ near point of action | APA ≡ away from point of action
UF ≡ unification-based solver | IT ≡ iterative solver
AVAIL ≡ prior availability of inference information assumed | d ≡ array dimensionality
■ ■ MATLAB effort | ■ APL effort | ■ Other language (e.g., IDL) effort

Table 2.1 Related Efforts at Speeding Up Typeless Array-Based Languages (contd. from previous page) transformations that can be extended in a straightforward way to the domain of array expressions without requiring the knowledge of types for their application. (See [95] for a discussion on some of these peephole optimizations as used in a commercial compiler for APL.) However, there also exist source-to-source transformations that can only be realized when types are available (see § 2.2.2). The same is true of parallelization strategies since they are typically based on the partitioning of array data across processors using information about an array's size or shape. Thus, while the three strategies are complementary to one another, type inference is a fundamental strategy in that it enhances the scope and power of the other two.

2.2.1 Type Inference

For the MATLAB statement `c ← a*b` discussed on page 5, the set-up time would include the time to establish whether `a` or `b` are scalars or conforming matrices, the time to determine the shape and intrinsic type of `c`, the time to allocate (or reallocate) storage for `c`, and the time to transfer control to either an elementwise multiplication routine, or an elementary matrix-multiplication procedure. Clearly, if `a` and `b` are merely scalars, this preprocessing is wastefully general. By deducing the characteristics of operands in advance, type inferences allow the omission of some, or all, of the subtasks associated with the preprocessing phase. In addition, they may permit a more optimized expression evaluation phase. For instance, if `a` and `b` in the current example were found to have the shapes of conforming small-sized vectors, their matrix-multiplication evaluation loops could be completely unrolled, eliminating the need for any loop-control code and improving the effectiveness of other optimizations such as software pipelining and instruction scheduling.

2.2.1.1 Makeup of a Type Inference Scheme

Ideally, we would like to infer the shape and intrinsic type of all MATLAB expressions statically, well in advance of their execution. However, this isn't always possible because an expression's type depends on the control-flow path exercised

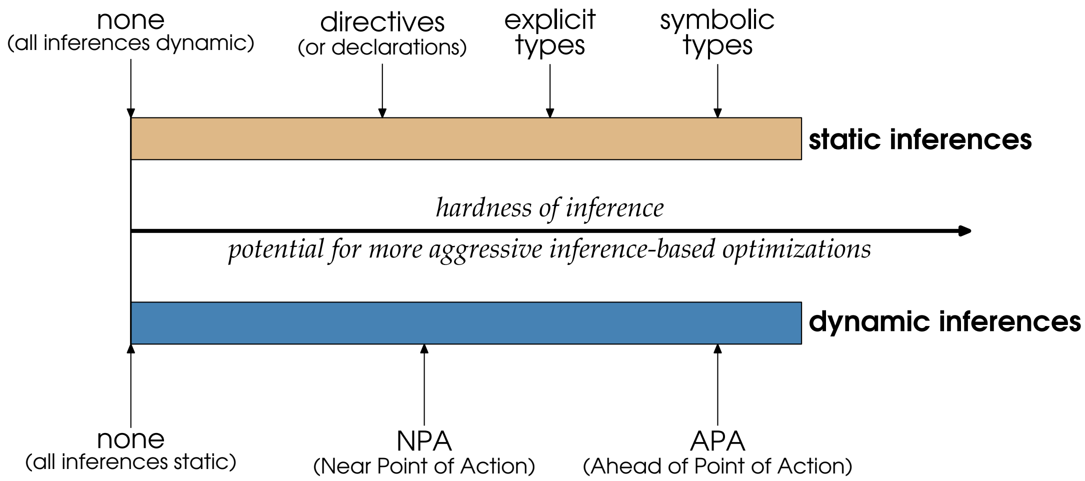

Fig. 2.1 Type Inference Scheme Makeup and Potential/Hardness Tradeoff

at run time, the precise determination of which is in general undecidable. Hence, as graphically shown in Figure 2.1, any type inference scheme for MATLAB that is also *complete* (i.e., guaranteed to produce an inference in every situation) would have to consist of a mixture of static and dynamic inferences.

The clincher is of course how ahead in the execution timeline an expression's type is deduced. For example, if `a` and `b` were statically inferred to be scalars, the MATLAB expression `a*b` could be compiled directly into a machine instruction well before its execution, thereby wholly obviating the set-up overhead. However, the more ahead of time a type is to be deduced, the more difficult the inference process becomes because a larger program context would have to be analyzed to maintain the same degree of specificity in an inference. Conversely, the less ahead of time a type is deduced, the easier it becomes to make the inference, though the inference itself becomes less useful either because opportunities for optimization have by then passed, or because the inference process begins to introduce its own overhead.

2.2.1.2 Static Inferences

The potential for speeding up tacitly typed array languages through the static validation of checks has been well known [13]. As far as we are aware, every previous method at statically inferring types in such languages has been by iterative data-flow analysis [95, 18, 22, 17, 23, 31, 21, 11]. The methods begin with some initial information about a program, which they then propagate through its various expressions and functions. This initial information may come from either user-provided directives and declarations [11, 18], or from expressions whose types are *explicit*, i.e., precisely ascertainable at compile time. Examples of such expressions are program constants and variable expressions with *explicit shapes*, i.e., shapes in which the dimensionality and array extents are static constants [22, 17, 23, 31]. A combination of both directives/declarations and explicit types could also be used [21, 95]; this is helpful when an expression's type cannot be precisely determined by automatic means, but manually specifying it could improve the quality of other inferences. The nice thing about data-flow-based inference methods is that it isn't necessary for the initial information to be precise for them to work. If it is conservative, a conservative inference is returned. Such an inference may still permit the static resolution of at least some of the type-dependent decisions that go into an expression's execution plan. Any remaining type-dependent decisions would then have to be resolved by performing a dynamic inference on the expression's type. Figure 2.1 displays the progression in hardness as we move from a directive/declaration-based static inference method to one founded on explicit types. Going a step further, we could envision a static inference method built on symbolic types. Its advantage would be the ability to derive useful static information even when the expression types aren't explicitly known. Such a method will be the focus of later chapters in this thesis.

2.2.1.3 Dynamic Inferences

A dynamic inference of an expression's type can be made at various points in the execution timeline. It could be done very close to the point in the program at which the expression is executed—we call this an NPA (Near Point of Action) dynamic inference—or it could be done at an early point in the execution timeline, ahead of the expression's execution—we call this an APA (Ahead of Point of Action) dynamic inference. Clearly, the prospects of using the inferred types to optimize an expression's execution is brighter in the latter case because there is a larger stretch of program context to work with. Since this also means that we are trying to forecast the type characteristics of an expression that is at least a couple of instructions away from being executed, a greater effort needs to be expended to maintain the same preciseness in the inference (as compared to an NPA dynamic inference).

MaJIC [7, 6] is an example of a system that relies on APA-style inferences to compile MATLAB code. Inferences in MaJIC happen at the granularity of functions. When control reaches the call-site of a function, MaJIC collects its arguments' types and propagates them into the function's body by an iterative data-flow process. This produces type inferences for the various variables defined and used in the function; these inferences are then used to optimize and dynamically compile the function.

To further understand the difference between NPA-style and APA-style dynamic inferences, consider how `mcc` [67, 92], a commercial compiler for MATLAB, translates a MATLAB expression. In version 2.2 of `mcc`, the statement `c` $\leftarrow$ `a*b` gets compiled into the following C code when `a` and `b` aren't *manifest* array constants.

```
mlfAssign(&c, mclMtimes(mclVv(a, "a"), mclVv(b, "b")));
```

The function calls `mlfAssign` and `mclMtimes` above correspond to the assignment and matrix-multiplication operations in the MATLAB source. The color coded `a`, `b` and `c` symbols are C pointers to data descriptors that `mcc` generates for every original MATLAB array. These data descriptors are instances of an `mxArray` C structure whose fields keep track of various array details such as the number of dimensions, the extents along the various dimensions, and the intrinsic type.[1]

The implementations of `mlfAssign` and `mclMtimes` carry out both the preprocessing as well as the evaluation of the assignment and matrix-multiplication

[1]This structure is defined in `matrix.h`, a header file in the MATLAB distribution.

```
typedef struct mxArray_tag mxArray;

struct mxArray_tag {
    char      name[mxMAXNAM];
    int       reserved1[2];
    void     *reserved2;
    int       number_of_dims;
    int       reserved3[4];
    union {
        struct {
            void  *pdata;
            void  *pimag_data;
            void  *reserved4;
            int    reserved5[3];
        }   number_array;
    }   data;
};
```

Fig. 2.2 Definition of the `mxArray` C Structure

operations. On Solaris, this can actually be verified by using an invaluable tool called `truss` that makes it possible to peer "underneath the hood" of proprietary vendor-supplied shared libraries.[2] Figure 2.3 displays an abbreviated trace of the internal calls made by `mclMtimes`, obtained using `truss`. The initial set of calls shown, like `mxGetM`, `mxGetN` and `mxGetDimensions`, perform subtasks in the preprocessing phase. The `dgemm_` call occurs as part of the evaluation phase.

2.2.2 Source-to-Source Transformations

As indicated earlier, source-to-source transformations try to reduce both the set-up and cumulative per-element times by replacing certain code sequences by equivalent but more efficient ones. This could be either by identifying and casting out redundancies (common-subexpression elimination and dead-code elimination), or by

[2] `truss` traces system and user-function calls into and within shared libraries and belongs to a family of tools such as `apptrace` and `appcert` that takes advantage of the industry-standard System V Application Binary Interface (ABI) [88] for run-time linking.

```
% truss -u '*::*' ./matmul > & ! matmul.truss ; more matmul.truss
                       ...
-> libmatlb:mclMtimes(0x1b06d8, 0x1b0740, 0xff32ced0, 0x1)
  -> libmatlb:mlfInitFcn(0x1b06d8, 0x0, 0x0, 0x0)
  <- libmatlb:mlfInitFcn() = 0xff330a62
                 ...
  -> libmx:mxCheckNumArgs(0x1, 0x1, 0x2, 0x2)
  <- libmx:mxCheckNumArgs() = 1
  -> libmx:mxGetM(0x1b06d8, 0x0, 0x0, 0x0)
    -> libmx:mxGetDimensions(0x1b06d8, 0x0, 0x0, 0x0)
    <- libmx:mxGetDimensions() = 0x1b0710
  <- libmx:mxGetM() = 2
  -> libmx:mxGetN(0x1b06d8, 0x0, 0x0, 0x0)
    -> libmx:mxGetDimensions(0x1b06d8, 0x0, 0x0, 0x0)
    <- libmx:mxGetDimensions() = 0x1b0710
    -> libc:.mul(0x1, 0x3, 0x0, 0x0)
    <- libc:.mul() = 3
  <- libmx:mxGetN() = 3
                 ...
  -> libmx:mxCreateUninitDoubleMatrix(0x0, 0x2, 0x4, 0x0)
                 ...
  <- libmx:mxCreateUninitDoubleMatrix() = 0x1b07a8
                 ...
  -> libmwlapack:dgemm_(0xfef07ecc, 0xfef07ecc, 0xffbef094, 0xffbef090)
                 ...
  <- libmwlapack:dgemm_() = 0xffbef08c
                 ...
  -> libmatlb:mlfCleanupFcn(0xffbef320, 0x2, 0x2, 0xffbef38c)
  <- libmatlb:mlfCleanupFcn() = 0xff32ced0
<- libmatlb:mclMtimes() = 0x1b07a8
                       ...
```

Fig. 2.3 Condensed Internal Call Trace of `mclMtimes`

making use of algebraic identities at the source level [70, 69]. While the former can be easily done without type information, the latter may require it to some extent.

For example, consider the replacement of `A'*q` in MATLAB by the numerically equivalent `(q'*A)'`. This is a profitable replacement only when `A` is an $n \times n$ matrix and `q` is an $n \times 1$ vector because only vectors need to be transposed in `(q'*A)'` as opposed to a matrix [70]. If `A` and `q` were an $n \times 1$ vector and $n \times n$ matrix instead, we are better off leaving `A'*q` as is. This is therefore a source-to-source transformation whose application is dependent on the availability of type information. Other source-to-source transformations that may need type information are vectorization and preallocation [70, 69]. It should be mentioned that not every algebraic identity will qualify for consideration as a source-to-source transformation simply because it leads to increased efficiency. This is because to maintain code correctness, numerical equivalency will also have to be preserved across transformations. Thus, while the authors in [70, 69] mention associativity as another algebraic property that could be leveraged in a source-to-source transformation, its exploitation is questionable since associativity isn't assured to hold under finite precision arithmetic.

An early framework on source-to-source transformations was a process called "drag-along" [1]. The idea in drag-along was to defer the evaluation of an expression as long as possible, with the goal of building up a context for optimization and simplification. Among the most common gains of this approach was the detection of code sequences that could share the same evaluation loops and array temporaries.

2.2.3 Specialized Libraries and Parallelization

Packages such as BLAS [35], LAPACK [8] and ATLAS [97] aim at improving the overall FLOPS rating (floating-point operations per second) by using blocking algorithms that attempt full reuse of data in the lower levels of the memory hierarchy. This ameliorates the so-called surface-to-volume effect [34]. Since a higher FLOPS rating means a lower per-element time, routines from such packages are ideal for realizing an array operation's expression evaluation phase. Indeed, MATLAB itself banks on some of these libraries to implement the expression evaluation phase of certain operations, an example being matrix multiplication that uses the `dgemm` routine of the Level 3 BLAS library (see Figure 2.3).[3]

The relevance of parallelization to speeding up typeless array-based languages actually goes beyond just overcoming the interpretive bottleneck. Unlike most "sequential" languages such as FORTRAN 77 and C that require extensive static analyses to determine program parts that could be executed in parallel, the semantics of

[3]On some platforms, like Solaris, the FORTRAN compiler appends an underscore to the names of subprograms. To access these interfaces from a C program, it is therefore necessary to affix an underscore to their names. Hence the underscore suffix to the `dgemm` name in Figure 2.3.

languages like APL, MATLAB and IDL are replete with explicit parallelism, especially of the data-parallel kind. Ironically, their "typelessness" has indirectly made parallelization an even more viable proposition for them. This is a fallout of the idiomatic forms that are advocated when writing code in them. The basic theme is one of eschewing program loops, and working on as much data as possible in individual operations. This espousal stems from interpretation being the customary vehicle of execution for programs written in these languages, primarily because they are so high-level that their compilation is often a difficult affair. Moreover, code development using these languages normally happens in an interactive setting for which interpretation is an apposite choice. This impact of a language's execution support on its programming style and evolution is aptly captured in the words of Adin D. Falkoff [39, page 11], one of APL's principal contributors:

> I think it is inevitable that compilers will force changes in the way the language is used. We already know that people respond to the idiosyncrasies of particular interpreters. If you are using an interpreter that you know runs fast with a particular primitive, or manages booleans poorly, or whatever the case may be, there is a natural tendency to take these things into account in your programming. I think this is inevitable, and we kid ourselves if we believe that compilers won't have some effect on how we think about our problems.

It is in this respect that MATLAB and its ilk are perhaps more amenable to parallelization than a language like FORTRAN 90 because though array-based, FORTRAN 90 is still closely tied to the underlying architecture, so that the writing of vector-style code isn't critical to its efficient compilation onto a sequential machine. Thus parallelism may not be as explicit in a FORTRAN 90 source as in an equivalent but sequentially efficient (in regards to interpretation) MATLAB or APL version.

2.3 Automatic Array Shape Determination

Early work in array shape determination was in the context of the more general problem of automatic type inference. Widely employed methodologies to 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 [44, 72], or by using lattice-theoretic fixed-point techniques [89, 58]. 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 [22, 17, 32, 7]. Though type estimation techniques founded on iterative data-flow 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, approaches grounded in lattice theory rely on partial orders that are also *subtyping* relations. A subtyping relation is a quasiorder[4] that captures the principle of *safe substitution* [74]: If s and t are two types such that $s \precsim t$ where $\precsim$ is a subtyping relation, then for every value of type s, there exists a value of type t that is a *context-insensitive* safe replacement for it. Implicit in this definition is that a type is a set of elements or *values*.[5] "Context-insensitive safeness" means that independent of program context, these replacements strictly preserve program correctness, i.e., ill-defined programs remain ill-defined, and well-defined programs retain their meanings. This makes lattices whose partial orders are also a subtyping very effective for problems such as *intrinsic type* determination (finding whether a program variable is an integer, a real, a complex number and so on) where there is a clear, context-insensitive notion of safeness, and they have been successfully used for this, both in MATLAB [32, 51, 7] and APL [22, 17].

2.3.1 Limitations of Lattice-Based Approaches

Presuming for the moment that programs are always shape correct, can a bounded lattice be defined for MATLAB such that its greatest element denotes some specific shape and such that its partial order is also a subtyping? We define a *shape restriction* to be a proper subset of the set of *legal* shape tuples $\mathbb{L}_{\mathbb{S}}$, where each legal shape

[4] A quasiorder, also known as a preorder, is a binary relation that is reflexive and transitive.

[5] A subtyping needn't be a subset relation. An example is a subtyping on record types [74].

tuple $\langle p_1, p_2, \ldots, p_k \rangle$ denotes the shape of a k-dimensional array with the extent p_i along the ith dimension and where each p_i is an element in the set of nonnegative integers $\mathbb{W}$.[6] If such a lattice could be defined, then its greatest element would also be a safe, fallback inference for which code *specializable* to some degree (because it isn't the universal set of shapes) may be generated. Unfortunately, as Theorem 1 below will show, it is impossible to define such a lattice for MATLAB because of the way an operation like matrix multiplication is defined in the language.

Theorem 1 *It is impossible to define a bounded lattice for MATLAB on some nonempty subset of $2^{\mathbb{L}_\mathbb{S}}$, the union of whose members is $\mathbb{L}_\mathbb{S}$, such that its greatest element is a proper subset of $\mathbb{L}_\mathbb{S}$ and such that its partial order is also a subtyping.*

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

$$\bigcup_{j \geq 1} l_j = \mathbb{L}_\mathbb{S} \tag{2.1}$$

and let $\mathbf{1}$, its greatest element, be a proper subset of $\mathbb{L}_\mathbb{S}$. Then there exists a shape tuple $\langle z_1, z_2, \ldots, z_k \rangle$ such that

$$\langle z_1, z_2, \ldots, z_k \rangle \notin \mathbf{1}. \tag{2.2}$$

From Equation (2.1), there exists a lattice point l_Z such that

$$l_Z \leq \mathbf{1},$$
$$\langle z_1, z_2, \ldots, z_k \rangle \in l_Z.$$

Consider the MATLAB statement

$$\texttt{c} \leftarrow \texttt{a*b}$$

and let $\langle z_1, z_2, \ldots, z_k \rangle$ be the shape tuple of `a`. Thus, with respect to L, l_Z would denote the shape of `a`. Now the only replacements to $\langle z_1, z_2, \ldots, z_k \rangle$ that won't

[6]We choose $\mathbb{W}$ so as to include the empty array construct.

violate the shape requirements of the above operation are

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

(the last one if $k = 2$ or $z_3 = z_4 = \cdots = z_k = 1$) where $o \in \mathbb{W}$. Similarly, if $\langle z_1, z_2, \ldots, z_k \rangle$ is the shape tuple of `b`, then its only valid replacements are

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

(the last one if $k = 2$ or $z_3 = z_4 = \cdots = z_k = 1$) where $r \in \mathbb{W}$. Therefore, the only shape tuples that can be used as *context-insensitive* substitutes for $\langle z_1, z_2, \ldots, z_k \rangle$ so as to comply with the shape semantics of a matrix-multiplication operation are

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

However, substituting $\langle z_1, z_2, \ldots, z_k \rangle$ by $\langle 1, 1, \ldots, 1 \rangle$ won't always preserve the meaning of a well-defined MATLAB program. For instance, if the shapes of `a` and `b` were $\langle 2, 3 \rangle$ and $\langle 3, 4 \rangle$, `c` would be of shape $\langle 2, 4 \rangle$. This would have a different meaning than the one when the shapes of both `a` and `b` are set to $\langle 1, 1 \rangle$—in the latter case, `c` would be a different object, of shape $\langle 1, 1 \rangle$. Hence from Equation (2.2), there doesn't exist a context-insensitive safe replacement in $\mathbf{1}$ for $\langle z_1, z_2, \ldots, z_k \rangle$; thus the partial order isn't a subtyping. □

2.3.1.1 The Problem of Separate Compilation

A corollary to Theorem 1 is that the greatest element in any bounded lattice that is used to infer shapes in a shape-correct MATLAB program must be some abstraction

```
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. 2.4 Numerical Computation of $\int_a^b 13(x - x^2)e^{\frac{-3x}{2}}\,dx$ by Simpson's Rule

that represents $\mathbb{L}_{\mathbb{S}}$. This means that if the greatest element of such a lattice were returned as an inference, no new information would be obtained since the greatest element signifies the set of all possibilities. For example, if the lattice in [30, 32] were used to infer the shapes in the following code, excerpted from the adaptive quadrature program in the FALCON benchmark suite [30], its greatest element $\langle \mathsf{UNKNOWN}, \mathsf{UNKNOWN} \rangle$[7] would be returned as the shape of each variable defined in the procedure. 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 2.4 are *always* identical.

> Inference 6.
> For `quadrature` to be well defined, `a` and `b` must be square matrices, not necessarily of the same shape (this includes the case of them being scalars). Then the shapes of all the left-hand side variables will be identical.

A translator could use Inference 6 to collapse the shape conformance checks for the function to a single one situated at its entry, and to also do a Chaitin-style coalescing of the storages assigned to `Fa`, `Fmid`, `Fb` and `S` (see Chapter 9).

[7] Note that the lattice defined in [30, 32] is at the level of array extents; the Cartesian product of this lattice with itself gives the shape lattice effectively used.

```
1   function unsafe()

2   x ← rand(3, 2);
3   a ← rand;

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

Fig. 2.5 Eliciting an Unsafe Shape Solution from MaJIC

2.3.1.2 Unsafe Shape Solutions in MaJIC

Nearly every previous lattice used for deducing shapes in MATLAB and APL has adhered to the requirements of Theorem 1.[8] 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 [6, 7], where $\bot_s = \langle 0, 0\rangle$ and $\top_s = \langle \infty, \infty \rangle$ were the least and greatest elements in L_s, and where the lattice's partial order $\sqsubseteq_s$ was

$$\langle a, b\rangle \sqsubseteq_s \langle c, d\rangle \qquad \text{iff } a \leq c \text{ and } b \leq d \text{ for all } a, b, c, d \in \mathbb{W}. \tag{2.3}$$

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 an unsafe solution is shown in Figure 2.5. The function `unsafe` incrementally grows `a` within a `while` loop into an indefinite-sized row vector. 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 [6, page 33]. However, $\top_s$ won't always be a safe solution for `a` because it may not conform to the rules of the matrix-multiplication operation on the last line, even though the `while` loop test p

[8] Theorem 1 can also be proved for APL by considering the inner tensor product operation.

may be such that the loop either doesn't iterate or iterates exactly thrice on every execution run, thereby ensuring that the program is always well defined.[9]

2.3.2 Shape-Aware Array Language Designs

There has also been work in the bottom-up design of tacitly typed array languages that permit a complete static analysis of shape. An example is the FISh language [47], which achieves this by the imposition of shape constraints such as not allowing an array's shape to change during execution. These kind of restrictions 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 [47].

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

Chapter 3

Selective Rank Demotion

...the source of all great mathematics is the special case, the concrete example. It is frequent in mathematics that every instance of a concept of seemingly great generality is in essence the same as a small and concrete special case.
— PAUL R. HALMOS
"I Want to be a Mathematician: An Automathography,"
Springer-Verlag New York, Inc., 1985. Page 324.

Logically, all arrays in MATLAB have at least two dimensions. For example, a scalar is a 1×1 matrix and a column vector is an $x \times 1$ matrix. A pivotal decision that went into MATLAB's design, whether intentional or not, was to completely ignore trailing array 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 were of shape $\langle 2, 3, 1, 2\rangle$. This chapter discusses this selective demotion of an array's rank, and shows how it enables a simpler shape calculus for MATLAB by allowing the ranks of all arrays to be set to a common, fixed value.

3.1 A Selective Rank Demotion Subtyping

Selective rank demotion is the property by which trailing extents of unity beyond some dimension can be discarded from an array's shape without affecting semantics.

This permits all array processing to be done either by first demoting ranks by expunging dimensions, or by first promoting ranks by inserting dimensions.

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

$$\mathbb{L}_{\mathbb{S}} = \{\langle w_1, w_2, \ldots, w_m \rangle \mid m \geq 2 \wedge w_i \in \mathbb{W} \text{ for all } 1 \leq i \leq m\}, \tag{3.1}$$

selective rank demotion in MATLAB is in essence a subtyping $\wp$ on $\mathbb{L}_{\mathbb{S}}$ such that 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 $\wp$ iff they at most differ by trailing extents of unity beyond the second dimension:

$$\boldsymbol{s} \wp \boldsymbol{t} \iff \boldsymbol{s} = \boldsymbol{t} \vee \boldsymbol{s} = \langle q_1, q_2, \ldots, q_l, 1, \ldots, 1 \rangle \vee \boldsymbol{t} = \langle p_1, p_2, \ldots, p_k, 1, \ldots, 1 \rangle. \tag{3.2}$$

3.2 Canonical Shape Tuples and Ranks

Clearly, $\wp$ is also an equivalence relation on $\mathbb{L}_{\mathbb{S}}$. If $\overline{\boldsymbol{s}}$ is the equivalence class under $\wp$ of the shape tuple $\boldsymbol{s}$ of a MATLAB expression e, then any element in $\overline{\boldsymbol{s}}$ would be an equally valid denotation of e's shape. Therefore, the term *shape* in this thesis will formally mean an equivalence class, or *shape-tuple class*, under $\wp$. The concept of equivalent shape tuples leads to the ideas 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.[1]

3.2.1 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 $\mathbb{L}_{\mathbb{S}}$. A suitable choice for $\mathbb{I}_{\mathbb{S}}$ that doesn't clash with any subset of $\mathbb{L}_{\mathbb{S}}$ and that also forms an equivalence class under $\wp$ would be

$$\mathbb{I}_{\mathbb{S}} = \{\langle \pi_1, \pi_2 \rangle, \langle \pi_1, \pi_2, 1 \rangle, \langle \pi_1, \pi_2, 1, 1 \rangle, \ldots\}. \tag{3.3}$$

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

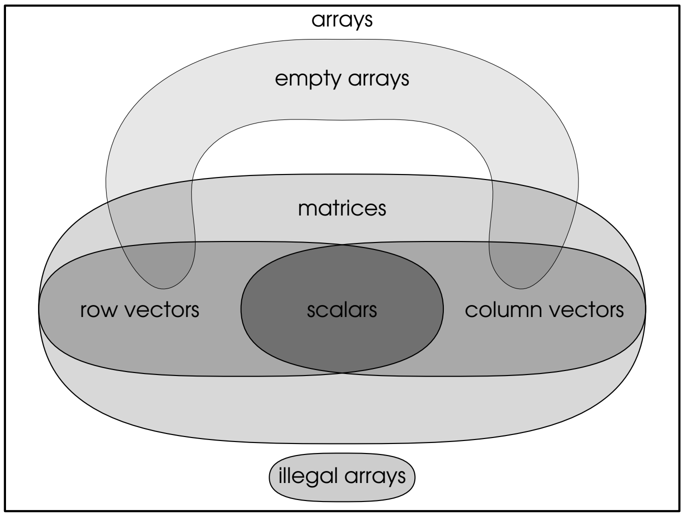

Fig. 3.1 The Set of MATLAB Arrays $\mathbb{A}$

π_1 and π_2 in the above are integers such that either $\pi_1 < 0$ 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 thesis. The *quotient set* of $\mathbb{S}$ by $\wp$ [93], which is the set of all shape-tuple classes under $\wp$, will be denoted by $\mathbb{S}_{\wp}$.

3.2.2 Terminology

When describing array structure in MATLAB, 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 2.
>
> **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.

The arrays covered by these definitions are graphically shown in Figure 3.1. We shall use the phrase "higher dimensional array" when referring to legal arrays whose canonical ranks are at least 3. These arrays lie in the unshaded region of Figure 3.1. 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.

3.2.3 Empty Arrays

MATLAB, like APL and unlike IDL, also supports empty arrays. These are legal arrays that contain no data but that still have a shape. To quote [66]:

> 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 with a $\langle 0, 3\rangle$ matrix, the product would be a $\langle 2, 3\rangle$ matrix with all of its elements set to 0.

3.3 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 is 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 ω, where ω is an ordinal number larger than any finite integer,[2] without altering a program's meaning.

While selective rank demotion and its incorporation into an array language's operational semantics may seem obvious, it should be noted that it doesn't exist in languages like 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, both of which have meanings different from a scalar. In MATLAB, the three are indistinguishable. An example of this in FORTRAN 90 are the arrays defined below.

[2] ω can be considered to be the first transfinite number, the "smallest" infinity with this property.

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

The shapes of `A` and `B` from the above, 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.

Chapter 4

A Shape Algebraic System

What is it indeed that gives us the feeling of elegance in a solution, in a demonstration? It is the harmony of the diverse parts, their symmetry, their happy balance; in a word it is all that introduces order, all that gives unity, that permits us to see clearly and to comprehend at once both the ensemble and the details.
— JULES HENRI POINCARÉ
As quoted in "Mathematical Maxims and Minims," N. J. Rose and J. de Pillis, Rome Press, Inc., 1988. Page 34.

This chapter lays out a shape calculus for MATLAB that capitalizes on the property of selective rank demotion presented in Chapter 3. 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. 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 viewed as defining a one-to-one correspondence among shape-tuple classes. In this way, 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 could then be exploited by a code translator even when explicit information on array shapes is lacking at translation time.

4.1 Shape-Tuple Notation Duality

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

$$\langle p_1, p_2, \ldots, p_n\rangle = \begin{pmatrix} p_1 & 0 & \ldots & 0 \\ 0 & p_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & p_n \end{pmatrix}. \tag{4.1}$$

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

The notation $\langle p_1, p_2, \ldots, p_n\rangle^{\boldsymbol{\cdot}}$ will be used to mean the shape tuple equivalent to $\langle p_1, p_2, \ldots, p_n\rangle$ by the subtyping $\wp$ in Equation (3.2), but having ω components:

$$\langle p_1, p_2, \ldots, p_n\rangle^{\boldsymbol{\cdot}} = \begin{pmatrix} p_1 & 0 & \ldots & 0 & 0 & \ldots \\ 0 & p_2 & \ldots & 0 & 0 & \ldots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \\ 0 & 0 & \ldots & p_n & 0 & \ldots \\ 0 & 0 & \ldots & 0 & 1 & \ldots \\ \vdots & \vdots & & \vdots & \vdots & \ddots \end{pmatrix}. \tag{4.2}$$

From Equation (4.2), it is clear that $\boldsymbol{s}^{\boldsymbol{\cdot\cdot}} = \boldsymbol{s}^{\boldsymbol{\cdot}}$ for any shape tuple $\boldsymbol{s}$.

4.2 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 § 3.2.2 for terminology) for the operation to be legal [66]. We say that the shape semantics in this case is conditioned on the operand 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

$$P_S(\boldsymbol{s}) = \begin{cases} 1 & \text{if } \overline{\boldsymbol{s}} \in S, \\ 0 & \text{if } \overline{\boldsymbol{s}} \notin S. \end{cases} \tag{4.3}$$

4.2.1 An Extended Discrete Delta Function

All SS Predicates can be uniformly expressed by using an extension of the Discrete Delta Function δ to square matrices. That is, 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 $\boldsymbol{0}$ as follows:[1]

$$\delta(\boldsymbol{s}) = \begin{cases} \boldsymbol{I} & \text{if } \boldsymbol{s} = \boldsymbol{0}, \\ \boldsymbol{0} & \text{if } \boldsymbol{s} \neq \boldsymbol{0}. \end{cases} \tag{4.4}$$

Hence, any SS Predicate P_S can be formulated as

$$P_S(\boldsymbol{s}) = |\boldsymbol{\iota}^{\boldsymbol{\cdot}} - \prod_{l} (\boldsymbol{\iota}^{\boldsymbol{\cdot}} - \delta(\boldsymbol{s}^{\boldsymbol{\cdot}} - \boldsymbol{t_l}^{\boldsymbol{\cdot}}))|, \tag{4.5}$$

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

4.2.2 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 predicate α that flags a scalar shape, the *legal* SS predicate[2] θ that

[1] This extension differs slightly from the one in [56, 50].

[2] Referred to as the *correctness shape predicate* in [56, 50].

indicates a legal shape, and the *matrix* SS predicate β that flags a matrix shape:

$$\alpha(\boldsymbol{s}) = P_{\{\overline{\boldsymbol{\iota}}\}}(\boldsymbol{s}) = |\delta(\boldsymbol{s}^{\bullet} - \boldsymbol{\iota}^{\bullet})|, \tag{4.6}$$

$$\theta(\boldsymbol{s}) = P_{\{\overline{\boldsymbol{t}} \,\mid\, \overline{\boldsymbol{t}} \neq \overline{\boldsymbol{\pi}}\}}(\boldsymbol{s}) = |\boldsymbol{\iota}^{\bullet} - \delta(\boldsymbol{s}^{\bullet} - \boldsymbol{\pi}^{\bullet})|, \tag{4.7}$$

$$\beta(\boldsymbol{s}) = P_{\{\overline{\boldsymbol{t}} \,\mid\, \overline{\boldsymbol{t}} \neq \overline{\boldsymbol{\pi}} \wedge \boldsymbol{t}^{\bullet}\boldsymbol{\epsilon}^{\bullet} = \boldsymbol{\epsilon}^{\bullet}\}}(\boldsymbol{s}) = \theta(\boldsymbol{s})|\delta(\boldsymbol{s}^{\bullet}\boldsymbol{\epsilon}^{\bullet} - \boldsymbol{\epsilon}^{\bullet})|. \tag{4.8}$$

In Equation (4.8), $\boldsymbol{\epsilon} = \langle 0, 0 \rangle$ is the shape tuple of a particular kind of empty matrix.

4.3 Shape-Tuple Equations

For certain functions in MATLAB that we call Type I (see Chapter 7), knowing the shapes of the operands suffices 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 ith operand ($1 \leq i \leq m$) to f, the following *shape-tuple equation* can be shown to always hold among them:

$$\boldsymbol{u}^{\bullet} = \overbrace{\big(1 - \theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})\big)\boldsymbol{\pi}^{\bullet}}^{\text{illegal shape-tuple combo}} + \underbrace{\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m}) H_f(\boldsymbol{s_1}^{\bullet}, \boldsymbol{s_2}^{\bullet}, \ldots, \boldsymbol{s_m}^{\bullet})}_{\text{legal shape-tuple combo}}. \tag{4.9}$$

To understand Equation (4.9), we begin by observing that $\boldsymbol{u}$ is always either a legal or an 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. And 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? For this case, we postulate that $\boldsymbol{u}$ is also an illegal shape tuple. Doing so ensures that in the denotational semantics, information pertaining to a shape error, once engendered, is propagated through the rest of the program from the point of origination. Thus, θ_f can be formally defined as follows:

$$\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m}) = \begin{cases} 1 & \text{if } \overline{\boldsymbol{s_i}} \neq \overline{\boldsymbol{\pi}} \text{ for all } 1 \leq i \leq m, \text{ and} \\ & \text{the } \boldsymbol{s_i}\text{s conform to the shape rules of } f, \\ 0 & \text{otherwise.} \end{cases} \tag{4.10}$$

The function $H_f : \mathbb{S}_\omega^m \mapsto \mathbb{S}_\omega$ describes how combinations of legal shapes that accord with the shape semantics of f map to a legal shape. In particular, $\mathbb{S}_\omega$ is the subset of all those shape tuples in $\mathbb{S}$ having ω components. This guarantees that the matrix arithmetic in Equation (4.9) is well defined. Since the effect of $H_f(\boldsymbol{s_1}^\bullet, \boldsymbol{s_2}^\bullet, \ldots, \boldsymbol{s_m}^\bullet)$ in Equation (4.9) comes through to the left-hand side only when $\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m}) = 1$, H_f has free reign on how to map combinations involving illegal shapes, or combinations of legal shapes that *don't* conform to the shape rules of f, because its contribution in those cases is suppressed by a zero-valued $\theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})$. This provides a degree of flexibility in the formulation of H_f.

4.4 Sample Shape-Tuple Equations

Therefore, to form Equation (4.9) for a specific f, only θ_f and H_f need to be ascertained. This can be done by couching the descriptions in [66] 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. These shape semantics can be translated into a θ_f and H_f as follows:

$$\theta_{'}(\boldsymbol{s}) = \beta(\boldsymbol{s}), \tag{4.11}$$

$$H_{'}(\boldsymbol{s}^\bullet) = \boldsymbol{\Psi}\boldsymbol{s}^\bullet\boldsymbol{\Psi}, \tag{4.12}$$

where $\mathbf{\Psi}$ stands for a particular $\omega \times \omega$ *elementary matrix* [12]:

$$\mathbf{\Psi} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 & 0 & \dots \\ 1 & 0 & 0 & \dots & 0 & 0 & \dots \\ 0 & 0 & 1 & \dots & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \\ 0 & 0 & 0 & \dots & 1 & 0 & \dots \\ 0 & 0 & 0 & \dots & 0 & 1 & \dots \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \ddots \end{pmatrix}. \tag{4.13}$$

Elementary matrices arise in connection with the elementary row and column operations in linear algebra [12]. A multiplication with these matrices results in either 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 $\mathbf{\Psi}$, 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 (4.12) is to exchange the first two principal diagonal elements of any $\omega \times \omega$ matrix $\boldsymbol{s}^{\boldsymbol{\cdot}}$.

As another illustration of how θ_f and H_f may be determined, consider

```
c ← a+b
```

where f is the array addition operation in MATLAB. The shape semantics of this built-in entail 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

$$\theta_+(\boldsymbol{s}, \boldsymbol{t}) = \theta(\boldsymbol{s})\theta(\boldsymbol{t})(\alpha(\boldsymbol{s}) + \alpha(\boldsymbol{t}) + |\delta(\boldsymbol{s}^{\boldsymbol{\cdot}} - \boldsymbol{t}^{\boldsymbol{\cdot}})| - 2\alpha(\boldsymbol{s})\alpha(\boldsymbol{t})), \tag{4.14}$$

$$H_+(\boldsymbol{s}^{\boldsymbol{\cdot}}, \boldsymbol{t}^{\boldsymbol{\cdot}}) = \alpha(\boldsymbol{s}^{\boldsymbol{\cdot}})\boldsymbol{t}^{\boldsymbol{\cdot}} + (1 - \alpha(\boldsymbol{s}^{\boldsymbol{\cdot}}))\boldsymbol{s}^{\boldsymbol{\cdot}}. \tag{4.15}$$

Notice that Equation (4.14) complies with Equation (4.10) and that Equation (4.15) 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 4.1 displays the θ_f and H_f functions for a few Type I built-ins f. For a discussion on how these were arrived at and a longer list, see [56]. The symbols $\boldsymbol{\epsilon_1}$ and $\boldsymbol{\epsilon_2}$ in the table stand for $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$ respectively.

4.5 Towards a Congruent Subtyping

Consider a closed binary operation $\bullet$ on some set S. An equivalence relation E on S is said to have the *Substitution Property* [93] with respect to $\bullet$ iff

$$(x\ E\ x') \wedge (y\ E\ y') \Longrightarrow (x \bullet y)\ E\ (x' \bullet y') \qquad \text{for all } x, y, x', y' \in S.$$

If the right-hand side of Equation (4.9) 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 2 *Consider two systems of solutions for Equation* (4.9)*:*

$$\boldsymbol{u}^{\boldsymbol{\cdot}} = \big(1 - \theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})\big)\boldsymbol{\pi}^{\boldsymbol{\cdot}} + \theta_f(\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots, \boldsymbol{s_m})H_f(\boldsymbol{s_1}^{\boldsymbol{\cdot}}, \boldsymbol{s_2}^{\boldsymbol{\cdot}}, \ldots, \boldsymbol{s_m}^{\boldsymbol{\cdot}}), \quad (4.16)$$

$$\boldsymbol{v}^{\boldsymbol{\cdot}} = \big(1 - \theta_f(\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_m})\big)\boldsymbol{\pi}^{\boldsymbol{\cdot}} + \theta_f(\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_m})H_f(\boldsymbol{t_1}^{\boldsymbol{\cdot}}, \boldsymbol{t_2}^{\boldsymbol{\cdot}}, \ldots, \boldsymbol{t_m}^{\boldsymbol{\cdot}}). \quad (4.17)$$

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 $1 \le i \le m$,

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

follows from Equation (4.10) and the fact that $\wp$ is a subtyping. Moreover, since $\boldsymbol{s_i} \wp \boldsymbol{t_i}$ implies $\boldsymbol{s_i}^{\boldsymbol{\cdot}} = \boldsymbol{t_i}^{\boldsymbol{\cdot}}$ by Equation (3.2), we have

$$H_f(\boldsymbol{s_1}^{\boldsymbol{\cdot}}, \boldsymbol{s_2}^{\boldsymbol{\cdot}}, \ldots, \boldsymbol{s_m}^{\boldsymbol{\cdot}}) = H_f(\boldsymbol{t_1}^{\boldsymbol{\cdot}}, \boldsymbol{t_2}^{\boldsymbol{\cdot}}, \ldots, \boldsymbol{t_m}^{\boldsymbol{\cdot}}).$$

Hence, Equations (4.16) and (4.17) yield $\boldsymbol{u}^{\boldsymbol{\cdot}} = \boldsymbol{v}^{\boldsymbol{\cdot}}$, thus implying $\boldsymbol{u} \wp \boldsymbol{v}$. □

Equivalence relations like $\wp$ that satisfy the Substitution Property with respect to some operation are called *congruences* [93]. Congruences are useful because they expose an algebraic structure among equivalence classes.

<table>
<tr><th>f</th><th>θ_f</th><th>H_f</th></tr>
<tr><td><code>a * b</code></td><td>$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t}))(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})\delta(\boldsymbol{s}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}}-\boldsymbol{\Psi}\boldsymbol{t}^{\cdot}\boldsymbol{\Psi}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}}))\big)$</td><td>$\alpha(\boldsymbol{t}^{\cdot})\boldsymbol{s}^{\cdot}+(1-\alpha(\boldsymbol{t}^{\cdot}))\big(\alpha(\boldsymbol{s}^{\cdot})\boldsymbol{t}^{\cdot}+(1-\alpha(\boldsymbol{s}^{\cdot}))(\boldsymbol{s}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{2}}+\boldsymbol{t}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}}-\boldsymbol{\iota}^{\cdot}\boldsymbol{\epsilon}^{\cdot})\big)$</td></tr>
<tr><td><code>a+b</code>
<code>a.*b</code>
<code>a.^b</code>
<code>a./b</code>
<code>a.\b</code>
<code>a<b</code>
<code>a|b</code></td><td>$\theta(\boldsymbol{s})\theta(\boldsymbol{t})(\alpha(\boldsymbol{s})+\alpha(\boldsymbol{t})+\lvert\delta(\boldsymbol{s}^{\cdot}-\boldsymbol{t}^{\cdot})\rvert-2\alpha(\boldsymbol{s})\alpha(\boldsymbol{t}))$</td><td>$\alpha(\boldsymbol{s}^{\cdot})\boldsymbol{t}^{\cdot}+(1-\alpha(\boldsymbol{s}^{\cdot}))\boldsymbol{s}^{\cdot}$</td></tr>
<tr><td><code>a^b</code></td><td>$\beta(\boldsymbol{s})\beta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s})\delta(\boldsymbol{\Psi}\boldsymbol{t}^{\cdot}\boldsymbol{\Psi}-\boldsymbol{t}^{\cdot}))(1-\alpha(\boldsymbol{t})\delta(\boldsymbol{\Psi}\boldsymbol{s}^{\cdot}\boldsymbol{\Psi}-\boldsymbol{s}^{\cdot}))\big)$</td><td>$\alpha(\boldsymbol{s}^{\cdot})\boldsymbol{t}^{\cdot}+(1-\alpha(\boldsymbol{s}^{\cdot}))\boldsymbol{s}^{\cdot}$</td></tr>
<tr><td><code>a'</code>
<code>a.'</code></td><td>$\beta(\boldsymbol{s})$</td><td>$\boldsymbol{\Psi}\boldsymbol{s}^{\cdot}\boldsymbol{\Psi}$</td></tr>
<tr><td><code>a/b</code></td><td>$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t}))(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})\delta(\boldsymbol{s}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}}-\boldsymbol{t}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}})\big)$</td><td>$\alpha(\boldsymbol{s}^{\cdot})\boldsymbol{t}^{\cdot}+(1-\beta(\boldsymbol{s}^{\cdot}))\boldsymbol{t}^{\cdot}+(1-\alpha(\boldsymbol{s}^{\cdot}))\beta(\boldsymbol{s}^{\cdot})(\boldsymbol{s}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{2}}+\boldsymbol{\Psi}\boldsymbol{t}^{\cdot}\boldsymbol{\Psi}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}}-\boldsymbol{\iota}^{\cdot}\boldsymbol{\epsilon}^{\cdot})$</td></tr>
<tr><td><code>a\b</code></td><td>$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\big(1-(1-\alpha(\boldsymbol{s}))(1-\alpha(\boldsymbol{t}))(1-\beta(\boldsymbol{s})\beta(\boldsymbol{t})\delta(\boldsymbol{s}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{2}}-\boldsymbol{t}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{2}})\big)$</td><td>$\alpha(\boldsymbol{t}^{\cdot})\boldsymbol{s}^{\cdot}+(1-\beta(\boldsymbol{t}^{\cdot}))\boldsymbol{s}^{\cdot}+(1-\alpha(\boldsymbol{s}^{\cdot}))\beta(\boldsymbol{s}^{\cdot})(\boldsymbol{\Psi}\boldsymbol{s}^{\cdot}\boldsymbol{\Psi}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{2}}+\boldsymbol{t}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}}-\boldsymbol{\iota}^{\cdot}\boldsymbol{\epsilon}^{\cdot})$</td></tr>
<tr><td><code>[a; b]</code></td><td>$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\delta(\boldsymbol{s}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}}-\boldsymbol{t}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}})$</td><td>$\boldsymbol{s}^{\cdot}+\boldsymbol{t}^{\cdot}(\boldsymbol{\iota}^{\cdot}-\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{1}})$</td></tr>
<tr><td><code>[a, b]</code></td><td>$\theta(\boldsymbol{s})\theta(\boldsymbol{t})\delta(\boldsymbol{s}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{2}}-\boldsymbol{t}^{\cdot}\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{2}})$</td><td>$\boldsymbol{s}^{\cdot}+\boldsymbol{t}^{\cdot}(\boldsymbol{\iota}^{\cdot}-\boldsymbol{\epsilon}^{\cdot}_{\boldsymbol{2}})$</td></tr>
<tr><td><code>~a</code>
<code>tan(a)</code>
<code>exp(a)</code>
<code>fix(a)</code>
<code>fft(a)</code></td><td>$\theta(\boldsymbol{s})$</td><td>$\boldsymbol{s}^{\cdot}$</td></tr>
</table>

Table 4.1 θ_f and H_f for Some Type I Language Operators in MATLAB

4.6 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

$$(\overline{\boldsymbol{s_1}}, \overline{\boldsymbol{s_2}}, \ldots, \overline{\boldsymbol{s_m}})\ \dot{f}\ \overline{\boldsymbol{u}} \qquad \begin{array}{l}\text{iff } \exists \boldsymbol{y} \in \overline{\boldsymbol{u}}, \text{ and for all } 1 \leq i \leq m, \exists \boldsymbol{x_i} \in \overline{\boldsymbol{s_i}} \\ \text{such that } \boldsymbol{y}^{\boldsymbol{\cdot}} = \ddot{f}(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_m}),\end{array} \tag{4.18}$$

where the function $\ddot{f} : \mathbb{S}^m \mapsto \mathbb{S}$ stands for the right-hand side of Equation (4.9). Since there always exists a $\boldsymbol{u}$ that honors Equation (4.9) 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 2, $\dot{f}$ must be *single-valued* since

$$\Big((z, \overline{\boldsymbol{u}}) \in \dot{f}\Big) \wedge \Big((z, \overline{\boldsymbol{v}}) \in \dot{f}\Big) \Longrightarrow \overline{\boldsymbol{u}} = \overline{\boldsymbol{v}} \qquad \text{for all } z \in \mathbb{S}_\wp^m.$$

Hence, $\dot{f}$ defines an m-ary *operation* on $\mathbb{S}_\wp$.[3] 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 [93]. 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.

4.7 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 θ_f and H_f functions. Lemma 1 shows an example.

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

[3] 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 [93].

PROOF. From Equations (4.9) and (4.18),

$$
\begin{aligned}
\overline{\boldsymbol{a}} = \overline{\boldsymbol{t}} \dot{+} \overline{\boldsymbol{u}} \quad &\Longleftrightarrow \quad \boldsymbol{a}^{\bullet} = \big(1 - \theta_{+}(\boldsymbol{t}, \boldsymbol{u})\big)\boldsymbol{\pi}^{\bullet} + \theta_{+}(\boldsymbol{t}, \boldsymbol{u}) H_{+}(\boldsymbol{t}^{\bullet}, \boldsymbol{u}^{\bullet}), && (4.19)\\
\overline{\boldsymbol{b}} = \overline{\boldsymbol{s}} \dot{+} \overline{\boldsymbol{a}} \quad &\Longleftrightarrow \quad \boldsymbol{b}^{\bullet} = \big(1 - \theta_{+}(\boldsymbol{s}, \boldsymbol{a})\big)\boldsymbol{\pi}^{\bullet} + \theta_{+}(\boldsymbol{s}, \boldsymbol{a}) H_{+}(\boldsymbol{s}^{\bullet}, \boldsymbol{a}^{\bullet}), && (4.20)\\
\overline{\boldsymbol{x}} = \overline{\boldsymbol{s}} \dot{+} \overline{\boldsymbol{t}} \quad &\Longleftrightarrow \quad \boldsymbol{x}^{\bullet} = \big(1 - \theta_{+}(\boldsymbol{s}, \boldsymbol{t})\big)\boldsymbol{\pi}^{\bullet} + \theta_{+}(\boldsymbol{s}, \boldsymbol{t}) H_{+}(\boldsymbol{s}^{\bullet}, \boldsymbol{t}^{\bullet}), && (4.21)\\
\overline{\boldsymbol{y}} = \overline{\boldsymbol{x}} \dot{+} \overline{\boldsymbol{u}} \quad &\Longleftrightarrow \quad \boldsymbol{y}^{\bullet} = \big(1 - \theta_{+}(\boldsymbol{x}, \boldsymbol{u})\big)\boldsymbol{\pi}^{\bullet} + \theta_{+}(\boldsymbol{x}, \boldsymbol{u}) H_{+}(\boldsymbol{x}^{\bullet}, \boldsymbol{u}^{\bullet}). && (4.22)
\end{aligned}
$$

Our objective is to show $(\overline{\boldsymbol{s}} \dot{+} \overline{\boldsymbol{t}}) \dot{+} \overline{\boldsymbol{u}} = \overline{\boldsymbol{s}} \dot{+} (\overline{\boldsymbol{t}} \dot{+} \overline{\boldsymbol{u}})$; we consider three cases for this:

- *At least one shape among $\overline{\boldsymbol{s}}$, $\overline{\boldsymbol{t}}$ and $\overline{\boldsymbol{u}}$ is illegal.*

  Suppose $\overline{\boldsymbol{s}} = \overline{\boldsymbol{\pi}}$. Then $\overline{\boldsymbol{b}} = \overline{\boldsymbol{x}} = \overline{\boldsymbol{\pi}}$ from Equations (4.20) and (4.21) so that $(\overline{\boldsymbol{s}} \dot{+} \overline{\boldsymbol{t}}) \dot{+} \overline{\boldsymbol{u}} = \overline{\boldsymbol{s}} \dot{+} (\overline{\boldsymbol{t}} \dot{+} \overline{\boldsymbol{u}})$ is trivially true. Similarly if either $\overline{\boldsymbol{t}}$ or $\overline{\boldsymbol{u}}$ are $\overline{\boldsymbol{\pi}}$.

- *At least one among the legal shapes $\overline{\boldsymbol{s}}$, $\overline{\boldsymbol{t}}$ and $\overline{\boldsymbol{u}}$ represents a scalar.*

  Suppose $\overline{\boldsymbol{s}} = \overline{\boldsymbol{\iota}}$. Then looking up the expression for θ_{+} in Table 4.1, we get

  $$
  \begin{aligned}
  \theta_{+}(\boldsymbol{s}, \boldsymbol{t}) &= 1,\\
  \theta_{+}(\boldsymbol{s}, \boldsymbol{a}) &= \theta(\boldsymbol{a}).
  \end{aligned}
  $$

  Also from Table 4.1,

  $$
  \begin{aligned}
  H_{+}(\boldsymbol{s}^{\bullet}, \boldsymbol{a}^{\bullet}) &= \boldsymbol{a}^{\bullet},\\
  H_{+}(\boldsymbol{s}^{\bullet}, \boldsymbol{t}^{\bullet}) &= \boldsymbol{t}^{\bullet}.
  \end{aligned}
  $$

  Hence, Equations (4.20) and (4.21) simplify to

  $$
  \begin{aligned}
  \boldsymbol{b}^{\bullet} &= (1 - \theta(\boldsymbol{a}))\boldsymbol{\pi}^{\bullet} + \theta(\boldsymbol{a})\boldsymbol{a}^{\bullet} = \boldsymbol{a}^{\bullet}, && (4.23)\\
  \boldsymbol{x}^{\bullet} &= H_{+}(\boldsymbol{s}^{\bullet}, \boldsymbol{t}^{\bullet}) = \boldsymbol{t}^{\bullet}. && (4.24)
  \end{aligned}
  $$

  By plugging Equations (4.19) and (4.24) respectively into Equations (4.23) and (4.22), we obtain

  $$
  \begin{aligned}
  \boldsymbol{b}^{\bullet} &= \big(1 - \theta_{+}(\boldsymbol{t}, \boldsymbol{u})\big)\boldsymbol{\pi}^{\bullet} + \theta_{+}(\boldsymbol{t}, \boldsymbol{u}) H_{+}(\boldsymbol{t}^{\bullet}, \boldsymbol{u}^{\bullet}),\\
  \boldsymbol{y}^{\bullet} &= \big(1 - \theta_{+}(\boldsymbol{t}, \boldsymbol{u})\big)\boldsymbol{\pi}^{\bullet} + \theta_{+}(\boldsymbol{t}, \boldsymbol{u}) H_{+}(\boldsymbol{t}^{\bullet}, \boldsymbol{u}^{\bullet}).
  \end{aligned}
  $$

  This implies $(\overline{\boldsymbol{s}} \dot{+} \overline{\boldsymbol{t}}) \dot{+} \overline{\boldsymbol{u}} = \overline{\boldsymbol{s}} \dot{+} (\overline{\boldsymbol{t}} \dot{+} \overline{\boldsymbol{u}})$. The same is true if either $\overline{\boldsymbol{t}}$ or $\overline{\boldsymbol{u}}$ are $\overline{\boldsymbol{\iota}}$.

- *None of the legal shapes $\overline{\boldsymbol{s}}$, $\overline{\boldsymbol{t}}$ and $\overline{\boldsymbol{u}}$ represents a scalar.*

  From Table 4.1, $\theta_{+}(\boldsymbol{s}^{\bullet}, \boldsymbol{t}^{\bullet}) = |\delta(\boldsymbol{s}^{\bullet} - \boldsymbol{t}^{\bullet})|$ and $H_{+}(\boldsymbol{s}^{\bullet}, \boldsymbol{t}^{\bullet}) = \boldsymbol{s}^{\bullet}$. Therefore,

  $$
  \boldsymbol{x}^{\bullet} = (1 - |\delta(\boldsymbol{s}^{\bullet} - \boldsymbol{t}^{\bullet})|)\boldsymbol{\pi}^{\bullet} + |\delta(\boldsymbol{s}^{\bullet} - \boldsymbol{t}^{\bullet})|\boldsymbol{s}^{\bullet}
  $$

from Equation (4.21). Hence

$$\begin{aligned}\alpha(\boldsymbol{x}^{\cdot}) &= (1 - |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})|)\alpha(\boldsymbol{\pi}^{\cdot}) + |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})|\alpha(\boldsymbol{s}^{\cdot}),\\ &= 0\end{aligned}$$

since by the assumption for this case, $\alpha(\boldsymbol{s}^{\cdot}) = 0$. Therefore, from Table 4.1,

$$\begin{aligned}\theta_{+}(\boldsymbol{x}, \boldsymbol{u}) &= \theta(\boldsymbol{x})|\delta(\boldsymbol{x}^{\cdot} - \boldsymbol{u}^{\cdot})|,\\ &= |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})\delta(\boldsymbol{x}^{\cdot} - \boldsymbol{u}^{\cdot})|,\\ &= |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})\delta\big((1 - |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})|)\boldsymbol{\pi}^{\cdot} + |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})|\boldsymbol{s}^{\cdot} - \boldsymbol{u}^{\cdot}\big)|,\\ &= |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})\delta\big((1 - |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})|)(\boldsymbol{\pi}^{\cdot} - \boldsymbol{u}^{\cdot}) + |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})|(\boldsymbol{t}^{\cdot} - \boldsymbol{u}^{\cdot})\big)|,\\ &= |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})\big((1 - |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})|)\delta(\boldsymbol{\pi}^{\cdot} - \boldsymbol{u}^{\cdot}) + |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})|\delta(\boldsymbol{t}^{\cdot} - \boldsymbol{u}^{\cdot})\big)|,\\ &= |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})\delta(\boldsymbol{t}^{\cdot} - \boldsymbol{u}^{\cdot})|. \end{aligned} \tag{4.25}$$

Substituting Equation (4.25) into Equation (4.22) produces

$$\boldsymbol{y}^{\cdot} = (1 - |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})\delta(\boldsymbol{t}^{\cdot} - \boldsymbol{u}^{\cdot})|)\boldsymbol{\pi}^{\cdot} + |\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})\delta(\boldsymbol{t}^{\cdot} - \boldsymbol{u}^{\cdot})|\boldsymbol{u}^{\cdot}. \tag{4.26}$$

We can similarly show that

$$\boldsymbol{b}^{\cdot} = (1 - |\delta(\boldsymbol{t}^{\cdot} - \boldsymbol{u}^{\cdot})\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{u}^{\cdot})|)\boldsymbol{\pi}^{\cdot} + |\delta(\boldsymbol{t}^{\cdot} - \boldsymbol{u}^{\cdot})\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{u}^{\cdot})|\boldsymbol{u}^{\cdot}. \tag{4.27}$$

But $|\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{t}^{\cdot})\delta(\boldsymbol{t}^{\cdot} - \boldsymbol{u}^{\cdot})| = |\delta(\boldsymbol{t}^{\cdot} - \boldsymbol{u}^{\cdot})\delta(\boldsymbol{s}^{\cdot} - \boldsymbol{u}^{\cdot})|$. Hence from Equations (4.26) and (4.27), $\boldsymbol{y}^{\cdot} = \boldsymbol{b}^{\cdot}$. Thus, $(\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}})\dot{+}\overline{\boldsymbol{u}} = \overline{\boldsymbol{s}}\dot{+}(\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{u}})$ is true in this case too.

Since the above three cases cover all possibilities, the claim of the lemma follows. □

A number of other identities similar in spirit to Lemma 1 can be characterized. Some of these are listed in Table 4.2. (For their proofs, see [56, 53].) As Table 4.2 shows, even though $\dot{*}$ isn't a fully associative operator,[4] 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. Some of these are also shown in Table 4.2 and proved in [53]. We adopt the convention that $\dot{+}$, $\dot{*}$ and $\dot{-}$ have increasing precedence; this will make the use of parentheses unnecessary whenever operators can be bound according to precedence.

[4] 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 $\overline{\langle 1,2\rangle}\dot{*}(\overline{\langle 2,1\rangle}\dot{*}\overline{\langle 3,2\rangle}) = \overline{\boldsymbol{\pi}}$.

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

Heterogenous Identities
$\overline{\boldsymbol{s}}\dot{+}(\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}) = \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}$
$(\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}})\dot{*}(\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}) = \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$
$(\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}})\dot{*}(\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}) = \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$
$(\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}})\dot{*}\overline{\boldsymbol{t}} = \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$
$\overline{\boldsymbol{s}}\dot{*}(\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}) = \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}$
$\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}} = \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}$
$\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}} = \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$
$(\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}})\dot{*}(\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{s}}) = \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$
$(\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}})\dot{*}\overline{\boldsymbol{t}} = \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$
$\overline{\boldsymbol{s}}\dot{*}(\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}) = \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}$
$\dot{\neg}(\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}}) = \dot{\neg}\,\overline{\boldsymbol{t}}\dot{*}\dot{\neg}\,\overline{\boldsymbol{s}}$
$\overline{\boldsymbol{s}}\dot{+}\dot{\neg}\,\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}$

Table 4.2 Sample Shape Identities for Type I MATLAB Functions

CHAPTER 5

A Symbolic Shape Evaluator

The practice of first developing a clear and precise definition of a process without regard to efficiency, and then using it as a guide and a test in exploring equivalent processes possessing other characteristics, such as greater efficiency, is very common in mathematics. It is a very fruitful practice which should not be blighted by premature emphasis in computer execution.
— KENNETH E. IVERSON
"Notation as a Tool of Thought,"
Communications of the ACM, Vol. 23, No. 8, August 1980. Page 464.

This chapter presents a symbolic evaluator called MULTI-PASS-EVAL that automates the shape calculus described in Chapter 4. MULTI-PASS-EVAL takes in a tree representation of a shape expression and returns its symbolically evaluated version. The kernel in MULTI-PASS-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 commutativity and associativity, these characterizations are encoded as rewriting rules and are brought to bear by a rewriting process. Commutativity and associativity are handled specially by a pair of flattening and "unflattening" structural transformations called SQUASH and UNSQUASH.

Sufficient conditions under which MULTI-PASS-EVAL is strongly and uniquely normalizing are later presented. The chapter concludes by showing the running time of MULTI-PASS-EVAL to be $O(R\|s\|^3 \log \|s\|)$, where $\|s\|$ is the number of nodes in the shape expression tree s, and R is an upper bound on canonical ranks.

5.1 Shape Terms

Given the set Λ^* of syntactic representations of elements in $\mathbb{S}_\wp$ and a set $\mathcal{S}$ of "shape variable" symbols, we define the set Λ of *well-formed* rooted ordered trees over $\mathcal{S}$ and Λ^* as the smallest set containing $\mathcal{S}$ and Λ^* such that $\dot{f}_n(t_1, t_2, \ldots, t_n)$ is in Λ whenever $\dot{f}_n$ is an n-ary shape operator in $\dot{\mathcal{F}}$, and t_i is in Λ 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*—the set $\widehat{\mathcal{S}}$ of shape variables that are used to denote *explicit* shape terms (elements in Λ^*), 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 chapter is to symbolically evaluate elements in Λ 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 modelled 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 will be specifically depicted as $l \to r$ [33]. As we shall see later, it isn't necessary that all the properties of $[\mathbb{S}_\wp, \dot{\mathcal{F}}]$ be known; the more we know, the greater the potential for a "better" inference, i.e., an inference that conveys more specialized information. Greater the specificity of an inference, greater is the degree of tailoring possible in a compiler's or interpretor's generated code.

5.2 Rewriting with Rules

We say that a rule $l \to r$ rewrites a shape term s if there exists a *substitution* [83] of shape variables to shape terms that causes l to *match* s. Though a substitution is defined as a mapping ξ from $\mathcal{S}$ to Λ, it is used after being homomorphically extended to a mapping from Λ to Λ. This extension is done in such a way so that

$$\xi(\dot{f}_n(t_1, t_2, \ldots, t_n)) = \dot{f}_n(\xi(t_1), \xi(t_2), \ldots, \xi(t_n)) \tag{5.1}$$

holds for every shape operator $\dot{f}_n$, where each t_i is any shape term. The matching of l to s is a type of unification [62]—if l matches s, then there exists a substitution ξ such that $\xi(l) = s$.[1] In the event that $l \to r$ does rewrite s, the rewritten shape term is $\xi(r)$ where ξ is the matching substitution used. The particular variant of unification considered in this thesis is a many-sorted one [94] according to which different sorts of variables in $\mathcal{S}$ unify only with terms in Λ 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 Λ^*, while a symbolic shape variable $\widetilde{y}$ in $\widetilde{\mathcal{S}}$ unifies only with terms in $\Lambda - \Lambda^*$. An unqualified shape variable z is allowed to unify with any term in Λ.

5.2.1 Uniqueness of Matching Substitutions

Let $\mathcal{V}(s)$ stand for the set of all variables in a shape term s and let Ξ be the set of all possible *valid*[2] substitutions $\xi : \mathcal{S} \mapsto \Lambda$. Then, as Lemma 2 below will show, if ξ' and ξ'' are two mappings in Ξ that match s to a target shape term, the *restrictions* of ξ' and ξ'' 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' [93]; this is usually designated as g/X'.

Lemma 2 *If ξ' and ξ'' are substitutions in Ξ 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

[1] 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 individual function symbols.

[2] Every valid substitution is only allowed to map variables to their corresponding sorts.

inductive step, we shall prove that if the claim holds for a group of s_i $(1 \leq i \leq n)$ in Λ, 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 (5.1),

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

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

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

But from the induction hypothesis, $\xi'/\mathcal{V}(s_i) = \xi''/\mathcal{V}(s_i)$. Hence,

$$\bigcup_{i=1}^{n} \xi'/\mathcal{V}(s_i) = \bigcup_{i=1}^{n} \xi''/\mathcal{V}(s_i). \tag{5.2}$$

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 (5.2) can be rewritten as

$$\xi'/\bigcup_{i=1}^{n} \mathcal{V}(s_i) = \xi''/\bigcup_{i=1}^{n} \mathcal{V}(s_i). \tag{5.3}$$

But $\cup_{i=1}^{n}\mathcal{V}(s_i) = \mathcal{V}(\dot{f}_n(s_1, s_2, \ldots, s_n))$. Therefore, Equation (5.3) gives us

$$\xi'/\mathcal{V}(\dot{f}_n(s_1, s_2, \ldots, s_n)) = \xi''/\mathcal{V}(\dot{f}_n(s_1, s_2, \ldots, s_n)), \tag{5.4}$$

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

5.2.2 Connecting the Syntactic and Semantic Domains

Every element in $\mathbb{S}_\wp$ has at least one syntactic representation in Λ, namely a representation in Λ^*. In fact, there could be an infinite number of representations of a shape-tuple class $\overline{\boldsymbol{s}}$ in Λ^*.[3] An injective mapping Ψ from $\mathbb{S}_\wp$ to Λ^* can therefore be defined such that $\Psi(\overline{\boldsymbol{s}})$ is the syntactic denotation of $\overline{\boldsymbol{s}}$. 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 $\overline{\boldsymbol{s}}$.

[3] 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.

Similarly, a surjective function ψ could be defined that maps every *ground* shape term to its unique meaning in $\mathbb{S}_\wp$. Ground shape terms, which form a subset Λ_g of Λ, 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 (5.5) below 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 Λ^*. Hence, what the rule in Equation (5.5) indicates is how a shape term depicting an operation on explicit shape terms could be rewritten to an explicit shape term. (Note that $\dot{f}_n$ is used syntactically on the left-hand side and semantically on the right-hand side.)

$$\dot{f}_n(\widehat{x_1}, \widehat{x_2}, \ldots, \widehat{x_n}) \to \Psi\big(\dot{f}_n(\psi(\widehat{x_1}), \psi(\widehat{x_2}), \ldots, \psi(\widehat{x_n}))\big). \tag{5.5}$$

5.3 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:

$$\mathcal{A}(s, (l_1 \to r_1, l_2 \to r_2, \ldots, l_k \to r_k)) = \begin{cases} \xi(r_1) & \text{if } \exists \xi \in \Xi \text{ such that } \xi(l_1) = s, \\ s & \text{else if } k = 1, \\ \mathcal{A}(s, (l_2 \to r_2, \ldots, l_k \to r_k)) & \text{otherwise.} \end{cases} \tag{5.6}$$

The rewriting that occurs in Equation (5.6) diverges from traditional rewriting processes, such as that in [33], in two respects: (1) the rules applicator disregards the rewriting of subterms, limiting itself to the rewriting of 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 § 5.5.2). Note that while the rewrite rule choice in Equation (5.6) is deterministic, the outcome of the rewrite is nonetheless nondeterministic because the particular matching substitution ξ used is unspecified. This is why $\mathcal{A}$, though a total relation from $\Lambda \times \mathcal{R}$ to Λ, isn't a function.

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

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

In the first case, the rule used is $z\dot{+}z \to z$, the substitution being $\{z \hookrightarrow x\dot{+}x\}$. (Conventionally, substitutions are written explicitly as a set of bindings of only those variables that aren't mapped to themselves.) In the second case, no rule matches $x\dot{+}(y\dot{+}y)$. In the third, the matching substitution used is $\{\widehat{u} \hookrightarrow \overline{\langle 2,3\rangle}, \widehat{v} \hookrightarrow \overline{\langle 3,4\rangle}\}$.

5.4 Commutativity and Associativity

The properties of commutativity and associativity are treated specially in our approach and aren't modelled 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 § 5.5). This was done so as to achieve a terminating evaluator, which may not otherwise be possible if a rule like $x\dot{+}y \to y\dot{+}x$ were 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.[4]) In our approach, commutativity is handled by sorting shape terms according to a total order $\sqsubseteq$ on the elements in Λ, and associativity is managed by a pair of transformations called SQUASH and UNSQUASH that essentially flatten and "unflatten" shape terms.

[4] In traditional rewriting, ad hoc extensions are usually used to cope with the associativity and commutativity axioms; the result is often termed *AC rewriting* [33].

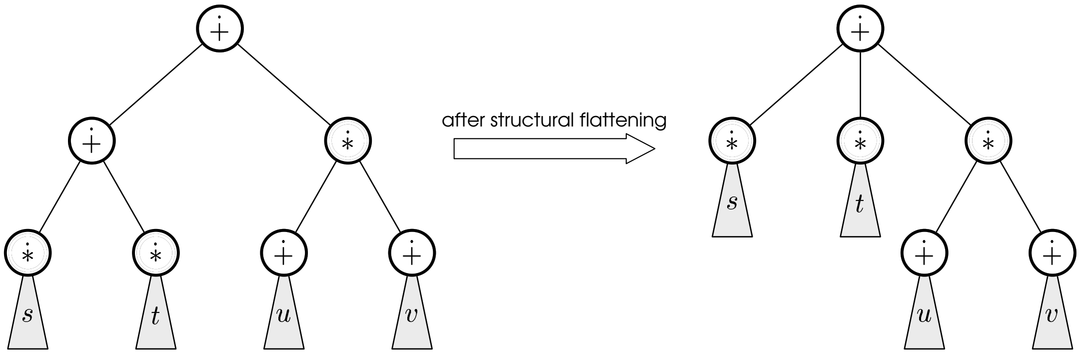

Fig. 5.1 Structural Flattening of a Shape Term's Rooted Ordered Tree

5.4.1 Squashing Shape Terms

SQUASH is motivated by the observation that parentheses in an associative operation are irrelevant. Its workings on $\dot{f}_n(s_1, s_2, \ldots, s_n)$ could be perceived as 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. For instance,

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

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 5.1, such a structure may not be well-formed under the syntactical rules of Λ. (The flattened structure in Figure 5.1 isn't well-formed because $\dot{+}$ isn't a ternary operator.)

Pseudo code given in Figure 5.2 provides the algorithmic definition of SQUASH. Lines prefixed by the ☛ character are comments, and procedures like ***root*** denote primitives that are documented in Table 5.1 on page 59. 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

SQUASH(s)
if *root*(s) $= s$
    ☛ This means $s \in \Lambda^* \cup \mathcal{S}$.
    return (s)
☛ Beyond this point, s is of
☛ the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$.
$\dot{f}_n \leftarrow$ *root*(s)
$n \leftarrow$ *arity*($\dot{f}_n$)
$tuple \leftarrow$ *children*(s)
for $i \leftarrow 1$ **to** n **do**
    $s_i \leftarrow$ *component*($tuple, i$)
    if *root*(s_i) $= \dot{f}_n$
        $tuple_i \leftarrow$ SQUASH(s_i)
    else
        $tuple_i \leftarrow (s_i)$
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 $1 \leq j \leq k$.
$k \leftarrow$ *length*($tuple$)
$n \leftarrow$ *arity*($\dot{f}_n$)
if $k < n$
    return $\perp$
else if $k = n$
    $t \leftarrow$ *build*($\dot{f}_n, tuple$)
    ☛ t is of the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$.
    return t
else if $n > 1$
    $t \leftarrow$ *build*($\dot{f}_n$, *take*($tuple, n$))
    ☛ t is of the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$.
    $tuple' \leftarrow$ *concat*((t), *drop*($tuple, n$))
    return UNSQUASH($\dot{f}_n, tuple'$)
else
    return $\perp$

Fig. 5.2 The SQUASH and UNSQUASH Procedures root's children to *tuple*. As indicated in Table 5.1, 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 Λ to $\mathcal{T}$. Lemma 3 below shows the connection between the arity of a shape term's root and the length of its tuple image under SQUASH.

Lemma 3 *If $s = \dot{f}_n(s_1, s_2, \ldots, s_n)$, then*

$$\textit{length}(\text{SQUASH}(s)) = n + m(n-1) \tag{5.7}$$

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

PROOF. We prove this claim using structural induction on the immediate nesting of $\dot{f}_n$. The base case occurs when none of the s_i have $\dot{f}_n$ as their root. The claim then trivially holds since $\text{SQUASH}(\dot{f}_n(s_1, s_2, \ldots, s_n))$ returns a tuple of length n. Therefore, let $k > 0$ be the number of s_i in $\dot{f}_n(s_1, s_2, \ldots, s_n)$ that have $\dot{f}_n$ as their root. For such s_i, $\text{SQUASH}(s_i)$ will return a tuple of length $n + m_i(n-1)$ by the induction hypothesis. From Lines 9, 10 and 11 of the SQUASH algorithm, the length of the tuple returned by $\text{SQUASH}(\dot{f}_n(s_1, s_2, \ldots, s_n))$ would then be

$$n - k + \sum_{i=1}^{i=k} (n + m_i(n-1)).$$

The above simplifies to $n + (n-1)(k + \sum_{i=1}^{i=k} m_i)$, where $k + \sum_{i=1}^{i=k} m_i$ is the number of immediately nested occurrences of $\dot{f}_n$ in $\dot{f}_n(s_1, s_2, \ldots, s_n)$. $\square$

5.4.2 Unsquashing Shape Terms

UNSQUASH does the opposite of SQUASH. It rematerializes a shape term from $\dot{f}_n$ and *tuple* by recursively left associating the shape terms in *tuple* with the n-ary shape

- 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 ϑ in the asymptotic upper bound for *sort* depends on the time taken to compare a pair of shape terms by $\sqsubseteq$.
- $subs$ is a substitution represented as a linked list of B bindings of variables to terms.
- $||t||$ stands for the number of nodes in t.

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)	kth 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 $subs$	$O(B\|\|t\|\|)$

Table 5.1 Primitive Procedures

operator $\dot{f}_n$, or returns $\perp$ if such an association isn't possible. Some examples:

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

Figure 5.2 also presents the pseudo code for UNSQUASH. Four possibilities could 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 $\dot{f}_n$ cannot be directly applied on the shape terms in *tuple* so that $\perp$ 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; finally, (4) if $k > n$ but $n = 1$, $\perp$ is returned because an associative regrouping 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 4 below presents a necessary and sufficient condition for the third case to always yield a shape term.

Lemma 4 UNSQUASH$(\dot{f}_n, tuple)$ *returns a shape term iff*

$$\textit{length}(tuple) = n + m(n-1) \tag{5.8}$$

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

PROOF. We use induction on the length $n+m(n-1)$ of *tuple* to prove sufficiency. The base case corresponds to m being 0, in which case Lines 5 to 7 will be executed, causing UNSQUASH to return a shape term. For the induction hypothesis, suppose sufficiency holds for all $0 \leq m < M$ where M is some positive integer. Let $\textit{length}(tuple)$ be $n + M(n-1)$. Two situations may arise depending on the arity n:

1. If $n = 1$, then $n + M(n-1)$ will also be 1, in which case the test on Line 5 will succeed causing a shape term to be returned on Line 7.

2. If $n > 1$, then Lines 8 to 11 get executed, and UNSQUASH gets called recursively on a tuple of length $n + M(n-1) - n + 1 = n + (M-1)(n-1)$. From the induction hypothesis, this means that Line 11 will return a shape term.

Hence, UNSQUASH returns a shape term whenever Equation (5.8) holds.

To prove necessity, we assume $\mathit{length}(tuple) \neq n + m(n-1)$ for any $m \geq 0$. If $k = \mathit{length}(tuple)$ is less than n, then Line 4 in UNSQUASH will return $\perp$. Therefore, let $k > n$. We can also assume that $n > 1$ in this situation because $\perp$ would otherwise be returned on Line 12. When $k > n$ and $n > 1$, Lines 8 to 11 get executed, leading to a recursive call of UNSQUASH on a tuple of length $k - n + 1$. Let p be the number of such recursive calls that happen in succession. Since every such recursive call occurs on a tuple shorter by $n - 1$ components, the base case on Lines 6 and 7 would eventually be reached only if

$$k - p(n-1) = n.$$

But this would mean $k = n + p(n-1)$, contradicting our assumption that for any $m \geq 0$, $\mathit{length}(tuple) \neq n + m(n-1)$. Hence, the only base case that can be reached is the one on Lines 3 and 4, implying a return value of $\perp$. □

5.4.3 An Invariant under Squashing and Unsquashing

From Lemmas 3 and 4, UNSQUASH$(\dot{f}_n, \text{SQUASH}(s))$ is always a shape term if $s = \dot{f}_n(s_1, s_2, \ldots, s_n)$. Obviously, UNSQUASH$(\dot{f}_n, \text{SQUASH}(s))$ may not be the same as s. However, it follows from Lemma 5 below that they will have the same number of immediately nested $\dot{f}_n$ tree nodes. Since SQUASH(s) only knocks off immediately nested occurrences of $\dot{f}_n$ in s, this also means that both s and UNSQUASH$(\dot{f}_n, \text{SQUASH}(s))$ will have exactly the same nodes, although not necessarily in the same positions.

Lemma 5 *If* UNSQUASH$(\dot{f}_n, tuple)$ *returns a shape term* s*, then*

$$(n-1)d = \mathit{length}(tuple) - n \tag{5.9}$$

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

PROOF. Since UNSQUASH($\dot{f}_n$, *tuple*) returns a shape term,

$$\textsf{length}(tuple) = n + m(n-1)$$

for some nonnegative integer m by Lemma 4. We now prove this claim using induction on m. When $m = 0$, Lines 5 to 7 in UNSQUASH get executed. Since UNSQUASH in this situation only introduces the topmost $\dot{f}_n$ node, both d and $\textsf{length}(tuple) - n$ are 0, so that the claim obviously holds for the base case.[5]

Suppose the claim is true for all $0 \leq m < M$ where M is some positive integer. If $\textsf{length}(tuple) = n + M(n-1)$, two situations could arise:

1. $n = 1$, in which case Lines 5 to 7 get executed. Then as before,

$$(n-1)d = 0 = \textsf{length}(tuple) - n.$$

2. $n > 1$, in which case Lines 8 to 11 get executed. Then

$$\textsf{length}(tuple') = \textsf{length}(tuple) - n + 1 = n + (M-1)(n-1). \tag{5.10}$$

From Lemma 4, this means that UNSQUASH($\dot{f}_n$, *tuple'*) on Line 11 will return a shape term, say s'. If d' is the number of nested occurrences of $\dot{f}_n$ in s' added by UNSQUASH, then from the induction hypothesis of this lemma,

$$(n-1)d' = \textsf{length}(tuple') - n.$$

However, $d = d' + 1$ because the ongoing invocation UNSQUASH($\dot{f}_n$, *tuple*) introduces a $\dot{f}_n$ on Line 9 that becomes nested on return from Line 11. Therefore

$$(n-1)d = (n-1)d' + (n-1) = \textsf{length}(tuple') - n + n - 1.$$

Using Equation (5.10) in the above, this gives us

$$(n-1)d = \textsf{length}(tuple) - n + 1 - 1 = \textsf{length}(tuple) - n.$$

From the above two cases, the claim is true for $m = M$, if true for $m = M - 1$. Considering the base case, this implies that the claim is always true. □

[5]The topmost $\dot{f}_n$ doesn't count because d is the number of nested occurrences of $\dot{f}_n$.

5.5 A Symbolic Evaluator

Figure 5.3 shows an algorithm called ONE-PASS-EVAL, inspired by the Standard Evaluation Sequence in the Mathematica programming language [99, page 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 $\mathcal{A}$. The algorithm 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.
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 could be viewed as a canonicalized version of $\dot{f}_n(s'_1, s'_2, \ldots, s'_n)$.
4. Line 17 that rewrites the above result by using $\mathcal{A}$ with a fixed rewrite tuple ϱ.

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 3, 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 4. 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 5.4 shows MULTI-PASS-EVAL, a general-purpose evaluator of shape terms that uses the services of ONE-PASS-EVAL. MULTI-PASS-EVAL works by repeatedly applying ONE-PASS-EVAL until two successive applications produce the same result.

5.5.1 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

ONE-PASS-EVAL(s)

if *root*$(s) = s$
    return s
$\dot{f}_n \leftarrow$ *root*(s)
$n \leftarrow$ *arity*$(\dot{f}_n)$
$tuple \leftarrow$ *children*(s)
for $i \leftarrow 1$ **to** n **do**
        $s_i \leftarrow$ *component*$(tuple, i)$
        $s'_i \leftarrow$ ONE-PASS-EVAL(s_i)
$tuple' \leftarrow (s'_1, s'_2, \ldots, s'_n)$
if *isassociative*$(\dot{f}_n)$
    $tuple' \leftarrow$ SQUASH(*build*$(\dot{f}_n, tuple')$)
if *iscommutative*$(\dot{f}_n)$
    $tuple' \leftarrow$ *sort*$(\sqsubseteq, tuple')$
if *isassociative*$(\dot{f}_n)$
    $t \leftarrow$ UNSQUASH$(\dot{f}_n, tuple')$
else
    $t \leftarrow$ *build*$(\dot{f}_n, tuple')$
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. 5.3 A One-Pass Depth-First Evaluator

MULTI-PASS-EVAL(s)

$s' \leftarrow s$
do
        $s'' \leftarrow s'$
        $s' \leftarrow$ ONE-PASS-EVAL(s')
until $s' = s''$
return s'

Fig. 5.4 A Multi-Pass Evaluator

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 [33]. A binary relation $\rhd$ on W is defined to be well-founded if there are no infinitely decreasing chains of the form $w_0 \rhd w_1 \rhd w_2 \rhd \ldots$, where w_i is an element in W [74]. 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 3 below shows that MULTI-PASS-EVAL will always terminate if $\rhd$ is a well-founded relation such that $s \unrhd$ ONE-PASS-EVAL(s) for every shape term s. By considering the steps in the code for ONE-PASS-EVAL in Figure 5.3, Lemma 6 then presents a set of conditions that if honored by $\rhd$, ensures $s \unrhd$ ONE-PASS-EVAL(s) for every s in Λ.

Theorem 3 *If $\rhd$ is a well-founded relation such that $s \unrhd$* ONE-PASS-EVAL(s) *for any shape term s, then* MULTI-PASS-EVAL *will always terminate.*

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

$$\text{ONE-PASS-EVAL}^i(s) = \begin{cases} \text{ONE-PASS-EVAL}(\text{ONE-PASS-EVAL}^{i-1}(s)) & \text{if } i > 0, \\ s & \text{if } i = 0. \end{cases} \tag{5.11}$$

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 MULTI-PASS-EVAL to succeed. □

Lemma 6 *If $\rhd$ is a transitive binary relation on Λ such that*

$$\begin{aligned}&\dot{f}_n(s_1, s_2, \ldots, s_n) \unrhd \dot{f}_n(s_1', s_2', \ldots, s_n')\\ &\qquad \textit{if } s_i \unrhd s_i' \textit{ for all } 1 \le i \le n,\end{aligned} \tag{5.12}$$

$$\begin{aligned}&\dot{f}_n(s_1, s_2, \ldots, s_n) \unrhd \textsf{\textit{build}}\big(\dot{f}_n, \textsf{\textit{sort}}(\sqsubseteq, (s_1, s_2, \ldots, s_n))\big)\\ &\qquad \textit{if } \textsf{\textit{isassociative}}(\dot{f}_n) = \textsf{\textit{false}} \wedge \textsf{\textit{iscommutative}}(\dot{f}_n) = \textsf{\textit{true}},\end{aligned} \tag{5.13}$$

$$\begin{aligned}&\dot{f}_n(s_1, s_2, \ldots, s_n) \unrhd \textsc{unsquash}\big(\dot{f}_n, \textsc{squash}(\dot{f}_n(s_1, s_2, \ldots, s_n))\big)\\ &\qquad \textit{if } \textsf{\textit{isassociative}}(\dot{f}_n) = \textsf{\textit{true}} \wedge \textsf{\textit{iscommutative}}(\dot{f}_n) = \textsf{\textit{false}},\end{aligned} \tag{5.14}$$

$$\begin{aligned}&\dot{f}_n(s_1, s_2, \ldots, s_n) \unrhd \textsc{unsquash}\big(\dot{f}_n, \textsf{\textit{sort}}(\sqsubseteq, \textsc{squash}(\dot{f}_n(s_1, s_2, \ldots, s_n)))\big)\\ &\qquad \textit{if } \textsf{\textit{isassociative}}(\dot{f}_n) = \textsf{\textit{true}} \wedge \textsf{\textit{iscommutative}}(\dot{f}_n) = \textsf{\textit{true}},\end{aligned} \tag{5.15}$$

$$\xi(l) \unrhd \xi(r) \textit{ for all } \xi \in \Xi \textit{ and all } l \to r \textit{ in } \varrho, \tag{5.16}$$

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

Proof. We prove this claim using structural induction on s. If s is atomic, then ONE-PASS-EVAL returns s at Line 2 so that ONE-PASS-EVAL$(s) = s$. Therefore, let s be of the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$ and let $s_i \unrhd$ ONE-PASS-EVAL(s_i) for all $1 \le i \le n$ be the induction hypothesis. Then at the end of the **for** loop on Lines 6 to 8,

$$s_i \unrhd s_i', \tag{5.17}$$

and just before Line 17, any one of the following equations will hold, depending on whether $\dot{f}_n$ is noncommutative and nonassociative, commutative and nonassociative, noncommutative and associative, or commutative and associative respectively:

$$t = \dot{f}_n(s_1', s_2', \ldots, s_n'), \tag{5.18}$$

$$t = \textsf{\textit{build}}\big(\dot{f}_n, \textsf{\textit{sort}}(\sqsubseteq, (s_1', s_2', \ldots, s_n'))\big), \tag{5.19}$$

$$t = \textsc{unsquash}\big(\dot{f}_n, \textsc{squash}(\dot{f}_n(s_1', s_2', \ldots, s_n'))\big), \tag{5.20}$$

$$t = \textsc{unsquash}\big(\dot{f}_n, \textsf{\textit{sort}}\big(\sqsubseteq, \textsc{squash}(\dot{f}_n(s_1', s_2', \ldots, s_n'))\big)\big). \tag{5.21}$$

Suppose $t = \dot{f}_n(s_1', s_2', \ldots, s_n')$ by Equation (5.18). Then from the induction hypothesis and Equation (5.12), $\dot{f}_n(s_1, s_2, \ldots, s_n) \unrhd t$ implying $s \unrhd t$. Similarly, if

$t = \mathit{build}\big(\dot{f}_n, \mathit{sort}(\sqsubseteq, (s'_1, s'_2, \ldots, s'_n))\big)$, then from Equation (5.13)

$$\dot{f}_n(s'_1, s'_2, \ldots, s'_n) \trianglerighteq t.$$

Because $\dot{f}_n(s_1, s_2, \ldots, s_n) \trianglerighteq \dot{f}_n(s'_1, s'_2, \ldots, s'_n)$ from the induction hypothesis and Equation (5.12), and since $\triangleright$ is stated to be transitive, this gives us

$$\dot{f}_n(s_1, s_2, \ldots, s_n) \trianglerighteq t.$$

Likewise, by considering each of the remaining Equations (5.20) to (5.21), and the corresponding equation in Equations (5.14) to (5.15) along with the transitivity of $\triangleright$ and the induction hypothesis, we can show that

$$s \trianglerighteq t \tag{5.22}$$

is always true after Line 16. From Equations (5.6) and (5.16), we also know that

$$t \trianglerighteq \mathcal{A}(t, \varrho). \tag{5.23}$$

From the transitivity of $\triangleright$, we thus obtain $s \trianglerighteq \mathcal{A}(t, \varrho)$. But $\textsc{one-pass-eval}(s) = \mathcal{A}(t, \varrho)$. Hence $s \trianglerighteq \textsc{one-pass-eval}(s)$ is always true. □

5.5.1.1 Fulfilling the Termination Condition

Does there exist a well-founded relation $\triangleright$ on Λ that satisfies the conditions in Lemma 6? Clearly, the existence of such a $\triangleright$ is also dependent on which rules are included in ϱ. We shall now show that even if every rule $l \to r$ in ϱ modelled an identity $l = r$ (and *not* $r = l$) in Table 4.2, excluding of course the commutativity and associativity identities, then there exists a well-founded $\triangleright$ that complies with the requirements of Lemma 6. Our technique of constructing such a $\triangleright$ will be by assigning "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

$$\varpi(s) = \begin{cases} c & \text{if } s \in \Lambda^* \cup \mathcal{S}, \\ \varpi(s_1)\varpi(s_2) & \text{if } s = s_1 \dot{*} s_2, \\ c^{\varpi(s_1)} & \text{if } s = \dot{\neg}\, s_1, \\ 1 + c\sum_{i=1}^{n} \varpi(s_i) & \text{if } s = \mathit{build}\big(\dot{f}_n, \mathit{sort}(\sqsubseteq, \text{SQUASH}(s))\big) \\ & \quad \wedge\ \mathit{isassociative}(\dot{f}_n) = \mathit{false} \\ & \quad \wedge\ \mathit{iscommutative}(\dot{f}_n) = \mathit{true}, \\ 1 + c\sum_{i=1}^{n} \varpi(s_i) & \text{if } s = \text{UNSQUASH}(\dot{f}_n, \text{SQUASH}(s)) \\ & \quad \wedge\ \mathit{isassociative}(\dot{f}_n) = \mathit{true} \\ & \quad \wedge\ \mathit{iscommutative}(\dot{f}_n) = \mathit{false}, \\ 1 + c\sum_{i=1}^{n} \varpi(s_i) & \text{if } s = \text{UNSQUASH}\big(\dot{f}_n, \mathit{sort}(\sqsubseteq, \text{SQUASH}(s))\big) \\ & \quad \wedge\ \mathit{isassociative}(\dot{f}_n) = \mathit{true} \\ & \quad \wedge\ \mathit{iscommutative}(\dot{f}_n) = \mathit{true}, \\ c + c\sum_{i=1}^{n} \varpi(s_i) & \text{else if } s = \dot{f}_n(s_1, s_2, \ldots, s_n), \end{cases} \tag{5.24}$$

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

$$s \rhd t \iff \varpi(s) > \varpi(t). \tag{5.25}$$

Transitivity and Proof of Satisfaction of Equation (5.12) To prove that the above relation honors Lemma 6, we first need to show that $\rhd$ is transitive. This is obvious from Equation (5.25). We next need to show that $\rhd$ satisfies Equation (5.12). 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 $s_i \unrhd s'_i$ for all $1 \leq i \leq n$. Since Equation (5.12) 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:

$$\varpi(s) - \varpi(s') = \begin{cases} c\sum_{i=1}^{n}(\varpi(s_i) - \varpi(s'_i)), \\ (c-1) + c\sum_{i=1}^{n}(\varpi(s_i) - \varpi(s'_i)), \\ -(c-1) + c\sum_{i=1}^{n}(\varpi(s_i) - \varpi(s'_i)). \end{cases} \tag{5.26}$$

Since $s_j \rhd s'_j$, we have $\varpi(s_j) > \varpi(s'_j)$ from Equation (5.25). 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 (5.26) is always positive. The remaining case is when $\dot{f}_n$ is either $\dot{*}$ or $\dot{\neg}$. From Equation (5.24), we then have

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

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 (5.13) to (5.15) It is straightforward to show that $\rhd$ fulfills each of the Equations (5.13) to (5.15). For example, let $\dot{f}_n$ be a nonassociative and commutative shape operator and let

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

This gives us the following weights from Equation (5.24):

$$\begin{aligned} \varpi(\dot{f}_n(s_1, s_2, \ldots, s_n)) &= c + c\sum_{i=1}^{n}\varpi(s_i), \\ \varpi\big(\textsf{\textit{build}}\big(\dot{f}_n, \textsf{\textit{sort}}(\sqsubseteq, (s_1, s_2, \ldots, s_n))\big)\big) &= 1 + c\sum_{i=1}^{n}\varpi(s_i). \end{aligned}$$

Thus $\varpi(\dot{f}_n(s_1, s_2, \ldots, s_n)) > \varpi\big(\textsf{\textit{build}}\big(\dot{f}_n, \textsf{\textit{sort}}(\sqsubseteq, (s_1, s_2, \ldots, s_n))\big)\big)$ from the above.

Proof of Satisfaction of Equation (5.16) Consider the rules $l \to r$ in Table 5.2 on pages 71 to 72 that model every identity $l = r$ from Table 4.2, and some more. The additional identities, shown as a bottom section of the table, describe how operations on explicit shape terms map to explicit shape terms. To demonstrate that $\rhd$ honors Equation (5.16) even if all of these rules were included in the fixed rewrite tuple ϱ 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 5.2 gives a lower bound on these computed differences. For instance, for the identity $(s \dotplus t) \dot{*} (s \dotplus t) = s \dot{*} s \dotplus t \dot{*} t$ in Table 4.2,

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

where x and y are shorthands for $\xi(s)$ and $\xi(t)$. But from Equation (5.24),

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

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

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

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

But both X and Y are at least c by Equation (5.24). Therefore, since c is greater than 1, the last expression in the above is clearly positive. It can be similarly verified that all the other weight-difference lower bounds in Table 5.2 are positive.[6] For the last three rules, the lower bounds are obtained by noting that $\xi(\widehat{u})$, $\xi(\widehat{v})$ and the rules' right-hand sides are shape terms in Λ^* for any substitution $\xi \in \Xi$.

[6] As an example, how can we show that $1 + cX + c^{X+1} - X^2$, which is the lower bound on $\varpi(\xi(s \dotplus \dot{\neg} s)) - \varpi(\xi(s \dot{*} s))$ where $X = \varpi(\xi(s))$, is positive when c is an integer greater than 1? This

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

$l \to r$	Lower Bound on $\varpi(\xi(l)) - \varpi(\xi(r))$
$\overline{\boldsymbol{\pi}} \dot{+} s \to \overline{\boldsymbol{\pi}}$	$c(X + c - 1) + 1$
$s \dot{+} \overline{\boldsymbol{\pi}} \to \overline{\boldsymbol{\pi}}$	$c(X + c - 1) + 1$
$\overline{\boldsymbol{\iota}} \dot{+} s \to s$	$(c - 1)X + c^2 + 1$
$s \dot{+} \overline{\boldsymbol{\iota}} \to s$	$(c - 1)X + c^2 + 1$
$s \dot{+} s \to s$	$(2c - 1)X + 1$
$\overline{\boldsymbol{\pi}} \dot{*} s \to \overline{\boldsymbol{\pi}}$	$c(X - 1)$
$s \dot{*} \overline{\boldsymbol{\pi}} \to \overline{\boldsymbol{\pi}}$	$c(X - 1)$
$\overline{\boldsymbol{\iota}} \dot{*} s \to s$	$(c - 1)X$
$s \dot{*} \overline{\boldsymbol{\iota}} \to s$	$(c - 1)X$
$(s \dot{*} s) \dot{*} (s \dot{*} s) \to s \dot{*} s$	$X^2(X^2 - 1)$
$s \dot{*} (s \dot{*} s) \to s \dot{*} s$	$X^2(X - 1)$
$(s \dot{*} s) \dot{*} s \to s \dot{*} s$	$X^2(X - 1)$
$s \dot{*} \big(s \dot{*} (s \dot{*} (s \dot{*} t))\big) \to s \dot{*} (s \dot{*} (s \dot{*} t))$	$X^3 Y(X - 1)$
$\big(((s \dot{*} t) \dot{*} t) \dot{*} t\big) \dot{*} t \to ((s \dot{*} t) \dot{*} t) \dot{*} t$	$XY^3(Y - 1)$

Table 5.2 Rules from Table 4.2 with Differences (contd. on next page)

---

could be done by first finding the derivative of $Z = 1 + cX + c^{X+1} - X^2$ with respect to X:

$$\frac{dZ}{dX} = c + c^{X+1} \ln c - 2X. \tag{5.27}$$

By differentiating the above once more with respect to X, we have

$$\frac{d^2 Z}{dX} = c^{X+1} (\ln c)^2 - 2. \tag{5.28}$$

Because $X \geq c$, we get $c^{X+1}(\ln c)^2 \geq c^{c+1}(\ln c)^2 \geq 3.84362$. Therefore, $\frac{d^2 Z}{dX}$ in Equation (5.28) is always positive. This means that $\frac{dZ}{dX}$ is an increasing function of X. Also, $\frac{dZ}{dX}$ at $X = c$ is

$$c + c^{c+1} \ln c - 2c \geq 3.54518. \tag{5.29}$$

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

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

Table 5.2 Rules from Table 4.2 with Differences (contd. from previous page)

Proof of Satisfaction of the Termination Condition For the fulfillment of Theorem 3, all that remains is to show that $\rhd$ is well-founded. This is obvious since $\varpi(s) \geq c$ for any shape term s from Equation (5.24), and because on positive integers, the arithmetic ordering $<$ is known to be well-founded.

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

$$1 + c^2 + c^{c+1} - c^2 = 1 + c^{c+1}, \tag{5.30}$$

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

5.5.1.2 Off-line Verification of Termination

Clearly, coming up with a $\rhd$ that is well-founded and that satisfies all the requirements in Lemma 6 is a nontrivial task, exemplified by the exercise in § 5.5.1.1. It is easy to arrive at a well-founded $\rhd$ that meets *some* of the requirements in Lemma 6, 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.

Luckily, the weight function in Equation (5.24) 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 ϱ, the only condition that risks being broken is Equation (5.16). 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 (5.16) and reject others. Either way, observe that the testing of all these requirements can be done once, and off-line, before the evaluator is actually used in any interpretation or compilation run.

Another point to note is that it isn't necessary to know the particular total order $\sqsubseteq$ used in *sort* or the way UNSQUASH reassociates shape terms to come up with a $\rhd$ that will ensure MULTI-PASS-EVAL's termination. Thus, rather than returning a left-associated tree, UNSQUASH could have chosen any other association order as its canonical form and Equation (5.25) would have still defined a well-founded relation that honors Theorem 3. This affords some flexibility in an implementation.

5.5.2 Unique Normalization: The Church-Rosser Property

The next question is whether MULTI-PASS-EVAL(s) always produces the same result for a given shape term s on 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 on Line 17 isn't, as discussed in § 5.3. How can $\mathcal{A}$ be made a function? One way of achieving this, shown by Lemma 7 below, is to restrict $\mathcal{A}$ to a certain subset of $\mathcal{R}$.

Lemma 7 *Let $\acute{\mathcal{R}}$ be a subset of $\mathcal{R}$ such that*

$$\acute{\mathcal{R}} = \{(l_1 \to r_1, l_2 \to r_2, \ldots, l_k \to r_k) \mid (l_1 \to r_1, l_2 \to r_2, \ldots, l_k \to r_k) \in \mathcal{R} \\ \wedge \mathcal{V}(l_i) \supseteq \mathcal{V}(r_i) \text{ for all } 1 \leq i \leq k\}. \quad (5.31)$$

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

Proof. We prove this lemma using induction on the length of the rewrite tuples in $\acute{\mathcal{R}}$. The base case occurs for unit-length rewrite tuples. If there exists no ξ in Ξ that matches l_1 to s, then $\acute{\mathcal{A}}(s, (l_1 \to r_1)) = s$ by Equation (5.6). So suppose there exist two substitutions ξ' and ξ'' that match l_1 to s. Then by Lemma 2,

$$\xi'/\mathcal{V}(l_1) = \xi''/\mathcal{V}(l_1).$$

But $\mathcal{V}(l_1) \supseteq \mathcal{V}(r_1)$ from Equation (5.31). Hence,

$$\xi'/\mathcal{V}(r_1) = \xi''/\mathcal{V}(r_1),$$

implying that in this situation too, $\acute{\mathcal{A}}(s, (l_1 \to r_1))$ is single-valued.

For the induction hypothesis, assume the claim to be true for all rewrite tuples of length $k < K$ where K is some integer greater than 1. Consider the application $\acute{\mathcal{A}}(s, (l_1 \to r_1, l_2 \to r_2, \ldots, l_K \to r_K))$. If there exist substitutions in Ξ that match l_1 to s, we know by the earlier analysis that the image of $\acute{\mathcal{A}}$ will be unique. If there doesn't exist any substitutions that match l_1 to s, then by Equation (5.6),

$$\acute{\mathcal{A}}(s, (l_1 \to r_1, l_2 \to r_2, \ldots, l_K \to r_K)) = \acute{\mathcal{A}}(s, (l_2 \to r_2, \ldots, l_K \to r_K)),$$

which by the induction hypothesis is unique. Hence $\acute{\mathcal{A}}$ is always single-valued. □

Since $\mathcal{V}(l) \supseteq \mathcal{V}(r)$ for every rule $l \to r$ in Table 5.2, it follows that even if all the rules in Table 5.2 were included in the fixed rewrite tuple ϱ, the outcome of MULTI-PASS-EVAL(s) would still be unique on termination.

5.5.3 Soundness

A transformational system is *sound* if it is always meaning-preserving under conversions. That is, if such a system converts a representation A with meaning M to a representation B, then B will also have the meaning M. Hence to prove that MULTI-PASS-EVAL is sound, we have to demonstrate that MULTI-PASS-EVAL(s) and s are algebraically equivalent with respect to the rules in ϱ. This can be shown by considering the subset Ξ_g of substitutions that are mappings from $\mathcal{S}$ to the set of ground shape terms Λ_g and then proving that the *meanings* of $\xi_g(\text{ONE-PASS-EVAL}(s))$ and $\xi_g(s)$, where ξ_g is any substitution in Ξ_g, are always the same for any shape term s.

Lemma 8 *For any shape term s and any substitution $\xi_g \in \Xi_g$,*

$$\psi\big(\xi_g(\text{ONE-PASS-EVAL}(s))\big) = \psi(\xi_g(s)). \tag{5.32}$$

PROOF. We prove this claim using structural induction on s. If $s \in \Lambda^* \cup \mathcal{S}$, then since $\text{ONE-PASS-EVAL}(s) = s$, Equation (5.32) trivially holds for any substitution $\xi_g \in \Xi_g$. Therefore, suppose $s = \dot{f}_n(s_1, s_2, \ldots, s_n)$ and let

$$\psi(\xi_g(\text{ONE-PASS-EVAL}(s_i))) = \psi(\xi_g(s_i)) \tag{5.33}$$

for all $1 \leq i \leq n$ be the induction hypothesis. Now for any $\dot{f}_n$ and $g_i \in \Lambda_g$, we have

$$\psi(\dot{f}_n(g_1, g_2, \ldots, g_n)) = \dot{f}_n(\psi(g_1), \psi(g_2), \ldots, \psi(g_n)). \tag{5.34}$$

As in Equation (5.5) on page 54, $\dot{f}_n$ is used syntactically on the left-hand side and semantically on the right-hand side in Equation (5.34). If $\dot{f}_n$ were additionally given to be commutative, then from the algebraic definition of commutativity and the definitions of the *build* and *sort* primitives in Table 5.1, we would have

$$\psi(\dot{f}_n(g_1, g_2, \ldots, g_n)) = \psi\Big(\mathit{build}\big(\dot{f}_n, \mathit{sort}(\sqsubseteq, (g_1, g_2, \ldots, g_n))\big)\Big). \tag{5.35}$$

We also have the following two results, easily derivable from the definitions of SQUASH and UNSQUASH in Figure 5.2, that hold when $\dot{f}_n$ is associative, but noncommutative (Equation (5.36)) and commutative (Equation (5.37)) respectively:

$$\begin{aligned}&\psi(\dot{f}_n(g_1, g_2, \ldots, g_n)) = \\ &\qquad \psi\big(\text{UNSQUASH}\big(\dot{f}_n, \text{SQUASH}(\dot{f}_n(g_1, g_2, \ldots, g_n))\big)\big),\end{aligned} \tag{5.36}$$

$$\begin{aligned}&\psi(\dot{f}_n(g_1, g_2, \ldots, g_n)) = \\ &\qquad \psi\Big(\text{UNSQUASH}\big(\dot{f}_n, \textit{sort}\big(\sqsubseteq, \text{SQUASH}(\dot{f}_n(g_1, g_2, \ldots, g_n))\big)\big)\Big).\end{aligned} \tag{5.37}$$

Just before Line 17, depending on whether $\dot{f}_n$ is noncommutative and nonassociative, commutative and nonassociative, noncommutative and associative, or commutative and associative respectively, one of the following equations will hold:

$$t = \dot{f}_n(s'_1, s'_2, \ldots, s'_n), \tag{5.38}$$

$$t = \textit{build}\big(\dot{f}_n, \textit{sort}(\sqsubseteq, (s'_1, s'_2, \ldots, s'_n))\big), \tag{5.39}$$

$$t = \text{UNSQUASH}\big(\dot{f}_n, \text{SQUASH}(\dot{f}_n(s'_1, s'_2, \ldots, s'_n))\big), \tag{5.40}$$

$$t = \text{UNSQUASH}\big(\dot{f}_n, \textit{sort}\big(\sqsubseteq, \text{SQUASH}(\dot{f}_n(s'_1, s'_2, \ldots, s'_n))\big)\big), \tag{5.41}$$

where $s'_i = \text{ONE-PASS-EVAL}(s_i)$. By Equations (5.34) to (5.37), this means

$$\psi(\xi_g(t)) = \dot{f}_n\big(\psi(\xi_g(s'_1)), \psi(\xi_g(s'_2)), \ldots, \psi(\xi_g(s'_n))\big) \tag{5.42}$$

is always true just before Line 17. Hence from the induction hypothesis

$$\psi(\xi_g(t)) = \dot{f}_n\big(\psi(\xi_g(s_1)), \psi(\xi_g(s_2)), \ldots, \psi(\xi_g(s_n))\big) = \psi(\xi_g(s)). \tag{5.43}$$

Now from Equation (5.6), one of the following will hold on Line 17:

$$\mathcal{A}(t, \varrho) = \begin{cases} \xi'(r') & \text{if } \exists l' \to r' \text{ in } \varrho \text{ such that } \xi'(l') = t \text{ for some } \xi' \in \Xi, \\ t & \text{otherwise.} \end{cases}$$

Therefore $\psi\big(\xi_g(\mathcal{A}(t, \varrho))\big)$ can be either $\psi\big(\xi_g(\xi'(r'))\big)$ or $\psi(\xi_g(t))$. However, by the nature of characterizations, $\psi\big(\xi_g(\xi'(l))\big) = \psi\big(\xi_g(\xi'(r))\big)$ for all rules $l \to r$ in ϱ and all $\xi_g \in \Xi_g$. Because $\xi'(l') = t$ when $\mathcal{A}(t, \varrho) = \xi'(r')$, this therefore means

$$\psi\big(\xi_g(\mathcal{A}(t, \varrho))\big) = \psi(\xi_g(t))$$

always holds. Hence from the above and Equation (5.43),

$$\psi\big(\xi_g(\mathcal{A}(t,\varrho))\big) = \psi(\xi_g(s)).$$

Thus the claim $\psi\big(\xi_g(\text{ONE-PASS-EVAL}(s))\big) = \psi(\xi_g(s))$ for any substitution $\xi_g \in \Xi_g$ since $\mathcal{A}(t,\varrho)$ on Line 17 is ONE-PASS-EVAL(s). □

5.5.4 Complexity Analysis

If the termination condition in § 5.5.1 is satisfied, then for a certain total order $\sqsubseteq$,[7] MULTI-PASS-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. This figure can be derived by choosing linked lists as the data structure for ordered tuples, tree node children and substitutions, although as we shall see, it isn't so much determined by this choice of data structure but by the fact that a comparison sort of k elements takes $O(\vartheta k \log k)$ time, where ϑ is an upper bound on the time taken to compare a pair of elements. Table 5.1 on page 59 shows the asymptotic upper bounds on the running times of the various primitives when linked lists are the data structures used. Note that though this is $O(k)$ for *component*(T,k), *component*(T,k) is typically used in a sequence of *length*(T) successive accesses giving an amortized running time of $O(1)$.

5.5.4.1 Comparing Shape Terms

Let $\sqsubseteq_{\Lambda^*}$, $\sqsubseteq_{\mathcal{S}}$ and $\sqsubseteq_{\dot{\mathcal{F}}}$ be preexisting total orders on Λ^*, $\mathcal{S}$ and $\dot{\mathcal{F}}$ respectively. For example, $\sqsubseteq_{\mathcal{S}}$ and $\sqsubseteq_{\dot{\mathcal{F}}}$ could be the alphabetic order of symbol and function names, and $\sqsubseteq_{\Lambda^*}$ could be the lexicographic order [12] of canonical shape tuples that correspond to explicit shape terms. Elements in Λ can then be ordered by $\sqsubseteq$ as follows:

[7] From the discussions in § 5.5.1 and § 5.5.2, we know that a $\sqsubseteq$ can be chosen without affecting either the termination or the Church-Rosser traits of MULTI-PASS-EVAL.

$$
s \sqsubseteq t \iff \begin{cases}
\quad\big((s \in \Lambda^*) \wedge (t \in \Lambda^*) \wedge (s \sqsubseteq_{\Lambda^*} t)\big) \\
\bigvee\big((s \in \Lambda^*) \wedge (t \notin \Lambda^*)\big) \\
\bigvee\big((s \in \mathcal{S}) \wedge (t \in \mathcal{S}) \wedge (s \sqsubseteq_{\mathcal{S}} t)\big) \\
\bigvee\big((s \in \mathcal{S}) \wedge (t \notin \Lambda^* \cup \mathcal{S})\big) \\
\bigvee\big((s = \dot{f}_n(u_1, u_2, \ldots, u_n)) \\
\qquad \wedge (t = \dot{f}_n(u_1, u_2, \ldots, u_n))\big) \\
\bigvee\big((s = \dot{f}_n(s_1, s_2, \ldots, s_n)) \\
\qquad \wedge (t = \dot{g}_m(t_1, t_2, \ldots, t_m)) \\
\qquad \wedge (\dot{f}_n \neq \dot{g}_m) \wedge (\dot{f}_n \sqsubseteq_{\dot{\mathcal{F}}} \dot{g}_m)\big) \\
\bigvee\big((s = \dot{f}_n(u_1, u_2, \ldots, u_{j-1}, s_j, \ldots, s_n)) \\
\qquad \wedge (t = \dot{f}_n(u_1, u_2, \ldots, u_{j-1}, t_j, \ldots, t_n)) \\
\qquad \wedge (s_j \neq t_j) \wedge (s_j \sqsubseteq t_j) \wedge (1 \leq j \leq n)\big).
\end{cases} \tag{5.44}
$$

It is easy to verify that the seven disjunct clauses in Equation (5.44) are mutually exclusive. For instance, the first is true iff both s and t belong to Λ^* and $s \sqsubseteq_{\Lambda^*} t$. The second states that every explicit shape term precedes every symbolic shape term in $\sqsubseteq$. The third is true iff s and t are symbols and $s \sqsubseteq_{\mathcal{S}} t$. By the fourth, every symbol precedes every nonatomic shape term in $\sqsubseteq$. Finally, the last three clauses relate shape terms of the form $\dot{f}_n(s_1, s_2, \ldots, s_n)$ and $\dot{g}_m(t_1, t_2, \ldots, t_m)$.

The relation in Equation (5.44) can also be shown to be reflexive, antisymmetric and transitive, i.e., a partial order. Moreover, since either $s \sqsubseteq t$ or $t \sqsubseteq s$ will be true for any pair of shape terms s and t, $\sqsubseteq$ is a total order.

If the lengths of symbol names are bounded by a constant,[8] both $\sqsubseteq_{\mathcal{S}}$ and $\sqsubseteq_{\dot{\mathcal{F}}}$ can be tested in constant time. If $\sqsubseteq_{\Lambda^*}$ is defined as the lexicographic order of canonical shape tuples, then it can be tested in $O(R)$ time where R is an upper bound on an array's canonical rank. Hence, for any pair of shape terms s and t, whether $s \sqsubseteq t$ is true or false can be established in $O(\min(\|s\|, \|t\|)R)$ time.

[8] This is practically always true. Indeed, most languages impose a limit on the length of symbol names. For example, identifiers cannot exceed six characters in FORTRAN 77 [59], and only the first 31 characters are significant in MATLAB [66].

5.5.4.2 Many-Sorted Matching

Figure 5.6 displays an algorithm that realizes the rules applicator defined in Equation (5.6). It uses a procedure MATCH, shown in Figure 5.5, to determine if a substitution exists that matches the left-hand side l of a rule to the shape term s. MATCH(l, s) returns ***true*** if l can be matched to s, and *false* if not. In the former case, a matching substitution is also made available in a global data structure named $subs$. Prior to calling MATCH, $\mathcal{A}$ initializes $subs$ by setting the bindings of all *distinct* variables that occur in s to $\perp$, signifying their initial undefined status. If a match is detected, Line 6 proceeds to substitute only those variables in s that have well-defined bindings in $subs$. The MATCH algorithm consists of four parts:

1. Lines 1 to 2 that check whether l and s are equal explicit shape terms.
2. Lines 3 to 9 that test whether l is a shape variable of the same sort as s, and if so, update $subs$ with the binding $l \hookrightarrow s$. If either the sorts don't match, or if l is already bound to a shape term different from s, *false* is returned.
3. When control reaches Line 10, l will be of the form $\dot{f}_n(l'_1, l'_2, \ldots, l'_n)$. If s then doesn't have the same root as l, Line 11 returns *false*.
4. l and s are of the forms $\dot{f}_n(l'_1, l'_2, \ldots, l'_n)$ and $\dot{f}_n(s_1, s_2, \ldots, s_n)$ between Lines 12 and 19. Lines 15 to 19 then recursively match each l'_i to s_i.

The first three parts above form the base cases that break the recursion in MATCH. Both the first and third base cases run in constant time. The second base case, on Lines 3 to 9, involves the *lookup* and *update* primitives that require $O(B)$ time, where B is the number of distinct variables in s. In addition, the test on Line 8 takes $O(R\|s\|)$ time, where $O(R)$ is the time for comparing a pair of shape terms in Λ^* as discussed in § 5.5.4.1. Since any base case therefore runs in $O(R\|s\| + B)$ time, the total running time $\mathcal{T}(\text{MATCH}(l, s))$ of an invocation MATCH(l, s) becomes

$$\mathcal{T}(\text{MATCH}(l, s)) = O(R\|l\|\|s\| + B\|l\|). \tag{5.45}$$

This means that the total contribution MATCH(l, s) makes to the running time of $\mathcal{A}(s, tuple)$ in Figure 5.6 is $O(R(\|s\| + B) \sum_{i=1}^{k} \|l_i\|)$ where k is the length of $tuple$

```
MATCH(l, s)
    ☞ If there exists a substitution ξ such
    ☞ that ξ(l) = s, returns true; otherwise
    ☞ returns false. If true, the substitution
    ☞ used is stored in a globally visible
    ☞ data structure named subs.
    if (l ∈ Λ*) ∧ (l ≠ s)
       return false
    else if l ∈ S
       if (l ∈ Ŝ ∧ s ∉ Λ*) ∨ (l ∈ S̃ ∧ s ∈ Λ*)
          ☞ This means that the sort of l
          ☞ doesn't match the sort of s.
          return false
       else if lookup(l, subs) = ⊥
          update(l ↪ s, subs)
       else if lookup(l, subs) ≠ s
          return false
    else if root(l) ≠ root(s)
       return false
    else
       ☞ l and s are of the forms
       ☞ ḟ_n(l'_1, l'_2, ..., l'_n) and
       ☞ ḟ_n(s_1, s_2, ..., s_n).
       tuple_l ← children(l)
       tuple_s ← children(s)
       n ← arity(root(l))
       for i ← 1 to n do
          l'_i ← component(tuple_l, i)
          s_i ← component(tuple_s, i)
          if MATCH(l'_i, s_i) = false
             return false
    return true
```

Fig. 5.5 The Many-Sorted Matching Procedure

```
𝒜(s, tuple)
    ☞ Implements Equation (5.6).
    ☞ tuple is of the form
    ☞ (l_1 → r_1, l_2 → r_2, ..., l_k → r_k).
    k ← length(tuple)
    l → r ← component(tuple, 1)
    foreach x in 𝒱(s) do
        update(x ↪ ⊥, subs)
        ☞ ⊥ means the substitution for
        ☞ x hasn't yet been ascertained.
    if MATCH(l, s)
       return substitute(s, subs)
    else if k = 1
       return s
    else
       return 𝒜(s, drop(tuple, 1))
```

Fig. 5.6 The Rules Applicator Procedure and $l_i \rightarrow r_i$ are the rules in *tuple*. Since Lines 3 to 4 in Figure 5.6 take $O(B\|s\|)$ time and get executed until a match is finally detected on Line 5, we obtain

$$\mathcal{T}(\mathcal{A}(s, tuple)) = O(kB\|s\| + R(\|s\| + B)\sum_{i=1}^{k}\|l_i\|). \tag{5.46}$$

Though faster algorithms for MATCH exist [62], using them won't improve the asymptotic running time of ONE-PASS-EVAL because the asymptotic bottleneck lies elsewhere, specifically in the sorting step on Line 13 in Figure 5.3 (see § 5.5.4.3).

5.5.4.3 Asymptotic Upper Bound

It is obvious that ONE-PASS-EVAL(s) runs in constant time when s is atomic. Therefore, suppose that s is $\dot{f}_n(s_1, s_2, \ldots, s_n)$. From Figure 5.2, we know that

$$\mathcal{T}\big(\textsc{squash}(\textsf{\textit{build}}(\dot{f}_n, tuple'))\big) = O(\|\textsf{\textit{build}}(\dot{f}_n, tuple')\|)$$

for the invocation SQUASH(*build*$(\dot{f}_n, tuple')$) on Line 11 in Figure 5.3. But from Lines 8 and 9 in the same, $\|\textsf{\textit{build}}(\dot{f}_n, tuple')\| = O(\|\dot{f}_n(s'_1, s'_2, \ldots, s'_n)\|)$, where $s'_i =$ ONE-PASS-EVAL(s_i) for all $1 \leq i \leq n$. This gives us

$$\mathcal{T}\big(\textsc{squash}(\textsf{\textit{build}}(\dot{f}_n, tuple'))\big) = \sum_{i=1}^{n} O(\|s'_i\|). \tag{5.47}$$

How does $\|s'_i\|$ relate to $\|s_i\|$? This question is addressed in Lemma 9 below.

Lemma 9 *For any shape term s,* $\|$ONE-PASS-EVAL$(s)\| = O(\|s\|)$.

PROOF. This claim can be proved using structural induction on s. Let $s = \dot{f}_n(s_1, s_2, \ldots, s_n)$, and suppose that $\|$ONE-PASS-EVAL$(s_i)\| = O(\|s_i\|)$ for all $1 \leq i \leq n$ is the induction hypothesis. Then at the end of the **for** loop on Lines 6 to 8,

$$\|s'_i\| = O(\|s_i\|). \tag{5.48}$$

From the invariant in § 5.4.3, the shape term t computed on Lines 15 and 16 will always have exactly the same number of nodes as $\dot{f}_n(s'_1, s'_2, \ldots, s'_n)$. Therefore, after

Line 16, the following will hold by Equation (5.48):

$$\|t\| = 1 + \sum_{i=1}^{n} \|s_i'\| = 1 + \sum_{i=1}^{n} O(\|s_i\|) = O(\|s\|). \tag{5.49}$$

We now show that for the invocation $\mathcal{A}(t, \varrho)$ on Line 17,

$$\|\mathcal{A}(t, \varrho)\| = O(\|t\|). \tag{5.50}$$

The above is obvious if $\mathcal{A}(t, \varrho) = t$. Therefore, suppose $\mathcal{A}(t, \varrho) = \xi(r)$ for some substitution ξ, where $l \to r$ is a rule in the *fixed* rewrite tuple ϱ. Then, since the number of symbols in r is a constant, independent of the number of symbols in t, and since each symbol in the matching ξ cannot be bound to a shape term larger than $\|t\|$ (because $\xi(l) = t$), the number of nodes in $\xi(r)$ will also be $O(\|t\|)$.

Because $\textsc{one-pass-eval}(s) = \mathcal{A}(t, \varrho)$, we thus obtain

$$\|\textsc{one-pass-eval}(s)\| = O(\|s\|) \tag{5.51}$$

from Equations (5.49) and (5.50). Since the claim is obvious when s is atomic, we can therefore conclude that $\|\textsc{one-pass-eval}(s)\| = O(\|s\|)$ for any shape term s. □

Lemma 9 results in the following simplification of Equation (5.47):

$$\mathcal{T}\big(\textsc{squash}(\textit{\textbf{build}}(\dot{f}_n, \mathit{tuple}'))\big) = \sum_{i=1}^{n} O(\|s_i\|) = O(\|s\|). \tag{5.52}$$

Appealing to Lemmas 3 and 9, we also have that $\textit{\textbf{length}}(\mathit{tuple}') = O(\|s\|)$ on Line 13. Since any two shape terms in tuple' can be compared in $O(R\|s\|)$ time (see § 5.5.4.1),

$$\mathcal{T}(\textit{\textbf{sort}}(\sqsubseteq, \mathit{tuple}')) = O(R\|s\|^2 \log \|s\|). \tag{5.53}$$

Because $\textsc{unsquash}(\dot{f}_n, \mathit{tuple}')$ on Line 15 takes $O(\textit{\textbf{length}}(\mathit{tuple}'))$ time, we thus have

$$\mathcal{T}(\textsc{unsquash}(\dot{f}_n, \mathit{tuple}')) = O(\|s\|). \tag{5.54}$$

The rewrite tuple ϱ used on Line 17 is fixed. Therefore, Equation (5.46) gives us

$$\mathcal{T}(\mathcal{A}(t, \varrho)) = O(R\|t\| + B\|t\| + RB).$$

But from Equation (5.49), $\|t\| = O(\|s\|)$. Therefore

$$\mathcal{T}(\mathcal{A}(t, \varrho)) = O(R\|s\| + B\|s\| + RB). \tag{5.55}$$

Adding Equations (5.52) to (5.55) gives the following recurrence for a nonatomic s:

$$\begin{aligned}\mathcal{T}(\text{ONE-PASS-EVAL}(s)) = \sum_{i=1}^{n} &\mathcal{T}(\text{ONE-PASS-EVAL}(s_i)) \\ &+ O(R\|s\|^2 \log \|s\| + B\|s\| + RB).\end{aligned} \tag{5.56}$$

Since the base case runs in constant time and $B = O(\|s\|)$, we thus get

$$\mathcal{T}(\text{ONE-PASS-EVAL}(s)) = O(R\|s\|^3 \log \|s\|). \tag{5.57}$$

Chapter 6

Shape Tracking and Preallocation

However, new array programming computers are beginning to appear and, had they been standard 20 years ago, APL and not FORTRAN would have been the prototype of language development. I often wonder at what descriptive levels we would be programming today had that been the case!
— ALAN J. PERLIS
"In Praise of APL: a Language for Lyrical Programming,"
ACM SIGAPL APL Quote Quad, Vol. 8, Issue 2, December 1977. Page 47.

This chapter presents two applications of MULTI-PASS-EVAL for 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 can assume at run time.

6.1 Shape Tracking

Let $\mathtt{p}$ and $\mathtt{q}$ be variables defined in a MATLAB program at the assignment statements X and Y and let $\varphi(\mathtt{p})$ and $\varphi(\mathtt{q})$ be elements in Λ that denote their shapes by virtue of these assignments. We say that the shape of $\mathtt{q}$ *tracks* the shape of $\mathtt{p}$ if

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

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 [76]. A ϕ-function assignment is a pseudo operation located at a join node that merges different reaching definitions of a variable into a new definition. ϕ functions are an artificial device used in the conversion of a CFG into the Static Single-Assignment (SSA) form [29]. Note that the above 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[1] 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.

6.1.1 Empowering Optimizations

Tracked shapes can render some or all of the subtasks in an operation's preprocessing phase (see § 2.1 on page 12) 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 boils down to testing whether $\varphi(\mathtt{q})$ is $\overline{\boldsymbol{\pi}}$, 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 circumvented since p's storage is available for reuse at Y. Storage reuse is discussed further in Chapter 9.

Finally, it may even be possible to simplify or eliminate the dispatch logic (see Subtask 4 on page 12). As an example, suppose that the assignment statement Y is

[1]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 [76].

$$\texttt{q} \leftarrow \texttt{a+b}$$

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

6.1.2 Mapping MATLAB Expressions to Their Shapes

As indicated on page 84, φ is a *shape map* that associates MATLAB variables with their shape expressions in Λ. Like substitutions (see § 5.2 on page 51), φ can be homomorphically extended to a function from $\mathbb{M}$ to Λ, where $\mathbb{M}$ is the syntactic set of MATLAB expressions. That is, if f_n is an n-ary Type I MATLAB function, then

$$\varphi(f_n(e_1, e_2, \ldots, e_n)) = \dot{f}_n(\varphi(e_1), \varphi(e_2), \ldots, \varphi(e_n)), \tag{6.1}$$

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

6.1.3 Mapping ϕ Functions to Their Shapes

What should $\varphi(e)$ be when e is a ϕ-function expression of the form $\phi(e_1, e_2, \ldots, e_m)$? To answer this question, we observe that when the ϕ-function assignment $\texttt{c} \leftarrow \phi(e_1, e_2, \ldots, e_m)$ is executed, the variable `c` will be set to one of the m expressions e_i. m is the number of immediate predecessor nodes to the join node housing the ϕ-function assignment [29]. Exactly which e_i is assigned to `c` depends on the edge chosen to reach the join node at run time. Thus, a ϕ function isn't a mathematical function because different instances of $\phi(e_1, e_2, \ldots, e_m)$ may not yield the same e_i.

6.1.3.1 Gated ϕ Functions

The fact that ϕ functions aren't mappings in the mathematical sense can be remedied by *gating* them with an integer-valued run-time selector P:

$$\mathsf{c} \leftarrow \phi(P, e_1, e_2, \ldots, e_m). \tag{6.2}$$

Every join node in a CFG can be considered as having its own selector. If it has m predecessors, the selector assumes a value from 1 to m designating the particular predecessor from which control flows into it at run time. This value may change in the course of execution as join nodes get revisited from different predecessors.

6.1.3.2 Currying Gated ϕ Functions

Therefore, while a gated ϕ function can be viewed as a virtual MATLAB operation, it isn't of the Type I kind because just 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 ϕ function's outcome. The value assumed by the run-time selector P would also have to be known to make that determination. However, if we were to *curry* [74] the ϕ function on P, we would then obtain a pseudo language operator that has Type I characteristics:

$$\Phi(P)(e_1, e_2, \ldots, e_m) = \phi(P, e_1, e_2, \ldots, e_m). \tag{6.3}$$

Since each e_i in Equation (6.3) 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. Φ itself is a *higher-order function* [74] 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 $\Phi(P)$, and that gives us the following characterization for the shape of c in $\mathsf{c} \leftarrow \phi(P, e_1, e_2, \ldots, e_m)$:

$$\varphi(\mathsf{c}) = \dot{\Phi}(P)(\varphi(e_1), \varphi(e_2), \ldots, \varphi(e_m)). \tag{6.4}$$

6.1.4 The Shape-Tracking Algorithm

Figure 6.1 shows the shape-tracking algorithm. It makes use of five new primitives, explained below, in addition to those listed in Table 5.1 on page 59.

- *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 left-hand side of the assignment $astmt$.
- *DF*($astmt_2$) is the set of statements that form the dominance frontier of $astmt_2$.
- *SDOM*($astmt_2$) is the set of statements strictly dominated by $astmt_2$.

SHAPE-TRACKING 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 φ of variables to shapes; (3) the construction 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 conversion to the SSA form is performed so that the evolution of a variable's shape at its definition point (as a function of the shapes of the variables that occur in the defining MATLAB expression) also describes the evolution of the variable's shape across the entire program. FORM-SHAPE-MAP captures this evolution by trying to determine the "ultimate" shape of a variable—informally, its shape after taking into consideration the shapes of all variables directly and indirectly used by its defining MATLAB expression, and evaluating the resulting shape expression into some simplified, canonical form. 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, i.e., the substitution of a variable's shape by the shape of its defining MATLAB expression, the substitution of the shapes of variables that arise in the resulting shape expression by the shapes of their defining MATLAB expressions, and so on;
2. the evaluator's ability to detect an underlying, simpler canonical shape.

```
SHAPE-TRACKING(G)
   ☛ G is a CFG.
   G' ← SSA(G)
   FORM-SHAPE-MAP(G')
   ☛ After this point, a map of variables
   ☛ to shapes is available in φ.
   foreach astmt₁ in assignments(G') do
       q ← lhs(astmt₁)
       trackedby(q) ← q
       foreach astmt₂ in assignments(G') do
           p ← lhs(astmt₂)
           if astmt₁ ∈ DF(astmt₂) ∧ φ(q) = φ(p)
              trackedby(q) ← p
              break
           if astmt₁ ∈ SDOM(astmt₂) ∧ φ(q) = φ(p)
              trackedby(q) ← p
              break
```

Fig. 6.1 The SHAPE-TRACKING Procedure

```
FORM-SHAPE-MAP(G')
   ☛ G' is a CFG in the SSA form.
   G'' ← DFS(entry, G', forward_substitute)
   ☛ DFS(w, G', h) returns a processed
   ☛ version of G', obtained by running
   ☛ a depth-first search starting at node
   ☛ w in G' and invoking h(v, G') as
   ☛ each node v in G' is traversed.

   ☛ forward_substitute(v, G') visits
   ☛ assignments in the node (basic block)
   ☛ v in their lexical order and substitutes
   ☛ all forward uses of their left-hand
   ☛ sides, both in v as well as in the
   ☛ other basic blocks of the CFG G', by
   ☛ their respective right-hand sides.
   foreach astmt in assignments(G'') do
       c ← lhs(astmt)
       e ← rhs(astmt)
       φ(c) ← MULTI-PASS-EVAL(φ(e))
```

Fig. 6.2 The FORM-SHAPE-MAP Procedure

FORM-SHAPE-MAP begins by forward substituting all assignments in G' to obtain a new CFG G''. This is done in depth-first order (a breadth-first order could have also been used) starting from node entry so that all variables in the right-hand side e of an assignment $\mathtt{c} \leftarrow e$ are forward substituted to e', which is then substituted for all forward uses of c. Specifically, $\mathit{DFS}(\mathsf{entry}, G', \mathit{forward\_substitute})$ runs a depth-first search on G' by starting at node entry and invoking $\mathit{forward\_substitute}(v, G')$ on each node (basic block) v encountered along the way. The $\mathit{forward\_substitute}(v, G')$ call visits the assignments in v in their lexical order and substitutes all *forward* uses of their left-hand sides, both in v as well as in the other basic blocks of the CFG G', by their respective right-hand sides. Forward substitution is a standard compiler technique that inverts the effect of common subexpression elimination [76].

FORM-SHAPE-MAP then loops through each assignment *astmt* in G'', extracting its left-hand side c and right-hand side e. The loop updates the shape map φ at c by determining the shape expression of e (i.e., $\varphi(e)$), evaluating it (i.e., MULTI-PASS-EVAL($\varphi(e)$)), and assigning the result to $\varphi(\mathtt{c})$.

6.1.5 Confirming Inferences 1 and 2 in § 1.3

To demonstrate the workings of the SHAPE-TRACKING algorithm, we begin with the example in Figure 1.2 on page 5. The *pruned* SSA form [29] for this code fragment and its CFG are exhibited in Figure 6.3.[2] Shaded rectangles depict the basic blocks in the pruned SSA form. The rectangle containing Line 4 and the rectangle containing Line 8 correspond to CFG join nodes. Into these nodes are inserted the highlighted ϕ-function assignments to $\mathtt{a}_2$, $\mathtt{b}_1$, $\mathtt{e}_1$ and $\mathtt{e}_2$. The three variables $\mathtt{a}_0$, $\mathtt{b}_0$ and $\mathtt{e}_0$ are assumed live at node entry of the CFG.

Figure 6.4 shows the assignments after running a forward substitution, i.e., after Line 1 in FORM-SHAPE-MAP. Figure 6.5 displays the unevaluated shape expressions after applying φ on the right-hand sides in Figure 6.4 in accordance with Equation (6.1). Figure 6.6 shows the results of applying MULTI-PASS-EVAL on these un-

[2] A pruned SSA form forgoes ϕ-function assignments to variables that aren't subsequently used.

```
1    d_1 ← max(b_0*b_0, abs(a_0)*b_0);
2    a_1 ← atan2(b_0, b_0');
3    d_2 ← (b_0+d_1)*fix(b_0);

     a_2 ← Φ(P)(a_1, a_3);
     b_1 ← Φ(P)(b_0, b_2);
     e_1 ← Φ(P)(e_0, e_2);
4    while p_1,

5        c ← a_2*b_1;
6        a_3 ← c+a_2.^b_1;
7        b_2 ← c-a_3./c;

         e_2 ← Φ(Q)(e_1, e_3);
8        while p_2,

9            e_3 ← e_2*d_2;

         end;
     end;

10   h ← b_1.^b_1-a_2;
```

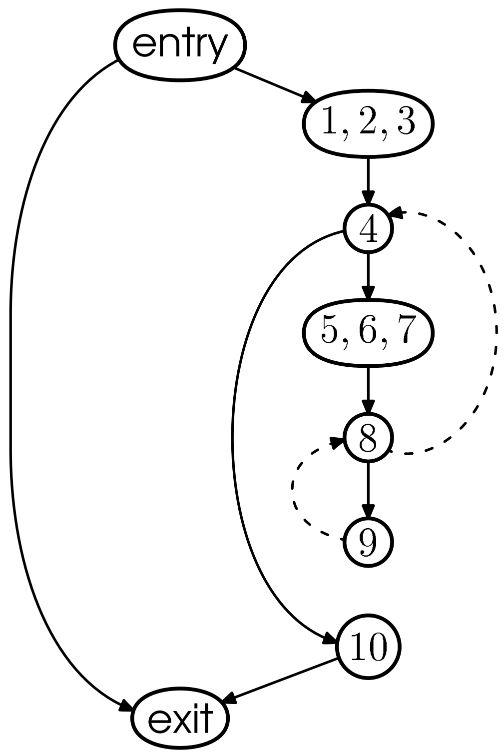

Fig. 6.3 Pruned SSA Form of Code in Figure 1.2 (with Shaded Rectangles Depicting Basic Blocks) and its CFG

```
d1 ← max(b0*b0, abs(a0)*b0);
a1 ← atan2(b0, b0');
d2 ← (b0+max(b0*b0, abs(a0)*b0))*fix(b0);
a2 ← Φ(P)(atan2(b0, b0'), a3);
b1 ← Φ(P)(b0, b2);
e1 ← Φ(P)(e0, e2);
c  ← Φ(P)(atan2(b0, b0'), a3)*Φ(P)(b0, b2);
a3 ← Φ(P)(atan2(b0, b0'), a3)*Φ(P)(b0, b2)
         +Φ(P)(atan2(b0, b0'), a3).^Φ(P)(b0, b2);
b2 ← Φ(P)(atan2(b0, b0'), a3)*Φ(P)(b0, b2)
         -(((Φ(P)(atan2(b0, b0'), a3)*Φ(P)(b0, b2))
               +(Φ(P)(atan2(b0, b0'), a3).^Φ(P)(b0, b2)))
            ./(Φ(P)(atan2(b0, b0'), a3)*Φ(P)(b0, b2)));
e2 ← Φ(Q)(Φ(P)(e0, e2), e3);
e3 ← Φ(Q)(Φ(P)(e0, e2), e3)
         *((b0+max(b0*b0, abs(a0)*b0))*fix(b0));
h  ← Φ(P)(b0, b2).^Φ(P)(b0, b2)-Φ(P)(atan2(b0, b0'), a3);
```

Fig. 6.4 After Forward Substitution on the SSA Code in Figure 6.3

evaluated shape expressions.[3] We observe that the results are mostly simpler when compared to the shape expressions in Figure 6.5, 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 (highlighted in boldface colors in Figure 6.6). Because $\mathtt{d}_1$ and $\mathtt{a}_3$ strictly dominate $\mathtt{d}_2$ and $\mathtt{b}_2$ respectively, Line 12 in SHAPE-TRACKING 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 § 1.3.

6.1.6 Confirming Inferences 5 and 6 in § 2.3.1.1

The `quadrature` function is already in the SSA form. Only `a` and `b` are live at its entry. Thus, by following the same steps described in § 6.1.5, we can apply

[3] This was done with an implementation of MULTI-PASS-EVAL that used Mathematica's canonical expression ordering [99, page 1031] as the total order $\sqsubseteq$. Employing a slightly different MATLAB fragment as the example, [53] discusses how Mathematica can be programmed for this evaluation.

$$
\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))\\
&\qquad\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. 6.5 Shape Expressions for the Right-Hand Sides in Figure 6.4

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

Fig. 6.6 After MULTI-PASS-EVAL on the Shape Expressions in Figure 6.5

SHAPE-TRACKING on `quadrature`'s CFG to obtain the following shape map:

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

Since the images of $\varphi(\mathtt{h})$, $\varphi(\mathtt{mid})$, $\varphi(\mathtt{Fa})$, $\varphi(\mathtt{Fmid})$ 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 on Line 2 in `quadrature` (Figure 2.4 on page 30) is well-defined, the computations up to Line 6 will also be well-defined. However, because $s\dot{*}s$ is $\boldsymbol{\pi}$ only when s isn't a square shape, $\varphi(\mathtt{S})$ on Line 7 is well-defined only when `a` and `b` are square matrices. Hence the claim in Inference 6.

6.1.7 Noncircular Inferences

A factor (one of two mentioned on page 88) that affects the quality of the shape map is whether a shape gets propagated from its origination point to a point that directly or indirectly consumes it. By resorting to forward substitution, FORM-SHAPE-MAP effectively ignores the propagation of shape information along CFG back edges.

Due to the nature of control flow in the MATLAB language (conditional statements, structured loops, and no `goto` statements), ϕ functions with only two arguments are needed at join nodes in a MATLAB CFG. The definitions that reach the ϕ function's arguments will always be from lexically preceding points, except for ϕ functions located at loop-header join nodes for which one definition will be lexically preceding and the other will be lexically succeeding.[4] Hence, a forward substitution on a MATLAB CFG essentially overlooks the propagation of shape information into

[4]Our implementation in MAGICA distinguishes these two kinds of ϕ functions by using a $\mathtt{phi}\mu$ Mathematica object for the former and a $\mathtt{phi}\nu$ Mathematica object for the latter [52].

one of the two arguments of a loop-header ϕ function. The next section presents a method that accounts for shape information along CFG back edges so as to give a fuller picture on how a variable's shape evolves in the course of program execution.

6.2 Preallocation

Let 2^{Λ} be the power set of Λ. Our approach to preallocation is based on the definitions of a lattice $\mathcal{L}_{2^{\Lambda}}$ on 2^{Λ} and a function space $\breve{\mathcal{F}}$ that consists of operators monotonic on $\mathcal{L}_{2^{\Lambda}}$. Members in $\breve{\mathcal{F}}$ are systematically formulated by using a procedure called SET-OF-SHAPES-EVAL. SET-OF-SHAPES-EVAL is an extension of MULTI-PASS-EVAL to *sets* of shape terms. That is, SET-OF-SHAPES-EVAL$(\{s_1, s_2, \ldots, s_m\})$ is a "simplified" version of $\{s_1, s_2, \ldots, s_m\}$, where each s_j $(1 \leq j \leq m)$ is a shape term in Λ. The extension operates in two steps:

1. it first simplifies each s_j in $\{s_1, s_2, \ldots, s_m\}$ by using MULTI-PASS-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, i.e., identities that exist between entire sets of shape terms.

6.2.1 The Set-of-Shapes Evaluator

Like the identities in § 4.7 that describe properties in the semantic domain $\mathbb{S}_{\wp}$, identities in the power set $2^{\mathbb{S}_{\wp}}$ can also be characterized. An example is

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

The above identity, proved in Lemma 10 below, follows from the fact that for a given pair of shape-tuple class expressions $\overline{\boldsymbol{s}}$ and $\overline{\boldsymbol{t}}$ in $\mathbb{S}_{\wp}$, $\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$ will always be either $\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}$ or $\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$. Another set-of-shapes identity is stated and proved in Lemma 11.

Lemma 10

$$\{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\} = \{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\}. \tag{6.5}$$

Proof. Obviously,

$$\{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\} \subseteq \{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\}. \tag{6.6}$$

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

$$\{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\} \subseteq \{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\} \tag{6.7}$$

is always true. Equation (6.5) follows from Equations (6.6) and (6.7). □

Lemma 11

$$\{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\} = \{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\}. \tag{6.8}$$

Proof. Once again,

$$\{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\} \subseteq \{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\}. \tag{6.9}$$

If $\overline{\boldsymbol{s}} = \overline{\boldsymbol{\iota}}$, or $\overline{\boldsymbol{t}} = \overline{\boldsymbol{\iota}}$, or $\overline{\boldsymbol{s}} = \overline{\boldsymbol{t}}$, then $\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}$ is obviously either $\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}$ or $\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$. Therefore, let $\overline{\boldsymbol{s}} \neq \overline{\boldsymbol{\iota}}$, $\overline{\boldsymbol{t}} \neq \overline{\boldsymbol{\iota}}$ and $\overline{\boldsymbol{s}} \neq \overline{\boldsymbol{t}}$. Then $\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}$ will be $\overline{\boldsymbol{\pi}}$. But then, both $\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}$ and $\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}$ will also be $\overline{\boldsymbol{\pi}}$. Hence

$$\{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{t}}\dot{+}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\} \subseteq \{\overline{\boldsymbol{s}}\dot{*}\overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}, \overline{\boldsymbol{s}}\dot{+}\overline{\boldsymbol{t}}\dot{*}\overline{\boldsymbol{t}}\} \tag{6.10}$$

is always true. Equation (6.8) follows from Equations (6.9) and (6.10). □

Figure 6.7 shows how the SET-OF-SHAPES-EVAL procedure uses these set-of-shapes identities. The transformer $\mathcal{A}^\dagger$ on Line 7 is as defined below:

$$\mathcal{A}^\dagger(S, (L_1 \to R_1, L_2 \to R_2, \ldots, L_k \to R_k)) =$$
$$\begin{cases} (S - \xi(L_1)) \cup \xi(R_1) & \text{if } \exists \xi \in \Xi \text{ such that } \xi(L_1) \subseteq S, \\ S & \text{else if } k = 1, \\ \mathcal{A}^\dagger(S, (L_2 \to R_2, \ldots, L_k \to R_k)) & \text{otherwise.} \end{cases} \tag{6.11}$$

```
SET-OF-SHAPES-EVAL({s_1, s_2, ..., s_m})
    ☛ Each s_j (1 ≤ j ≤ m) is a shape term in Λ.
    S' ← ∅
    for j ← 1 to m do
        s'_j ← MULTI-PASS-EVAL(s_j)
        S' ← S' ∪ s'_j
    do
        S'' ← S'
        S' ← A†(S', ϱ†)
        ☛ ϱ† is a fixed rewrite tuple of the
        ☛ form (L_1 → R_1, L_2 → R_2, ..., L_γ → R_γ),
        ☛ where each L_i → R_i is an ordered pair
        ☛ in 2^Λ × 2^Λ.
    until S' = S''
    return S'
```

Fig. 6.7 The SET-OF-SHAPES-EVAL Procedure

Each rule $L_i \to R_i$ in Equation (6.11) 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 (6.11) after being homomorphically extended to mappings from 2^{Λ} to 2^{Λ}:

$$\xi(\{s_1, s_2, \ldots, s_m\}) = \{\xi(s_1), \xi(s_2), \ldots, \xi(s_m)\}. \tag{6.12}$$

Thus, the transformer $\mathcal{A}^{\dagger}$ is similar to the rules applicator in Equation (5.6) except that subsets are matched in Equation (6.11) rather than individual shape terms as in Equation (5.6). It is used with a fixed tuple of rules $\varrho^{\dagger}$ in SET-OF-SHAPES-EVAL.

Since $\mathcal{A}^{\dagger}$ is many-valued like the rules applicator in Equation (5.6), we shall write $T \in \mathcal{A}^{\dagger}(S, \tau)$ if T is *an* image of $\mathcal{A}^{\dagger}(S, \tau)$ by Equation (6.11). Thus, if

$$\mathcal{B}(\{S_1, S_2, \ldots\}, \tau) = \bigcup_{i \geq 1} \mathcal{A}^{\dagger}(S_i, \tau), \tag{6.13}$$

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

6.2.1.1 Strong Normalization

Consider a relation $\rhd$ on 2^{Λ} 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,

$$S \rhd T \Leftrightarrow (S \neq T) \wedge (\exists j > 0 \text{ such that } T \in \mathcal{B}^{j}(\{S\}, \varrho^{\dagger})), \tag{6.14}$$

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

$$\mathcal{B}^{i}(\mathbf{S}, \varrho^{\dagger}) = \begin{cases} \mathcal{B}(\mathcal{B}^{i-1}(\mathbf{S}, \varrho^{\dagger}), \varrho^{\dagger}) & \text{if } i > 0, \\ \mathbf{S} & \text{if } i = 0. \end{cases} \tag{6.15}$$

If $\rhd$ is well-founded, then SET-OF-SHAPES-EVAL can also be shown to terminate by arguments similar to the proof for Theorem 3. Under what conditions is $\rhd$ well-founded? One such condition on $\varrho^{\dagger}$ is given in Lemma 12 below.

Lemma 12 *If every rule $L \rightarrow R$ in $\varrho^{\dagger}$ is such that either $L = R$ or $|L| > |R|$, then $\rhd$ will be well-founded.*[5]

PROOF. Suppose $V \in \mathcal{A}^{\dagger}(U, \varrho^{\dagger})$ and let $U \neq V$. Then from Equation (6.11),

$$V = (U - \xi(L_j)) \cup \xi(R_j) \tag{6.16}$$

for some $L_j \rightarrow R_j$ in $\varrho^{\dagger}$ and $\xi \in \Xi$. Furthermore, $L_j \neq R_j$ since otherwise U would have been the same as V (because $\xi(L_j) = \xi(R_j)$ for all $\xi \in \Xi$ when $L_j = R_j$). Thus from the theorem's hypothesis, $|L_j| > |R_j|$. But from Equation (6.12), $|\xi(L)| > |\xi(R)|$ for all $\xi \in \Xi$ when $|L| > |R|$. From Equation (6.16), we therefore get

$$|U| > |V|.$$

Hence, $S \rhd T$ implies $|S| > |T|$. Since $|T| \geq 0$ and since $>$ is a well-founded relation on the set of nonnegative integers, it follows that $\rhd$ is also well-founded. $\square$

[5] Vertical bars are used to denote the cardinalities of sets. Recollect from § 4.1 that they also express determinants—for instance, $|\boldsymbol{s}|$. The different usages will be clear from the context.

As an example, if the rules that formed $\varrho^\dagger$ in SET-OF-SHAPES-EVAL were

$$\{s\dot{*}s\dotplus t\dot{*}t, s\dot{*}s\dotplus t, s\dotplus t\dot{*}t\} \to \{s\dot{*}s\dotplus t, s\dotplus t\dot{*}t\}, \tag{6.17}$$

$$\{(s\dot{*}t\dotplus s)\dotplus t, s\dot{*}s\dotplus t, s\dotplus t\dot{*}t\} \to \{s\dot{*}s\dotplus t, s\dotplus t\dot{*}t\}, \tag{6.18}$$

$$\{s\dot{*}t\dotplus(s\dotplus t), s\dot{*}s\dotplus t, s\dotplus t\dot{*}t\} \to \{s\dot{*}s\dotplus t, s\dotplus t\dot{*}t\}, \tag{6.19}$$

which correspond to the identities in Equations (6.5) and (6.8),[6] then SET-OF-SHAPES-EVAL would have been strongly normalizing as a consequence of Lemma 12. It would have also been strongly normalizing if $\varrho^\dagger$ were defined as follows:

$$\varrho^\dagger = (\{s\} \to \{s\}). \tag{6.20}$$

6.2.1.2 Unique Normalization

Although SET-OF-SHAPES-EVAL(S) as defined in Figure 6.7 isn't assured to produce the same result for a given 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 [33]; or by (2) using techniques that are independent of the particular rewrite tuple used but that give the impression of uniqueness by doing away with the nondeterminism in Equation (6.11). An example of the latter is imposing a total order on Ξ and always choosing the "least" among candidate substitutions that achieve a subset match. Of course, while such techniques are limited in that they could overlook useful solutions, they won't impact situations where only one normal form exists for a solution.

For instance, suppose $\varrho^\dagger$ consisted of the rules in Equations (6.17) to (6.19). Then given $S = \{(u\dot{*}u)*(u\dot{*}u)\dotplus v\dot{*}v, (u\dot{*}u)*(u\dot{*}u)\dotplus v, u\dot{*}u\dotplus v\dot{*}v, u\dot{*}u\dotplus v, u\dotplus v\dot{*}v\}$, there are two solutions to $\mathcal{A}^\dagger(S, \varrho^\dagger)$ by Equation (6.11):

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

[6] Two rules are needed for the identity in Equation (6.8) due to the possible associations of $\dotplus$.

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 have ultimately produced either of two solutions:

$$\begin{tikzcd}[column sep=small]
 & \{(u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v, u\dot{*}u\dot{+}v\dot{*}v, u\dot{*}u\dot{+}v, u\dot{+}v\dot{*}v\} \arrow[dr] & \\
\textsc{set-of-shapes-eval}(S) \arrow[ur] \arrow[dr] & & \{(u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v, u\dot{*}u\dot{+}v, u\dot{+}v\dot{*}v\} \\
 & \{(u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v\dot{*}v, (u\dot{*}u)\dot{*}(u\dot{*}u)\dot{+}v, u\dot{*}u\dot{+}v, u\dot{+}v\dot{*}v\} &
\end{tikzcd}$$

However, if the $\varrho^\dagger$ defined in Equation (6.20) were used, then SET-OF-SHAPES-EVAL would trivially uniquely normalize because $\mathcal{A}^\dagger$ would then be single-valued.

6.2.2 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 (including the $\Phi(P)$ function), a *set-of-shapes* operator $\breve{f}_n : (2^\Lambda)^n \mapsto 2^\Lambda$ can be defined such that

$$\breve{f}_n(S_1, S_2, \ldots, S_n) = \begin{cases} \textsc{set-of-shapes-eval}(\bigcup_{i=1}^n S_i) & \text{if } \dot{f}_n = \dot{\Phi}(P), \\ \textsc{set-of-shapes-eval}(\{t \mid \exists s_i \in S_i \text{ for all} & \text{if } \dot{f}_n \neq \dot{\Phi}(P). \\ 1 \le i \le n \text{ such that } t = \dot{f}_n(s_1, s_2, \ldots, s_n)\}) & \end{cases} \tag{6.21}$$

Let $\breve{\mathcal{F}}$ be the set of all $\breve{f}_n$ operators defined by Equation (6.21) for various n. Then $\breve{\mathcal{F}}$ forms a *set-of-shapes* algebraic structure $[2^\Lambda, \breve{\mathcal{F}}]$ 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. $[2^\Lambda, \breve{\mathcal{F}}]$ is therefore 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 § 5.

Instances of set-of-shapes operators are $\breve{+}$, $\breve{*}$ and $\breve{\Phi}(P)$, which respectively correspond to the $\dot{+}$, $\dot{*}$ and $\dot{\Phi}(P)$ shape operators. They are demonstrated below:

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

6.2.2.1 Mitigating Exponential Blowup

From Equation (6.21), 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 MULTI-PASS-EVAL on Line 3 in SET-OF-SHAPES-EVAL may produce the same outcome on distinct shape terms; and (2) the set-of-shapes identities exercised on Line 7 in SET-OF-SHAPES-EVAL may simplify the resulting sets in their entirety. An example where a set-of-shapes identity causes a reduction is

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

The above simplification happens on account of the identity in Equation (6.5).

6.2.2.2 Monotone Set-of-Shapes Operators

Consider a lattice $\mathfrak{L}_{2^\Lambda}$ defined on 2^Λ in which the subset relation $\subseteq$ is the partial order, the empty set $\emptyset$ and Λ are the least and greatest elements, and set union $\cup$ and set intersection $\cap$ are the join and meet operations. From Equation (6.21), we see that $\breve{f}_n$ is monotonic on $\mathfrak{L}_{2^\Lambda}$ *if* SET-OF-SHAPES-EVAL is monotonic on $\subseteq$.

Let X and Y be the values of S' at the end of Line 4 when SET-OF-SHAPES-EVAL is invoked on S and T respectively. Then clearly, $X \subseteq Y$ if $S \subseteq T$. Thus the monotonicity of SET-OF-SHAPES-EVAL really hinges on what $\varrho^\dagger$ is. An obvious choice is given in Equation (6.20) because $U = \mathcal{A}^\dagger(U, (\{s\} \to \{s\}))$ for all $U \in 2^\Lambda$. We will assume this to be the $\varrho^\dagger$ in SET-OF-SHAPES-EVAL for the rest of this thesis. Since the function space $\breve{\mathcal{F}}$ on $\mathfrak{L}_{2^\Lambda}$ will then comprise monotone members, a fixed-point solution will *always* exist for a system of equations formed on the 2^Λ domain using the set-of-shapes operators in $\breve{\mathscr{F}}$. While monotonicity guarantees convergence to a solution, lattice-theoretic algorithms that seek such a solution aren't assured to terminate because $\mathfrak{L}_{2^\Lambda}$ has an infinite height; an indefinite number of iterations may therefore be necessary before a quiescent solution is reached.

6.2.3 An Iterative Forward Data-Flow Algorithm

Consider a function ς such that $\varsigma(\mathtt{c})$ is an element in 2^Λ that denotes the set of all possible shapes taken on by a MATLAB variable $\mathtt{c}$ across an entire program. Like φ, ς can be homomorphically extended to a function from $\mathbb{M}$ to 2^Λ so that

$$\varsigma(f_n(e_1, e_2, \ldots, e_n)) = \breve{f}_n(\varsigma(e_1), \varsigma(e_2), \ldots, \varsigma(e_n)), \tag{6.22}$$

where $\breve{f}_n$ is the n-ary set-of-shapes operator in $\breve{\mathscr{F}}$ corresponding to f_n. For example, $\varsigma(\mathtt{a} * \mathtt{b}. * \mathtt{c})$ is $\varsigma(\mathtt{a})\breve{*}\varsigma(\mathtt{b})\breve{\dot{+}}\varsigma(\mathtt{c})$,[7] and $\varsigma(\Phi(P)(e_1, e_2))$ is $\breve{\Phi}(P)(\varsigma(e_1), \varsigma(e_2))$.

The FORM-SET-OF-SHAPES-MAP algorithm, shown in Figure 6.8, takes a CFG G' in SSA form and builds the ς map of the MATLAB variables in it. It effectively constructs a set-of-shapes equation $\varsigma(\mathtt{c}) = \varsigma(e)$ for every assignment $\mathtt{c} \leftarrow e$ and iteratively solves the system up to *maxiters* times until a stationary solution is achieved. It initially sets $\varsigma(\mathtt{c})$ of every variable $\mathtt{c}$ 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 $\mathtt{c}$ however is live on entry, then $\varsigma(\mathtt{c})$ is set to $\{\varphi(\mathtt{c})\}$. The solution process then iteratively forward propagates the set-of-shapes information, capping the number of iterations at the predefined constant *maxiters* on account of the lattice's infinite height.

6.2.4 Confirming Inference 3 in § 1.3

The CFG in SSA form for the code in Figure 1.2 was shown in Figure 6.3. Since $\mathtt{a}_0$, $\mathtt{b}_0$ and $\mathtt{e}_0$ are live on entry, FORM-SET-OF-SHAPES-MAP initializes their images

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

```
FORM-SET-OF-SHAPES-MAP(G′)
    ☛ G′ is a CFG in the SSA form.
    foreach v in nodes(G′)
        foreach c in used_vars(v)
            ς(c) ← {φ(c)}
    foreach v in nodes(G′)
        foreach c in defined_vars(v)
            ς(c) ← ∅
    ☛ At this point, ς is initialized so that if a
    ☛ variable c is only used in the CFG but
    ☛ defined elsewhere, then ς(c) is set to
    ☛ {φ(c)}. If c is however defined in the
    ☛ CFG, then ς(c) is set to the empty set ∅.
    iters ← 0
    do
        fixedpoint ← true
        foreach astmt in assignments(G′) do
            c ← lhs(astmt)
            e ← rhs(astmt)
            if ς(c) ≠ ς(e)
                fixedpoint ← false
            ς(c) ← ς(e)
        iters ← iters + 1
    while (fixedpoint = false ∧ iters ≤ maxiters)
    ☛ maxiters caps the number of iterations
    ☛ because 𝔏_{2^Λ} is an infinite height lattice.
    if fixedpoint = false
        ☛ Fixed point not attained; force solution
        ☛ to the topmost element of the lattice.
        foreach v in nodes(G′)
            foreach c in defined_vars(v)
                ς(c) ← Λ
```

Fig. 6.8 The FORM-SET-OF-SHAPES-MAP Procedure

$$
\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. 6.9 Set-of-Shapes Map for the SSA Code in Figure 6.3

under ς to $\{\varphi(\mathtt{a}_0)\}$, $\{\varphi(\mathtt{b}_0)\}$ and $\{\varphi(\mathtt{e}_0)\}$ respectively. For every other variable, the image under ς is initially set to $\emptyset$. At the conclusion of FORM-SET-OF-SHAPES-MAP, executed with *maxiters* equal to ∞ and $\varrho^\dagger$ set according to Equation (6.20), the set-of-shapes map shown in Figure 6.9 is obtained as the fixedpoint.[8] From this resulting map, we see that there can only be at most two shapes for `c`, one for $\mathtt{a}_3$ and $\mathtt{b}_2$, and three for $\mathtt{e}_3$ during the entire execution lifetime of the loops.

[8]See [53] for a discussion on how Mathematica can be used for this.

6.2.5 A Note on Backward Data Flow

An alternative solution process would be to propagate the set-of-shapes information backward, from the left-hand side to the right-hand side of a MATLAB assignment. Backward data-flow techniques consider the context in which the left-hand side `c` of an assignment `c` $\leftarrow$ e is *subsequently* used, establish what `c` should be to satisfy the type correctness constraints imposed by that context, and then use that information to determine, or refine, the information of interest for all the subexpressions or variables in e. For instance, array subscripting expressions in MATLAB have to be of an integral intrinsic type for the indexing operation to be type correct. Therefore, if we have the assignment `i` $\leftarrow$ `j+1`, where `i` is later used as an array index, then `i` would have to be an integer for that use to be type correct. By backward data flow, this would mean that `j` would also have to be an integer.

However, because languages like MATLAB and APL are dynamically typed, the premise that a variable's use will be type correct at run time may not always hold. Consequently, the validity of the information inferred by backward data flow is only assured if the dynamically typed program turns out to be type correct at execution. It is due to this reason that previous lattice-based approaches to type inference in MATLAB and APL have usually abstained from using backward data flow [17, 32].

Chapter 7

The Shape Inference Framework

The whole is more than the sum of its parts.
— ARISTOTLE
Metaphysica 10f-1045a (ca 330 B.C.).

From a shape inference perspective, it suffices to focus on only those language operators that are built directly into the MATLAB system. 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.

7.1 A 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 output is dependent on the shape 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 multiply operator, and elementwise operators like 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 at least one argument is known. An example is the colon operator [66]. 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, where a' and b' represent the run-time *real values* of `a` and `b` respectively. If $a' > b'$, `c` is the empty row vector. Thus, the shape of `c` is determinable only when the elements of `a` and `b` are known. This built-in isn't Type I because the mere knowledge of the shapes of `a` and `b` is insufficient when computing the shape of `c`.

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 shape of the result. For example, there exists a built-in called `dbstack` that returns the stack trace as a column vector [66]. A complete execution history may be necessary in this case to determine the shape of the result. Type III operators appear to be few and will not be discussed further in this thesis.

Table 7.1 shows some of MATLAB's built-ins grouped by the above classification system. Members of the Type I class appear to be in the majority in MATLAB; in fact, we have have been able to so far uncover nine quotient algebras to which are isomorphic the shape semantics of over 50 Type I built-ins [56]. 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 allow for the execution of arbitrary strings. However, just knowing what these argument strings are will not 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.

7.2 Extensions to Type II Built-in Functions

Though the framework described in this thesis explicitly addresses the Type I function group, it can be extended to handle Type II operators by using the technique

Type I
Operator
a*b
a+b
a-b
a.*b
a.^b
a./b
a.\b
a==b
a~=b
a<b
a>b
a<=b
a>=b
a&b
a|b
a/b
a\b
[a, b]
[a; b]
+a
-a
~a
c(:) $\leftarrow$ a
a^b
a(:)
a'
a.'
rand
length(a)

Type II
Operator
a:b
a(e)
permute(a, e)
cat(e, a, b)
a($e_1, e_2, \ldots, e_n$)
rand($e_1, e_2, \ldots, e_n$)
randn($e_1, e_2, \ldots, e_n$)
zeros($e_1, e_2, \ldots, e_n$)
ones($e_1, e_2, \ldots, e_n$)
eye($e_1, e_2, \ldots, e_n$)
c(e) $\leftarrow$ a
c($e_1, e_2, \ldots, e_n$) $\leftarrow$ a

Type III
Operator
dbstack
eval(a)
evalin(a, b)
feval(a, b)

Table 7.1 Shape-Based Classification of MATLAB's Built-in Functions

of function currying [74]. By this method, a shape operator of a Type II built-in is itself treated as a function of the nonshape arguments.

Let $\mathbb{C}$ be the set of complex numbers and let $\Delta = \bigcup_{k\geq 1} \mathbb{C}^k$ be the set of all tuples comprising complex numbers. Suppose g is an n-ary Type II built-in function in MATLAB. Then by definition of a Type II built-in, there exists a mapping $\grave{g}$ from $(\Delta \times \mathbb{S}_{\wp})^n$ to $\mathbb{S}_{\wp}$ that describes the shape semantics of g. By currying $\grave{g}$ on Δ^n, we can obtain a shape operator with Type I characteristics:

$$\grave{g}(\varepsilon_1, \overline{\boldsymbol{s_1}}, \varepsilon_2, \overline{\boldsymbol{s_2}}, \ldots, \varepsilon_n, \overline{\boldsymbol{s_n}}) = \vec{g}(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n)(\overline{\boldsymbol{s_1}}, \overline{\boldsymbol{s_2}}, \ldots, \overline{\boldsymbol{s_n}}). \tag{7.1}$$

Each ε_i in the above is a tuple in Δ. $\vec{g}$ is therefore a function with Δ as its domain, and the set $\dot{\mathcal{F}}$ of shape operators for Type I built-ins as its codomain.

For instance, consider the right-hand side array indexing operation in $\texttt{c} \leftarrow \texttt{a(}i_1\texttt{,}\ i_2\texttt{,}\ \texttt{...,}\ i_m\texttt{)}$, which is a Type II built-in (see Table 7.1). In functional form, this could be written as $\texttt{c} \leftarrow \hbar_m\texttt{(a,}\ i_1\texttt{,}\ i_2\texttt{,}\ \texttt{...,}\ i_m\texttt{)}$, where $\hbar_m$ represents the right-hand side array indexing operation with m subscripts. Thus, if $\chi : \mathbb{A} \mapsto \Delta$ associates a MATLAB array in $\mathbb{A}$ (see Figure 3.1 on page 35) with a tuple in Δ by linearizing the array in some fixed order, then the Type I shape operator $\dot{\hbar}_m$ corresponding to $\hbar_m$ would be

$$\dot{\hbar}_m = \vec{\hbar}_m(\chi(\texttt{a}), \chi(i_1), \chi(i_2), \ldots, \chi(i_m)). \tag{7.2}$$

Example applications of $\dot{\hbar}_m$ are given below:

$$\vec{\hbar}_2((-1, 7, 2.3, 0, 11), (1), (2))(\overline{\langle 1, 5\rangle}, \overline{\boldsymbol{\iota}}, \overline{\boldsymbol{\iota}}) = \overline{\boldsymbol{\iota}},$$
$$\vec{\hbar}_2((-1, 7, 2.3, 0, 11), (2), (2))(\overline{\langle 1, 5\rangle}, \overline{\boldsymbol{\iota}}, \overline{\boldsymbol{\iota}}) = \overline{\boldsymbol{\pi}},$$
$$\vec{\hbar}_3((-99, 10, 21, 15, -0.99, 101), (1, 2), (2, 3), (1))(\overline{\langle 2, 3\rangle}, \overline{\langle 1, 2\rangle}, \overline{\langle 2, 1\rangle}, \overline{\boldsymbol{\iota}}) = \overline{\langle 2, 2\rangle}.$$

Since keeping track of the entire contents of a tuple in Δ may be prohibitive due to its size, an approximation is to only track 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, and where each t_i represents a MATLAB expression. Thus, $\chi(t_i)$ is a placeholder for an element in Δ. If we had no more information about any $\chi(t_i)$, the above may be the most simplified form of the shape.[1] However, if we were to know their value ranges, then it may 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:

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

In the last example, the shape remains unsimplified since $[\![1.1, 4.78]\!]$ straddles 3.

7.2.1 Type I Compositions of Type II Built-in Functions

The composition of certain Type II built-ins with Type I built-ins exhibit overall Type I characteristics. For instance, an idiom seen in many MATLAB programs is

$$\texttt{b} \leftarrow \texttt{zeros(size(a))}$$

where, instead of `zeros`, we could have also had `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 1.4 on page 8). The `size` built-in is Type I because knowing the shape of `a` is enough to establish the shape of `size(a)`. As an instance, if the shape of `a` is $\overline{\langle 2, 4, 5\rangle}$, the shape of `b` is $\overline{\langle 1, 3\rangle}$. `zeros`, on the other hand, is a Type II built-in because the actual values of the arguments would have to be known to determine the shape of the result. For example, if `x` is a row vector with the elements 5, 3 and 7, `zeros(x)` creates an array having the shape $\overline{\langle 5, 3, 7\rangle}$. Thus, the composition `zeros(size(a))` will *always* return an array with the same shape as `a`; it therefore is Type I in character.

[1] To ensure that shape errors propagate through the rest of the program from the point of origination, the above shape would always simplify to $\overline{\boldsymbol{\pi}}$ if there were even one s_i that was $\overline{\boldsymbol{\pi}}$.

7.2.2 Confirming Inference 4 in § 1.4

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))` [66]. Hence, the array assigned to `rank` in the `hilarray_acc` user-defined function on page 8 is always a scalar. From the discussion in § 7.2.1, both `indices` and `sindices` will always have the same shape as `a`. Now for any MATLAB array `x`, the call `cumsum(x,` e`)` always returns an array of the same shape as `x`, as long as e is a positive scalar integer [66]. But if the loop in Figure 1.4 gets executed, `k` is guaranteed to be a positive scalar integer. Thus, the outcome of `cumsum(indices, k)` in Figure 1.4 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 `b` computed on the last line of `hilarray_acc` will always have the same shape as `a`.

Chapter 8

A Type Inference Engine

A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use. . . . Violating the type system involves removing the protective set of clothing and operating directly on the naked representation.
— LUCA CARDELLI and PETER WEGNER
"On Understanding Types, Data Abstraction, and Polymorphism,"
ACM Computing Surveys, Vol. 17, Issue 4, December 1985. Page 474.

This chapter briefly describes an extensible inference engine called MAGICA (MAthematica system for General-purpose Inferring and Compile-time Analyses) that can determine the types (value range, intrinsic type and array shape) of expressions in a MATLAB program. MAGICA is a prototype implementation in Mathematica of the techniques and algorithms in Chapters 5 and 6. It is designed as an add-on module that any MATLAB compiler infrastructure can use to obtain high-quality type inferences—"high quality" because besides attempting to return as precise an inference as possible, the system will detect shape equivalence in cases where traditional lattice-based approaches would return a conservative result (see § 2.3.1).

Lines `In[1]` and `Out[1]` below demonstrate a simple interaction with MAGICA through a notebook interface.[1] On line `In[1]`, the MAGICA `type` function object is

[1]The outputs in this chapter can be exactly reproduced by typing the code shown against each `In[n]:=` prompt into a notebook interface to version 1.0 of MAGICA, running on Mathematica 4.1.

applied on a representation of the MATLAB expression `tanh(3.78i)`. MAGICA's response, shown on *Out[1]*, is the inferred type of `tanh(3.78i)`. In this case, "type" is the expression $\{v, i, s\}$ where v, i and s are the value range, intrinsic type and array shape of `tanh(3.78i)`; *Out[1]* indicates these to be the point 0.742071 ⅈ, the `$nonreal` intrinsic type designator, and the two-dimensional array shape with unit extents along both dimensions—that is, the scalar shape.

```
In[1]:= type[tanh[3.78ⅈ]]
Out[1]= {0.742071 ⅈ, $nonreal, {⟨1 ,  1⟩, 2}}
```

8.1 Feature Support

The above is a type inference on a single MATLAB expression. MAGICA can infer the types of whole MATLAB programs comprising an arbitrary number of user-defined functions, each having an arbitrary number of statements. User-defined functions can return multiple values, can consist of assignments, the `for` and `while` loops, and the `if` conditional statement. (All these MATLAB constructs are explained in [66].) In addition, MAGICA can handle close to 70 built-in functions in MATLAB. These include important Type II operations like `subsref`, `subsasgn` and `colon` that are used in array indexing and colon expressions. For the most part, the full or nearly the full semantics of a built-in function, as specified in [66], is supported. For instance, subscripts in array indexing expressions can themselves be arrays, and arrays can be complex-valued. Not all of MATLAB's features are currently handled; these include structures, cell arrays and recent additions like function handles.

8.2 Representing MATLAB in MAGICA

MAGICA symbolically represents MATLAB constructs. As an example, the Mathematica expression `plus[a, b]` is MAGICA's representation of the MATLAB expression `a+b`. On line *In[1]* above, the Mathematica expression `tanh[3.78ⅈ]` denotes the MATLAB expression `tanh(3.78i)`. The idea of functionally representing a MATLAB expression can also be used to denote high-level constructs. For instance, the MATLAB assignment `l` $\leftarrow$ `cos(3.099)`, where `l` is a MATLAB program variable, is represented in MAGICA as shown on line *In[2]* below.

```
In[2]:= assignment[$$lhs :→ l, $$rhs :→ cos[3.099]]
Out[2]= assignment($$lhs :→ l, $$rhs :→ cos(3.099))
```

The expression's *head* is `assignment` and this is used to uniquely identify MATLAB assignments. The *tags* `$$lhs` and `$$rhs` serve to identify the assignment's left-hand side and right-hand side. We call `l` and `cos[3.099]` as *tag values*. A tag value can be any expression; this allows the representation of arbitrary MATLAB assignments, including the multiple-value assignment [66].

In general, MATLAB statements are represented in MAGICA as

$$h[x_1 :\to y_1, x_2 :\to y_2, \ldots, x_n :\to y_n]$$

where the head h serves as a construct identifier, and where the delayed rules [99] $x_i :\to y_i$ $(1 \leq i \leq n)$ stand for *tag-value pairs*. The position of a tag-value pair isn't significant in MAGICA; this point should be kept in mind when making new definitions to extend the MAGICA system. A fair amount of documentation on data structure layouts has been coded into MAGICA itself as `usage` messages [99]; this provides a convenient, on-line way of pulling up layout information.

```
In[3]:= ?if
```

if[$$condition :> c_, $$then :> t_, $$else :> e_]
is the functional equivalent of an if statement in MATLAB. Forms such as
$$condition -> c , $$then -> t and $$else -> e can also be used.

8.3 Transitive Closure of a Graph

Figure 8.1 displays a complete MATLAB program that computes the transitive closure of a graph. The graph is represented as the $N \times N$ adjacency matrix `A`, which

```
function driver

N ← 512;

B ← tclosure(N);

disp(B);
```

```
function B = tclosure(N)
% Initialization.
A ← zeros(N, N);
for ii = 1:N,
  for jj = 1:N,
    if ii*jj < N/2,
      A(N-ii, ii+jj) ← 1;
      A(ii, N-ii-jj) ← 1;
    end;
    if ii == jj,
      A(ii, jj) ← 1;
  end;
end;
B ← A;
% Closure.
ii ← N/2;
while ii >= 1,
  B ← B*B;
  ii ← ii/2;
end;
B ← B > 0;
```

Fig. 8.1 The MATLAB Transitive Closure Program

is initialized arbitrarily in the function `tclosure`. Its transitive closure is returned in `B`. The shown code is directly from Alexey Malishevsky's thesis [65], with three nontrivial changes: (1) the `tic` and `toc` timing commands were removed, (2) `disp` was used to display `B`, and (3) the original monolithic script was reorganized into two files, one containing the function `driver` and the other containing `tclosure`.

8.3.1 M-File Contexts

Input files that constitute a MATLAB program are referred to as *M-files* in MATLAB parlance. Every M-file has its own parsed representation in MAGICA, which we call an *M-file context*. Through Mathematica's information-hiding context mechanism [99], MAGICA provides a way to save, and later retrieve, the M-file contexts of a MATLAB program. On line *In[4]* below, the M-file contexts of `driver` and `tclosure`, saved in an earlier session of MAGICA, are loaded from disk.[2]

```
In[4]:= Scan[load[#, load$Disk → True]&, {"tclosure`", "driver`"}]
```

An M-file context is basically a collection of Mathematica definitions that capture information about a user-defined MATLAB function. As an example, for a user-defined function f, a definition is made against the `statements` function object so that `statements[`f`]` expands to the function body of f. This is how the `type` object operates on the statements in the body of `driver` on line *In[5]* below.[3]

```
In[5]:= type[statements[driver]] // Timing
Out[5]= {5.47 Second, {_12 N1 → {512., $integer, {⟨1 ,  1⟩, 2}},
         _10 B1 → {⟦0, 1⟧, $boolean, {⟨512 ,  512⟩, 2}},
         {Indeterminate, $illegal, {⟨-1 ,  1⟩, 2}}}}
```

8.3.2 Interprocedural Type Inference

Definitions against the `type` object—currently over a 100—take care of propagating information across user-defined function interfaces. Thus the application of `type` on line *In[5]* causes the type information for N to be propagated into `tclosure`, resulting in the shown type inference for its output variable `B`. On line *Out[5]*, `_12N1` stands for N and `_10B1` for `B`; this renaming is an artifact of the way in which the Mathematica representations for this program were automatically generated. *Out[5]* shows that the value range of `B` is $[\![0,1]\!]$, its intrinsic type is `$boolean`, and that its shape is 512×512. The third inference on *Out[5]* represents the type of `disp`'s outcome; the shown values are because `disp` doesn't return anything.

8.3.3 Function Type Signatures and Procedure Cloning

The type signature of a function at a call site is a tuple of the type signatures of the actual arguments at that call site. An actual argument's type signature, in turn, is a triplet of its value range, intrinsic type and shape. MAGICA currently expects the

[2]These representations are in ASCII, and can be manually or automatically generated.

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

type signatures of an *M-file function* at all its call sites to be identical. This is an inconvenience of the implementation but not a limitation. If two call sites of an M-file function f differ in their type signatures, they must be replaced by invocations to f_1 and f_2, where f_1 and f_2 are cloned versions of f. In the current version of MAGICA, the front-end has the onus of performing such a duplication. The issue of procedure cloning doesn't apply to the example of this section because it invokes only one M-file, namely `tclosure`, at exactly one point in the driver M-file.

8.4 Fibonacci Numbers

MAGICA uses the expression `Sequence[`s_1`, `s_2`, ..., `s_n`]` to denote a sequence of statements. Each s_i can be an assignment, an `if` conditional, a `for` loop, a `while` loop, a `break` or a `return`. Anything else is taken to be an expression statement.[4] On *In[6]* below, a sequence of statements that calculates the nth Fibonacci number in `fn` is defined and assigned to the Mathematica symbol `stmts`.

```
In[6]:= stmts := Sequence[t1 ← sqrt[5], t2 ← plus[1, t1], t3 ←
        mrdivide[t2, 2], t4 ← mpower[t3, n], t5 ← mrdivide[t4, t1],
        fn ← round[t5]]
```

Every statement in the above sequence is an assignment; the construction $l \leftarrow r$ is a shorthand for `assignment[$$lhs :→ `l`, $$rhs :→ `r`]` in MAGICA.[5]

As an aside, MAGICA defines a function object called `show` that displays MAGICA data structures in MATLAB syntax. This is useful for visualization.

```
In[7]:= ?show
```

show[s_, opts___] displays a MATLAB structure s
as an appropriate MATLAB code fragment, under the control of options in opts.

show[] displays the MATLAB structure assigned to $mfile in the current context.

show has the following options: show$Stream, show$ASCII and show$Notebook. The last two are available only in a notebook – based front – end.

---

[4]Unlike languages such as C, Java, APL and Mathematica itself, MATLAB doesn't consider an assignment as an expression. Hence, a phrase like `(a ← 1)+1` is syntactically illegal in MATLAB.

[5]The symbol ← can be directly entered into a notebook by typing the key sequence ESC `<-` ESC.

Below we see what the MATLAB progenitor of `stmts` would have looked like.

In[8]:= **show[stmts]**

```
%  MATLAB Code Fragment
t1 = sqrt(5);
t2 = 1+t1;
t3 = t2/2;
t4 = t3^n;
t5 = t4/t1;
fn = round(t5);
```

8.4.1 A Type Inference on Statement Sequences

The MAGICA `type` object can be applied on a sequence of statements to produce a list of rewriting rules; *Out[9]* shows what happens when `type` operates on `stmts`.

In[9]:= **type[stmts]**

```
Out[9]= {t1 → {2.23607, $real, {⟨1, 1⟩, 2}},
         t2 → {3.23607, $real, {⟨1, 1⟩, 2}},
         t3 → {1.61803, $real, {⟨1, 1⟩, 2}},
         t4 → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {mpowerST(σ (n), ⟨1, 1⟩), 2}},
         t5 → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {mpowerST(σ (n), ⟨1, 1⟩), 2}},
         fn → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {mpowerST(σ (n), ⟨1, 1⟩), 2}}}
```

In the above, we see that the golden ratio computed in `t3` has the value 1.61803, the `$real` intrinsic type, and the $\langle 1, 1\rangle$ shape tuple with rank 2.[6]

8.4.1.1 Value Ranges in MAGICA

MAGICA's value-range inference subsystem is built on Mathematica's interval arithmetic [99]. The value range that MAGICA *conservatively* generates against a MATLAB expression consists of value bounds that *all* elements of that expression honor. (Keep in mind that a MATLAB expression can be a multi-element array.)

Complex Value Ranges MAGICA can denote everything from a value point to a complex value range. The latter are represented using interval arithmetic on real and imaginary subranges. If a MATLAB expression e has the value range

$$[\![r_l, r_h]\!] + [\![i_l, i_h]\!]\,\mathbb{i},$$

[6] The inferred rank is redundant here; it becomes significant when the shape tuple is symbolic.

it means that the elemental values of e have real and imaginary parts that lie between the inclusive end points r_l and r_h, and i_l and i_h, respectively.[7] Thus *Out[9]* shows that the value ranges of `t4`, `t5` and `fn` are all of the form

$$[\![-\infty, \infty]\!] + [\![-\infty, \infty]\!]\,\mathbb{i}.$$

Observe that for these program variables, this is also the best inferable value range. This is because `t4`, `t5` and `fn` are all dependent on `n`, whose value range is unknown.

Value-Range Operators MAGICA provides a number of useful "primitives" in connection with value ranges. These operators form the basis for MAGICA's value-range inference code. For example, the `nextN` function object returns the IEEE 754 machine *normal* number [41] that comes after a given real number.[8]

```
In[10]:= Names["Type`ValueRange`*"]
Out[10]= {nextN, prevN, υ, υAdd, υApproximation, υDivide, υExp,
          υIntersection, υLimit, υLog, υMemberQ, υMultiply,
          υPointQ, υPower, υRecombine, υUnion, υ$interprocedural}
```

In their ability to handle complex value ranges, value-range operators (the third to the second-last in the list on *Out[10]*) go beyond Mathematica's interval arithmetic. This is illustrated below, where `υExp` exponentiates a complex value range.

```
In[11]:= υExp[⟦1, π⟧+ⅈ⟦3.1, 4⟧]
Out[11]= ⟦-23.1407, -1.77679⟧ + ⅈ ⟦-17.5129, 0.962205⟧
```

If Mathematica's built-in `Exp` object is used instead, the result remains unevaluated because interval arithmetic in Mathematica is only set up for real value ranges.

```
In[12]:= Exp[⟦1, π⟧+ⅈ⟦3.1, 4⟧] // N
Out[12]= 2.71828^(⟦1.,3.14159⟧+(0.+1. ⅈ) ⟦3.1,4.⟧)
```

[7] MAGICA slightly alters Mathematica's syntax; the way the ⟦ and ⟧ symbols are used is an instance of this. In Mathematica *sans* MAGICA, these are used in a syntactically different way [99].

[8] `nextN` and `prevN` are used in MAGICA for correctly adjusting value-range end points.

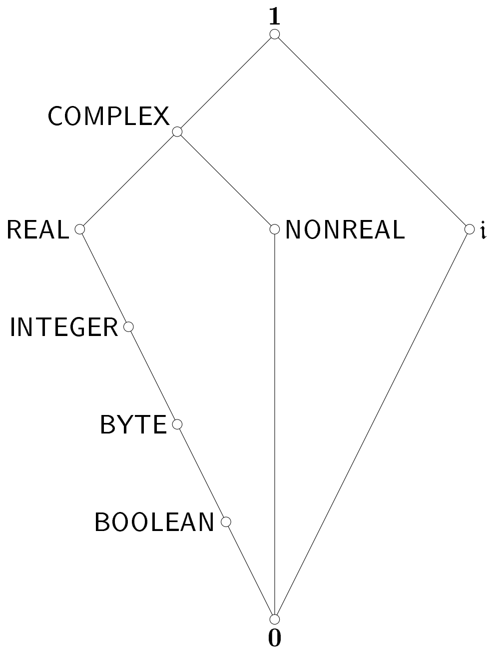

Fig. 8.2 The Lattice $\mathbb{T}$ of Intrinsic Types in MAGICA

8.4.1.2 Intrinsic Types in MAGICA

An intrinsic type in MAGICA denotes a logical set to which *all* elemental values of a MATLAB expression belong. MAGICA's intrinsic types are organized as a lattice $\mathbb{T}$ in which the partial order is value subsumption.[9] This lattice was described in [51] and is reproduced below. MAGICA represents the least and greatest elements of $\mathbb{T}$ by the symbols `$0` and `$1`. The symbols `$boolean`, `$byte`, `$integer`, `$real`, `$complex`, `$nonreal` and `$illegal` similarly denote BOOLEAN, BYTE, INTEGER, REAL, COMPLEX, NONREAL and $\mathfrak{i}$ respectively.

```
In[13]:= 𝕋
Out[13]= {$0, $boolean, $byte, $integer,
          $real, $nonreal, $complex, $illegal, $1}
```

In [51], a BOOLEAN stood for a 0 or 1, and a BYTE for an 8-bit unsigned integer. The same interpretations have been carried over to MAGICA. As in [51], INTEGER,

[9] The relation $s \leq t$ means that all values representable by s are also representable by t.

REAL, COMPLEX and NONREAL signify $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{C} - \mathbb{R}$ in MAGICA (the sets of integers, reals, complexes and strict complexes respectively). [51] didn't assume a specific bit width for INTEGER; MAGICA however promotes the intrinsic type of integral values outside the range $[\![-2^{32}+1, 2^{32}-1]\!]$ to REAL, which represents a double-precision number. Lastly, the abstract "illegal" intrinsic type $\mathfrak{i}$ signifies intrinsic type errors. Binary comparisons among the symbols work as expected.[10]

```
In[14]:= {$0 ≤ $boolean, $byte ≤ $nonreal, $nonreal ≤ $byte}
Out[14]= {True, False, False}
```

8.4.1.3 Array Shapes in MAGICA

The "array shape" of a MATLAB expression e in MAGICA is the pair $\{\sigma(e), \rho(e)\}$, where $\sigma(e)$ and $\rho(e)$ are the shape tuple and rank of e. (The term "array shape" is used in a specific sense here as opposed to the broader sense—namely, a set of shape tuples—of the earlier chapters.) On *Out[1]*, we saw that the shape tuple and rank of `tanh(3.78i)` were $\langle 1, 1\rangle$ and 2. *Out[9]* shows that the shape tuple of `t4` is

$$\texttt{mpowerST}(\sigma(\texttt{n}), \langle 1, 1\rangle).$$

MAGICA infers this from the right-hand side of `t4 ← t3^n`. Because `t3` is inferred to be a scalar from the preceding assignment in `stmts`, and because the shape tuple of `n` is unknown, MAGICA initially computes the shape tuple of `t4` to be

$$\texttt{mpowerST}(\langle 1, 1\rangle, \sigma(\texttt{n})).$$

The above expression fully describes the shape tuple of `t4` because the matrix power built-in function in MATLAB (invoked either as `mpower(t3, n)` or as `t3^n`) is a Type I operation (see Chapter 3). `mpowerST` above is its shape-tuple operator.

Shape Semantics of the Matrix Power Built-In Function Why is the shape tuple of `t4` symbolic? This is due to the shape semantics of `mpower`:

[10] The two `False` values on *Out[14]* mean that NONREAL and BYTE aren't comparable.

1. When both `a` and `b` are scalar, `a^b` is the elementary power operation.
2. Otherwise, when `b` is a nonnegative integer, `a^b` is computed by repeated squaring [66]. This means that `a` must be a square matrix for the operation to be valid. The result then has the same shape as `a`. If `b` is a negative integer, MATLAB inverts `a` before proceeding according to this case.
3. For other scalar values of `b`, MATLAB calculates `a^b` using eigenvalues and eigenvectors [66]. Once again, the operation is valid only if `a` is a square matrix, in which case the result has the same shape as `a`.
4. If `b` is a nonscalar square matrix, `a` has to be scalar for the operation to valid. In this case, the shape of the result is the same as that of `b`.
5. Any other shape for `a` or `b` results in an error.

Therefore, because `t3` is a scalar and `n` has an unknown shape, `t3^n` could have either the same shape as `n`, or an "illegal" shape depending on whether `n` is a square matrix. To capture these possibilities, MAGICA returns a symbolic shape tuple.

Canonicalization After arriving at $\texttt{mpowerST}(\langle 1,1\rangle, \sigma(\texttt{n}))$, MAGICA rearranges it to $\texttt{mpowerST}(\sigma(\texttt{n}), \langle 1,1\rangle)$. This happens because for any two shape tuples $\sigma(s)$ and $\sigma(t)$, $\texttt{mpowerST}(\sigma(s), \sigma(t))$ and $\texttt{mpowerST}(\sigma(t), \sigma(s))$ are *equivalent*—that is, they represent the same shape. This commutative property is coded against `mpowerST` in MAGICA and causes expressions involving `mpowerST` to be automatically reduced to a *canonical form* [99]. Canonical forms enable MAGICA to detect shape equivalence, which in turn permits the reuse of shape-tuple expressions.

```
In[15]:= typeReuse[%9]
Out[15]= {t1 → {2.23607, $real, {⟨1, 1⟩, 2}},
          t2 → {3.23607, $real, {⟨1, 1⟩, 2}},
          t3 → {1.61803, $real, {⟨1, 1⟩, 2}},
          t4 → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {mpowerST(σ (n), ⟨1, 1⟩), 2}},
          t5 → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {σ (t4), 2}},
          fn → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {σ (t4), 2}}}
```

8.4.2 Replacing Matrix Power by Array Power

The previous MATLAB Fibonacci code will compute the nth Fibonacci number whenever n is a nonnegative scalar integer. When n is a square matrix, the code will still execute though what is computed in `fn` will probably little resemble a traditional Fibonacci number. And when n is a rectangular matrix or a higher dimensional array, the code will fail due to reasons mentioned earlier.

If we replace the `mpower` built-in by the `power` built-in, we obtain a nice generalization of the code to arbitrary arrays. The `power` built-in, invoked either as `power(a, b)` or as `a.^b`, performs an *elementwise* power operation [66]:

1. When either `a` or `b` is a scalar, the other operand can be any array and the result has the same shape as the other operand.
2. Otherwise, the operation is valid only if `a` and `b` have the same shape. The shape of the result then is the common shape of `a` and `b`.

Substituting `mpower` by `power` allows us to compute an elementwise Fibonacci. The shape of `fn` will then be exactly the shape of `n`, as displayed below.

```
In[16]:= type[t1 ← sqrt[5], t2 ← plus[1, t1], t3 ← mrdivide[t2, 2],
         t4 ← power[t3, n], t5 ← mrdivide[t4, t1], fn ← round[t5]]
Out[16]= {t1 → {2.23607, $real, {⟨1, 1⟩, 2}},
          t2 → {3.23607, $real, {⟨1, 1⟩, 2}},
          t3 → {1.61803, $real, {⟨1, 1⟩, 2}},
          t4 → {(1 + i) ⟦-∞, ∞⟧, $complex, {σ(n), ρ(n)}},
          t5 → {(1 + i) ⟦-∞, ∞⟧, $complex, {σ(n), ρ(n)}},
          fn → {(1 + i) ⟦-∞, ∞⟧, $complex, {σ(n), ρ(n)}}}
```

8.4.3 Implicit Intrinsic Type Coercion

How can we gracefully handle situations in which the elemental values of n aren't nonnegative and integral? One way of doing this is by adopting a conversion policy. For instance, given an n, we could calculate the mth Fibonacci number where

$$m = \lfloor |R(n)| \rfloor,$$

and where R extracts the real part of n. The preceding *ad hoc* conversion will at least ensure that m equals n for a nonnegative integral n. When `type` is applied on the modified Fibonacci code, a more refined inference is obtained against `fn`.

```
In[17]:= type[t1 ← sqrt[5], t2 ← plus[1, t1], t3 ← mrdivide[t2, 2],
         n1 ← real[n], m ← floor[n1], t4 ← power[t3, m], t5 ←
         mrdivide[t4, t1], fn ← round[t5]]
Out[17]= {t1 → {2.23607, $real, {⟨1, 1⟩, 2}},
          t2 → {3.23607, $real, {⟨1, 1⟩, 2}},
          t3 → {1.61803, $real, {⟨1, 1⟩, 2}},
          n1 → {⟦-∞, ∞⟧, $real, {σ (n), ρ (n)}},
          m → {⟦-∞, ∞⟧, $real, {σ (n), ρ (n)}},
          t4 → {⟦0, ∞⟧, $real, {σ (n), ρ (n)}},
          t5 → {⟦0, ∞⟧, $real, {σ (n), ρ (n)}},
          fn → {⟦0, ∞⟧, $real, {σ (n), ρ (n)}}}
```

Observe that MAGICA determines the intrinsic type of `fn` to be `$real` rather than `$integer` because the upper bound of the elemental values in `fn` exceeds $2^{32} - 1$. For further examples of MAGICA's usage, the reader is referred to [52].

8.5 Architecture

MAGICA is used through a *front-end*, which is a separate operating system process that builds a Mathematica representation of an input MATLAB program. The front-end transfers the representation to MAGICA for type analysis; the type inferences that MAGICA generates are transferred back, for use in type-related optimizations, code generation or simply for code annotation and visualization.

Exchanges between the front-end and MAGICA happen across an interprocess communication link using the MathLink protocol [99]. Figure 8.3 shows three existing front-ends to MAGICA. Two of these—the GUI-based notebook and the text-based interface—are shipped with Mathematica. Interacting with MAGICA using them requires either the handcrafting of the program representations that are to be type inferred, or the availability of those representations on disk. (*In[1]* and *In[2]* are examples of handcrafted representations; *In[4]* uses prefabricated representations.) The third front-end, called MAT₂C, is a custom-built one that takes a MATLAB program in its native form and translates it to optimized C; it uses MAGICA as the inference engine to obtain the necessary type information. In fact, it was by using MAT₂C that the M-file contexts for the example in § 8.3 were produced in advance. Figure 8.3 also shows the disk image of a sample M-file context.

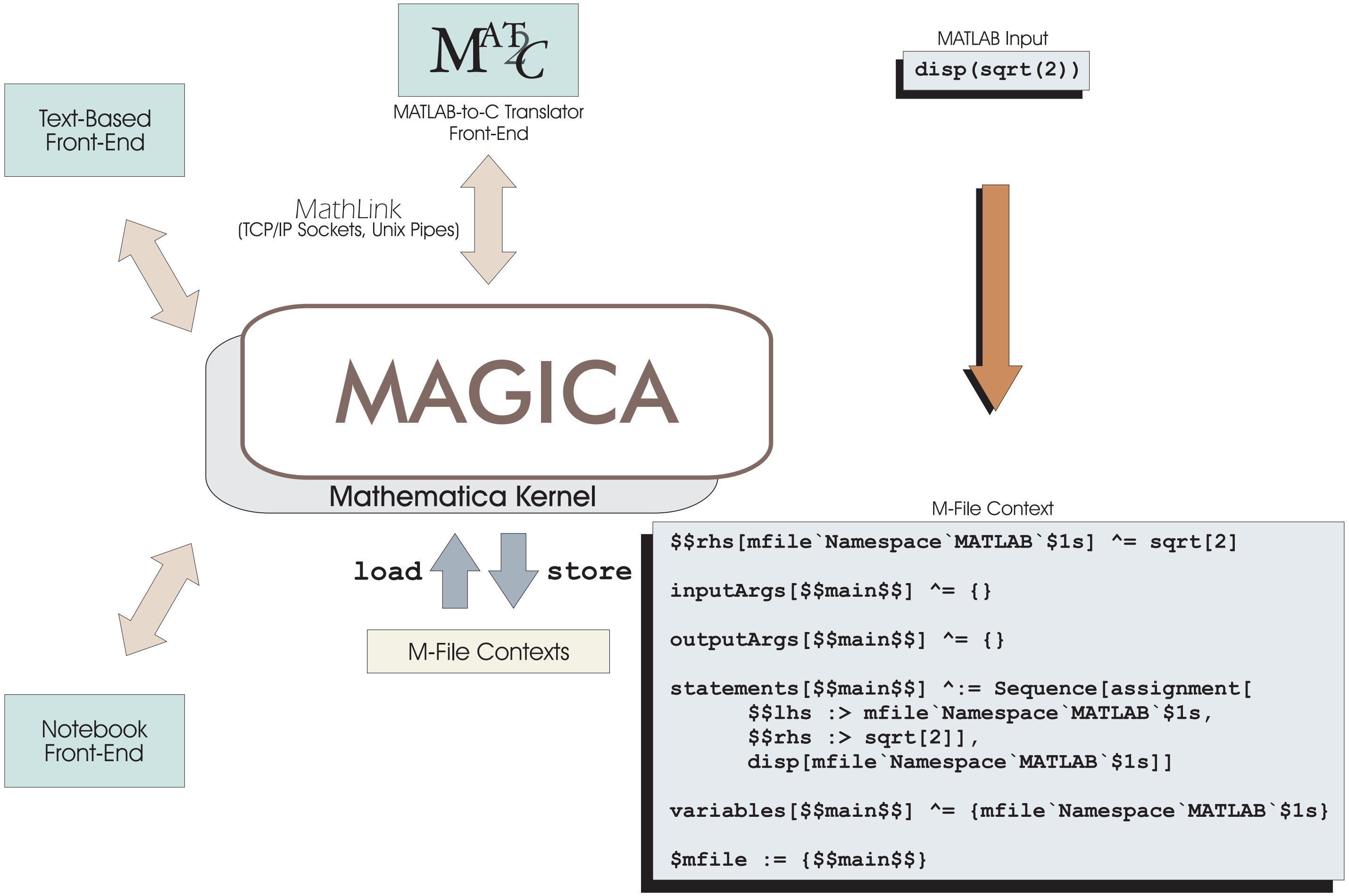

Fig. 8.3 The MAGICA Architecture

Chapter 9

Static Array Storage Optimization

And in those days many a clever programmer derived an immense intellectual satisfaction from the cunning tricks by means of which he contrived to squeeze the impossible into the constraints of his equipment.
— EDSGER W. DIJKSTRA
"The Humble Programmer,"
Communications of the ACM, Vol. 15, No. 10, October 1972. Page 860.

The automatic optimization of program storage is a well-known problem, one that has been extensively investigated and efficiently tackled in the past. Because programs in early computers were constrained by the sizes of their cores, initial work centered on main memory allocation [64, 37, 38]. As large main memories became cheaply available, interest in this area waned with the focus shifting to register allocation [19, 24, 16, 15, 79]. This chapter revisits the main memory allocation problem in the context of MATLAB and shows how the efficient management of array storage is crucial to its successful *static* compilation. The fact that we have to contend with arrays whose type characteristics aren't expressly specified is an important distinction between this work and previous efforts at name reclamation [28, 78] and storage reduction [100, 26]. The issue of storage optimization is particularly relevant to MATLAB because the language is increasingly being used in the prototyping of applications that ultimately get deployed on limited memory platforms such as DSPs and FPGAs, and because it espouses an array-centric programming philosophy in which the elemental manipulation of data is eschewed for whole array processing.

Closer to the optimization described in this chapter is Fabri's approach [38] which viewed main memory allocation as a weighted graph coloring problem. Nodes in the interference graph were weighted by the sizes of the arrays in question and a coloring was sought that minimized the total allocated storage. What complicates the problem in MATLAB is that size information may be absent since types aren't explicitly declared. In fact, even when all array sizes are known, the problem doesn't naturally map to graph coloring the way the register allocation problem does. Factors such as partial interference (see § 9.2.1) further ravel the picture.

Surprisingly, though the same problems existed in APL, completely different techniques, like "drag-along" and "beating" [1], delayed evaluation [42], chaining [22] and demand driven execution [17], were used to reduce storage overheads.

In research on static [32] and just-in-time [7] compilation of MATLAB, storage optimization has been an unaddressed area. Although systems that compile code during execution like MaJIC [7] are better poised to efficiently manage storage, we must keep in mind that such run-time systems will themselves compete for platform storage space. Therefore, we believe that there exists a niche of applications that could greatly benefit from the better static management of MATLAB's array storage.

9.1 Algorithm Overview

The input to the algorithm of this chapter is a control-flow graph (CFG) that is in the SSA form. The algorithm, referred to as GCTD (Graph Coloring with Type-based Decomposition) consists of two phases. The first involves the creation and coloring of an interference graph to determine variables that can share common storage. Because the objective isn't the ultimate assignment of variables to a machine's register set, we neither seek an n-coloring, where n is the number of available machine registers, nor deal with related issues such as register spilling, as is the case in [19]. Instead, the goal of the first phase is to determine, at least conservatively, noninterfering *classes* of variables that can be safely assigned to the same area of memory without changing the meaning of the program. To this end, the aim is to find a coloring of the interference graph.[1] The treatment of interference in the first phase is similar to that in [19] except that a new kind of conflict arising from operator semantics needs to be considered and resolved, perhaps through the involvement of inferred type information (see § 9.2.4). The second phase decomposes the color classes on the basis of expression types. Basically, the idea is to partition each color class into *groups* so that all variables in a group are laid out in memory starting from the same location as a certain distinguished set of variables in the group. Members of that distinguished set have the predominant storage requirement in the group. The cornerstone of the second phase is a partial order that relies on program variable types, and in the symbolic case, on control-flow information too.

9.2 Interference Graph (Phase 1)

The interference graph $G^*_f = (V_f, E_f)$, constructed at the level of a user-defined MATLAB function f, has a node set V_f that represents the variables defined in the CFG and an edge set E_f that represents the interference among the variables. Two variables are considered to interfere if their def-use chains (du-chains) overlap. In particular, we use the Chaitin et al. [19] notion of interference by which two variables interfere if there exists an execution path in f and a point therein at which: (1) the definitions of both variables are visible; (2) both variables aren't dead; and (3) the variables have different values. Because determining these criteria is in general undecidable, we use an approximation of interference by considering variables that are both *available* and *live* at each assignment [19, 15]. A variable v is regarded available at a statement s if there is a possible execution path from a definition of v to s. A variable w is treated live at s if there is a possible execution path from s to a

[1] Currently, our implementation seeks a coloring that is as close to the minimal coloring as possible. This, however, may not yield the final optimal solution to the problem.

use of w along which w isn't redefined. "Liveness" and "availability" as defined here are conservative because they indicate a potential, rather than a definitive, use and definition. Our approach to detecting variables that satisfy these two conditions is effectively that described in [15]. A basic block is traversed backwards starting with the set of variables that are live and available at its end. A definition encountered during the traversal is interfered with members of the set. Before moving to the next statement, the set is updated by removing the variable (or variables) defined at that statement and adding variables used at the same statement.

9.2.1 Partial Interference

The concept of interference described in [19] was intended for scalars. When directly applied to variables that can also be arrays, which is the case in MATLAB, it can be more pessimistic than necessary. This will happen when variables only partially interfere. As an example, the SSA conforming intermediate representation (IR)[2] shown below creates a pair of 2×2 matrices in `a` and `b`, assigns to `c` the first element in `a`,[3] then computes the sum of `b` and `c` in `d`, and finally displays `d`. Because the du-chain of `a` crosses that of `b`, `a` and `b` clearly interfere.

```
a ← rand(2, 2);
b ← rand(2, 2);
c ← subsref(a, 1);
d ← b+c;
disp(d);
```

However, by inspecting the index used in the access, we see that this interference is only in the first element of `a`. Although our current implementation considers `a` and `b` to fully interfere, which causes them to be assigned to disjoint areas in

[2]Assignments in our IR have right-hand sides that consist of at most a single MATLAB operation like `*`, or a pseudo operation like the ϕ function. MATLAB assignments can be cast into this Single Operator (SO) form through the introduction of temporaries. Each of these SO form IR assignments are then mapped by the code generation phase to C code.

[3]`subsref(a,` i_1, i_2, ..., i_m`)` returns the array element `a(`i_1, i_2, ..., i_m`)`.

memory, their storage areas could have been overlapped allowing for a total of five double-precision memory locations to perform the above computation in its entirety.

9.2.2 Handling Copies

A copy statement of the form $X \leftarrow Y$ will by itself not introduce an interference between X and Y. However, if the coloring step ascribes different colors to X and Y, the final code generation stage will have to emit code to copy Y to X. Because variables in MATLAB can be arrays, it is even more important, in relation to a situation that deals only with scalars, to avoid copy operations whenever possible. In [19], this was attacked by coalescing X and Y and then rebuilding the interference graph since the action of coalescing can alter the graph; the process was repeated up to a fixed number of times until no further coalescences were possible. Our strategy is different: we preprocess the CFG to free it of copies by subjecting it to a copy propagation pass followed by dead-code elimination.[4] Obviously, all copies cannot be eliminated this way; one such case is shown in Figure 9.1. In the displayed code fragment, copy propagating s_1 from $t_2 \leftarrow s_1$ to $t_1 \leftarrow \phi(t_0, t_2)$ will change the meaning of the IR. Of course, t_2 and s_1 in the interference graph will also not be coalescent because they interfere due to overlapping du-chains.

9.2.3 SSA Inversion Copies

Since converting back from the SSA form will reintroduce copies [29], it is imperative that the name defined by a ϕ-function assignment share storage with the names used by the ϕ function whenever possible. This will make the reintroduced copies identity assignments and therefore trivially removable. For instance, to achieve this for the ϕ-function assignment $Z \leftarrow \phi(X, Y)$, the nodes in the interference graph corresponding

---

[4]The implementation of the GCTD algorithm was done as part of the MAT2C translator. Besides code generation, the translator has over 20 passes that perform a variety of transformations such as global common-subexpression elimination, constant folding, constant propagation, copy propagation, type determination, dead-code elimination and code selection.

$$
\begin{aligned}
s_1 &\leftarrow \phi(s_0, s_2) \\
t_1 &\leftarrow \phi(t_0, t_2) \\
&\vdots \\
t_2 &\leftarrow s_1 \\
s_2 &\leftarrow \dots
\end{aligned}
$$

Fig. 9.1 Irremovable Copies

to Z and X (and then Z and Y) are examined for any interference. If they don't interfere, they are coalesced and the interference graph is appropriately updated. This will force any coloring to assign the same color to Z and X so that the copy $Z \leftarrow X$ reintroduced during the SSA inversion step becomes a trivially removable identity assignment. Of course, such a coalescing will affect the chromatic number of the graph and constrain the coloring finally obtained. But we have found the folding of copies to be indispensable to the generation of efficient code, because even a few copies, involving large-sized arrays and nested within loops, can significantly impact the generated code's performance through excessive paging activity.

9.2.4 Interference due to Operator Semantics

Consider the IR assignment `c` $\leftarrow$ `a*b`. If `a` and `b` are scalars, storage for `c` and `a`, or `c` and `b`, can be shared, assuming that these pairs don't otherwise interfere. We say that `c` can be computed *in-place* in either operand in the equivalent C code without violating the semantics of the operation. However, `*` can operate on both scalars and nonscalars in MATLAB. For example, when `a` and `b` are nonscalar matrices, `*` performs the elementary matrix multiplication operation. In that situation, computing `c` in-place in either `a` or `b` will risk violating the semantics of the operation because elements in `a` or `b` could get overwritten before being fully used. But when either `a` or `b` is a scalar, and the other operand is an arbitrary array, `c` is produced by the elementwise multiplication of the scalar with the array. In that case, `c` can be computed in-place in the array operand.

Thus, the storage coalescing pass inspects each IR assignment of the form $Y \leftarrow op(X_1, X_2, \ldots, X_m)$, and inserts additional interferences between Y and an X_i depending on what *op* is, and the type information inferred for each X_i. Hence, for the case `c` $\leftarrow$ `a*b`, an interference is *not* added between `c` and `a`, and between `c` and `b`, if either `a` or `b` can be determined to be scalars. Since `c` can then be calculated in-place in the larger sized operand, we still have to figure out which of the two operands can accommodate `c`. This is done in the second phase of the GCTD pass.

9.2.4.1 Array Addition

An instance of a different *op* is in `c` $\leftarrow$ `a+b`. Since `c` can be computed in-place in either `a` or `b` provided the operand is sufficiently sized, no additional interferences have to be inserted between `c` and either operand in this case. Figure 9.2 displays exactly how the C code generated by the MAT2C translator performs this in-place computation. (Not shown are preceding shape correctness and resizing checks.)

9.2.4.2 Right-Hand Side Array Indexing

In the IR statement `c` $\leftarrow$ `subsref(a, 1)` seen in § 9.2.1, *op* is `subsref`, which corresponds to the right-hand side array indexing operation in MATLAB. We call this the *R*-indexing operation for short. Note that in the case seen in § 9.2.1, `c` can be computed in-place in `a`. (There are only two possibilities: `subsref(a, 1)` is either an out-of-bounds access, which will happen if `a` is the empty array, or it is legal, in which case the result is a scalar.) However, if the statement were `c` $\leftarrow$ `subsref(a,` e`)`, where e was unknown, such an in-place computation may not be possible. This is because MATLAB permits e to itself be an array, thereby allowing an arbitrary permutation of the elements of `a` to be returned. For example, if e were the MATLAB colon expression `4:-1:1`, `subsref(a,` e`)` would give the elements of the 2×2 matrix `a` in reverse. Again, type information could be used to

```
/* ___STC(s, i) is a macro call that gets substituted
   by the extent of an array along dimension i. */
if ((___STC(_826s_4, 1) == 1 && ___STC(_826s_4, 2) == 1))
   { /* First operand is a scalar. */
    for (__i = 0; __i < (___STC(_833s_4, 1)*___STC(_833s_4, 2)); __i++)
    _811s_4[__i] = _811s_4[0]+_804s_4[__i];
    }
else if ((___STC(_846s_4, 1) == 1 && ___STC(_846s_4, 2) == 1))
   { /* Second operand is a scalar. */
    for (__i = 0; __i < (___STC(_833s_4, 1)*___STC(_833s_4, 2)); __i++)
    _811s_4[__i] = _811s_4[__i]+_804s_4[0];
    }
else
   { /* Both operands have identical shapes. */
    for (__i = 0; __i < (___STC(_833s_4, 1)*___STC(_833s_4, 2)); __i++)
    _811s_4[__i] = _811s_4[__i]+_804s_4[__i];
    }
```

Fig. 9.2 Partial MAT2C C Code for the IR _811s_4 $\leftarrow$ _811s_4+_804s_4 (from the capr Benchmark) differentiate these situations—in this case, whether the subscript e is a scalar.

9.2.4.3 Left-Hand Side Array Indexing

MATLAB also offers a left-hand side array indexing operation. In source form, this operation, which we call L-indexing for short, is specified by the construction

$$\texttt{a(}l_1\texttt{, } l_2\texttt{, } \ldots\texttt{, } l_m\texttt{)} \leftarrow r$$

which is represented in our IR as `a` $\leftarrow$ `subsasgn(a,` r`,` l_1`,` l_2`, ...,` l_m`)`. When cast to the SSA form, this becomes `b` $\leftarrow$ `subsasgn(a,` r`,` l_1`,` l_2`, ...,` l_m`)`. The meaning of this operation is as follows: `b` has the same elements as `a`, except for elements located by the m subscripts, which are set to elements from r. The `subsasgn` operator therefore exhibits semantics similar to that of the `Update` operation described in [29] except for two important departures:

- Each subscript l_i $(1 \leq i \leq m)$ can be an arbitrary array. The locations in `b` that are set to elements in r are obtained by taking the Cartesian product of the elements in the subscripts. If p_i is the number of elements in subscript l_i, r is required to have the shape $p_1 \times p_2 \times \cdots \times p_m$. (There are corner cases that need to be handled. See [66] for details.)
- If the maximum elemental value in l_i exceeds the extent of `a` along the ith dimension, `b` has that maximum value for its extent along the ith dimension. Locations created in `b` due to such expansions are set to 0.

In fact, there is one more case that can arise due to an additional “shrinkage” feature in the L-indexing operation. To illustrate, `a(:, :, 2)` $\leftarrow$ `[]` deletes all elements on the second “page” [66] of an $x \times y \times z$ array, shrinking it to an $x \times y \times (z-1)$ array. This feature isn’t currently supported by the MATLAB→C translator.

If the requisite amount of storage is available, can `b` in `b` $\leftarrow$ `subsasgn(a,` r`,` l_1`,` l_2`, ...,` l_m`)` be computed in-place in `a` for all possible l_i without risking the violation of semantics? It turns out that in the absence of the shrinkage feature, this can *always* be done by computing the elements of `b` from the last to the first. This works because there are only two cases that need consideration:

- The array `b` doesn't expand and has the same shape as `a`. Then, those elements of `a` that get carried over to `b` will occupy the same positions in memory.
- The array `b` expands along one or more dimensions of `a`. Then, those elements of `a` that get carried over to `b` will *not* be shifted downward in memory.[5]

Hence, by building `b` backwards, we are ensured that the elements in `a` get carried over to `b` before they can get overwritten.

9.2.5 Coloring Heuristic

A simple $O(V_f + E_f)$ greedy heuristic that attempts to use as few colors as possible has been used in our current implementation. The heuristic visits each node in turn, in the lexical order of the corresponding variable definitions, and assigns to it the smallest among the colors used so far that is consistent with its neighbors. If no such color exists, a new color is created and the node is assigned that color.

9.3 Type-Based Allocation (Phase 2)

Let $V_f(c)$ be a color class in the interference graph G_f^*. Let $\mathcal{S}(v)$ be the size of the storage allocated for a variable v in $V_f(c)$. Intuitively, if the storage sizes of all variables in $V_f(c)$ were known, they could all be overlaid starting from the same location as the largest sized variable among them. Unfortunately for the compiler writer, MATLAB features both implicit typing (an expression's type may not be explicit from program syntax) and dynamic typing (an expression's type may depend on the control-flow path exercised at run time). Consequently, not only may $\mathcal{S}(v)$ be statically indeterminable, but it may also vary during execution. Our heuristic approach to getting around these problems is to construct a storage-size partial order $\preceq$ that tries to capture a containment relationship while promoting spatial reuse. (The precise definition of this binary relation is given in § 9.3.2.) The construction is

[5] MATLAB lays out its arrays *à la* FORTRAN, i.e., in column-major order. But this observation is true even if the layout was row major.

done using *inferred* type information. Once $\preceq$ is formed, its maximal[6] elements are found, which are then used to decompose $V_f(c)$ into groups. If there are q maximal elements under $\preceq$, $V_f(c)$ is partitioned into a collection of q groups, one for each maximal element, such that the storage sizes of variables in a group are bounded by the maximal element corresponding to that group.

9.3.1 Obtaining Type Information

The MATC translator currently obtains the types of all variables defined in a program using MAGICA. Given a Mathematica representation of a MATLAB program, MAGICA infers the value range $\upsilon(w)$, intrinsic type $\tau(w)$, shape tuple $\sigma(w)$ and rank (i.e., array dimensionality) $\rho(w)$ of each variable w. If MAGICA cannot explicitly infer the extents or dimensionality of some w, it will return symbolic expressions for $\sigma(w)$ and $\rho(w)$ respectively. The unique aspect of these shape-tuple and rank expressions is that inferences are reused whenever symbolic equivalence can be established. That is, MAGICA will return the shape-tuple expression $\sigma(v)$ for the shape tuple of w if it can establish that $\sigma(v)$ will definitely be live (in the data-flow sense) at the definition of w and that $\sigma(v)$ and $\sigma(w)$ will always be equivalent.

9.3.2 The Storage-Size Partial Order

The GCTD pass determines the storage size of a variable u either by *statically estimating* it, or by symbolically calculating it to be $|\sigma(u)||\tau(u)|$ where $|\sigma(u)|$ denotes the number of elements in an array whose shape tuple is $\sigma(u)$, and where $|\tau(u)|$ denotes the storage size of a scalar having the intrinsic type $\tau(u)$. The pass uses the

[6]An element x is maximal under a partial order $\preceq$ if there exists no y such that $x \prec y$.

following formulation to relate the storage sizes of two variables u and v:

$$\mathcal{S}(u) \preceq \mathcal{S}(v) \text{ iff } \begin{cases} \begin{array}{l} \mathcal{S}(u) \text{ and } \mathcal{S}(v) \text{ can be} \\ \text{statically estimated,} \\ \tau(u) = \tau(v) \text{ and } \mathcal{S}(u) \leq \mathcal{S}(v), \end{array} \\ \\ \begin{array}{l} \mathcal{S}(u) \text{ and } \mathcal{S}(v) \text{ cannot be} \\ \text{statically estimated,} \\ u \text{ is available at} \\ \text{the definition of } v, \\ \tau(u) = \tau(v) \text{ and } \mathcal{S}(u) \leq \mathcal{S}(v). \end{array} \end{cases} \tag{9.1}$$

Because "available at the definition" is both a reflexive and transitive relationship (see § 9.2), it is easy to see from Relation 9.1 that $\preceq$ is indeed a partial order. The motivation behind the formulation is to identify two categories of arrays: those whose layouts can be fixed at compile time and those among the dynamically allocated arrays whose storages can be grown in a regular way. The two disjoint criteria for $\preceq$ require identical intrinsic types; this was intentional so as to avoid both the use of type casts in the generated C code and the issue of alignment restriction in C [61].

9.3.2.1 Stack Allocation

Static estimation of storage size is done in two situations:

1. the inferred shape tuple $\sigma(u)$ of a variable u is *explicit*—that is, of the form $\langle p_1, p_2, \ldots, p_k \rangle$ where each extent p_i $(1 \leq i \leq k)$ is an integer; or

2. the variable u is defined by the ϕ-function assignment $u \leftarrow \phi(v, w)$ and the sizes of both v and w are statically estimable.

The estimated size in the first case is $|\sigma(u)||\tau(u)|$, while in the second case it is $\max(\mathcal{S}(v), \mathcal{S}(w))$. Arrays whose sizes can be statically estimated get allocated on the stack in the C translation. In particular, scalars in MATLAB can be directly mapped to scalar automatics in C. Besides enabling procedures to be reentrant, a stack allocation discipline automatically materializes and disappears objects as activation records get pushed and popped. Back-end C compilers can also take advantage of the fact that the relative displacements of the arrays within a stack frame will be compile-time knowns; knowing this could be helpful while gathering data dependency or aliasing information. Moreover, because all statically estimable sizes of the same intrinsic type within a color class form a single chain under the partial order $\preceq$, the corresponding variables, which form a single group, could all be allocated within a single array whose size is the maximal element in the chain.

9.3.2.2 Heap Allocation

Variables that satisfy neither condition in § 9.3.2.1 are deemed as having statically inestimable sizes and get allocated on the heap. Their storage sizes are expressed symbolically as $|\sigma(u)||\tau(u)|$. The GCTD pass binds all variables within a group to a common storage area. Code generated by our implementation also attempts to alleviate heap memory pressure by resizing storage on the fly to the specific needs of each variable in a group. This is in contrast to the stack allocation case where the storage size of a group remains fixed at the maximal during a procedure activation. By incorporating the "available at the definition" clause, the second criterion of Relation 9.1 tries to identify stretches in the execution path along which arrays grow in one direction. Specifically, if $\mathcal{S}(u) \preceq \mathcal{S}(v)$ by the second criterion, then u must both be available at the definition of v and dead after it. (Otherwise, u would have interfered with v in Phase 1.) Thus, chained elements in $\preceq$ connected by the second criterion would potentially correspond to definitions and uses that get performed in sequence at run time and where the definitions step through nondecreasingly sized arrays. ("Potentially" because availability as defined in § 9.2 is conservative.) Moreover, the closer u is to v in the control-flow path, the better would be the spatial reuse characteristics because u would still be in the higher levels of a memory hierarchy when it gets resized to v. In fact, the closest that u could be to v is if it is used in the same statement that defines v. Closeness may thus be fostered by ensuring that the chains included in each group are as long as possible.

DECOMPOSE-COLOR-CLASS(G)

$G^{\text{SCC}} \leftarrow$ *component_graph*(G)
☛ The component graph $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$ of
☛ a directed graph $G = (V, E)$ has a node for each
☛ strongly connected component (SCC) in G, and
☛ a directed edge $x \to y$ if there is a directed edge
☛ from a node in the SCC in G corresponding to
☛ x, to a node in the SCC in G corresponding to y.
$G' \leftarrow$ *forest_with_in-degree_zero_roots*(G^{SCC})
☛ Decomposes G^{SCC} into a forest of trees by invoking
☛ either depth-first search (DFS) or bread-first search
☛ (BFS) on nodes with in-degrees of 0.
return G'

Fig. 9.3 The DECOMPOSE-COLOR-CLASS Procedure

9.3.3 Decomposing a Color Class into Groups

A directed graph $G = (V, E)$ is used to represent the storage-size partial order $\preceq$ on a color class V. A directed edge $u \to v$ is introduced between a pair of distinct variables u and v in V if and only if $\mathcal{S}(u) \preceq \mathcal{S}(v)$. The algorithm DECOMPOSE-COLOR-CLASS given in Figure 9.3 performs the decomposition. It uses two primitives: *component_graph*, which returns G^{SCC}, the component graph [27] of G; and *forest_with_in-degree_zero_roots*, which decomposes G^{SCC} into a forest of trees by running a DFS or BFS on nodes in G^{SCC} that have in-degrees of 0.

All variables in an SCC of G have the same storage size. Thus, the root of each tree returned by DECOMPOSE-COLOR-CLASS corresponds to an SCC in G whose variables have a maximal storage size under $\preceq$. This is because the roots have in-degrees of 0 in G^{SCC}. Note that in-degrees of 0 will exist because G^{SCC} has the important property of being acyclic [27]. Therefore, variables in all those SCCs in G that correspond to a returned tree's nodes have storage sizes that are bounded by a maximal element in $\preceq$. Hence, the collection of trees returned by DECOMPOSE-COLOR-CLASS forms a decomposition of a color class into groups.

Lemma 13 *If every node in a color class belongs to a unique maximal chain,*[7] *then* DECOMPOSE-COLOR-CLASS *assigns all nodes in a maximal chain to the same group.*

PROOF. Suppose there exists a maximal chain in which nodes belong to different groups. Because both BFS and DFS touch all nodes reachable from a root, this means that there exists a u such that $\mathcal{S}(u)$ is bounded by at least two distinct maximal elements, which is a contradiction. □

Hence, from Lemma 13, DECOMPOSE-COLOR-CLASS automatically fosters the closeness property mentioned in § 9.3.2.2 except in one situation. That situation occurs if a node is common to two or more maximal chains. The implementation currently assigns nodes common to two maximal chains wholly to one of them.

In terms of complexity, the running time of DECOMPOSE-COLOR-CLASS is $O(V+E)$ since Line 1 can be done in $O(V+E)$ time [27], and because Line 2 can also be done in $O(V+E)$ time since $V^{\mathrm{SCC}} = O(V)$ and $E^{\mathrm{SCC}} = O(E)$.

9.4 Examples

In this section, we present two examples that illustrate the GCTD algorithm on sample MATLAB code. The first shows opportunities for storage reuse in programs that process arrays through a chain of elementwise operations. The second demonstrates how storage could be shared even in situations wherein an array may expand.

9.4.1 Nonresized Arrays with Symbolic Types

When presented with the IR shown below on the left, MAGICA will infer the intrinsic types of t_1, t_2 and t_3 to be COMPLEX, assuming that nothing is known about t_0.

$$\begin{array}{lll} t_1 \leftarrow t_0\texttt{-1.345;} & & s^\diamond \leftarrow s\texttt{-1.345;} \\ t_2 \leftarrow \texttt{2.788.\^{}}t_1\texttt{;} & \Longrightarrow & s^\diamond \leftarrow \texttt{2.788.\^{}}s\texttt{;} \\ t_3 \leftarrow \texttt{tan(}t_2\texttt{);} & & s^\diamond \leftarrow \texttt{tan(}s\texttt{);} \end{array}$$

[7] A chain is maximal if no other chain properly subsets it.

MAGICA will also return the *symbolic* expression $\sigma(t_0)$ for the shape tuples of t_1, t_2 and t_3 (see Chapter 8). This reflects the fact that under *all* executions of the IR, the shapes of t_1, t_2 and t_3, which are all results of elementwise operations in MATLAB, will be identical to that of t_0. Assuming that t_0, t_1 and t_2 are dead on exit from the code fragment, we see that no pair of variables interfere, either due to overlapping du-chains or due to operator semantics. Therefore all nodes in the fragment's interference graph can be ascribed the same color. Furthermore, because $\sigma(t_0) = \sigma(t_1) = \sigma(t_2) = \sigma(t_3)$, $\tau(t_0) = \tau(t_1) = \tau(t_2) = \tau(t_3)$, and each t_i is available at the definition of t_{i+1} $(0 \leq i \leq 2)$, we get $\mathcal{S}(t_0) \preceq \mathcal{S}(t_1) \preceq \mathcal{S}(t_2) \preceq \mathcal{S}(t_3)$. Thus, all the variables in the fragment can be bound to a common storage that will be reused during the fragment's execution. In fact, since $|\sigma(t_0)||\tau(t_0)| = |\sigma(t_1)||\tau(t_1)| = |\sigma(t_2)||\tau(t_2)| = |\sigma(t_3)||\tau(t_3)|$, the storage sizes for all four variables can be statically determined to be the same. This means that at each of their definitions, their associated storage needn't be resized at run time. The IR shown above on the right indicates this by using the $\diamond$ superscript to denote a defined array that isn't resized.

9.4.2 Expandable Arrays with Symbolic Types

A more interesting example that may involve array expansion is shown in the IR given below. Here, an $x \times y$ identity matrix is created in `a`, which is then used to create a matrix `b` through the `subsasgn` operator.

$$\begin{array}{l} \texttt{a} \leftarrow \texttt{eye(}x\texttt{, } y\texttt{);} \\ \texttt{b} \leftarrow \texttt{subsasgn(a, 1, } i_1\texttt{, } i_2\texttt{);} \end{array}$$

Assuming that `a` is dead at the end of the code fragment, we see that the du-chains of `a` and `b` don't overlap. And because `b` can be formed in `a` provided the latter is large enough (see § 9.2.4.3), we observe that `a` and `b` don't interfere. Hence, a minimal coloring of the interference graph will put both `a` and `b` in the same color class. For the intrinsic types of `a` and `b`, MAGICA will return BOOLEAN; it will also return expressions for the shape tuples of `a` and `b` that in the absence of further information on x, y, i_1 and i_2 will likely be symbolic. However, because of the semantics described in § 9.2.4.3, we can be assured that $|\sigma(\mathtt{a})| \leq |\sigma(\mathtt{b})|$ will *always* be true. Thus, because $\tau(\mathtt{a}) = \tau(\mathtt{b})$ is also true, and `a` is available at the definition of `b`, $\mathcal{S}(\mathtt{a}) \preceq \mathcal{S}(\mathtt{b})$ will hold by Relation 9.1. Hence, `a` can share the same storage as `b`. If the storage sizes of both `a` and `b` are statically inestimable, a resizing check would have to be inserted before the code generated for each of the IR statements. If `a` and `b` have statically estimable sizes, both will be stack allocated within a maximal sized storage. These two situations are shown in the IR below where the $\pm$ superscript indicates a defined array that may need resizing and the $+$ superscript indicates a defined array that if resized, will be grown. The superscripts are applicable only in the dynamic case.

$$s^{\pm} \leftarrow \mathtt{eye(}x\mathtt{,}\ y\mathtt{);}$$
$$s^{+} \leftarrow \mathtt{subsasgn(}s\mathtt{,\ 1,}\ i_1\mathtt{,}\ i_2\mathtt{);}$$

There is one situation where `a` and `b` won't share storage even if they don't interfere; this will happen if the size of only one of them can be statically estimated.

Chapter 10

Performance Evaluation

When you measure what you are speaking about and express it in numbers, you know something about it, but when you cannot express it in numbers your knowledge about it is of a meagre and unsatisfactory kind.
— WILLIAM THOMSON (LORD KELVIN)
As quoted in "Comic Sections: The Book of Mathematical Jokes, Humour, Wit and Wisdom," Desmond MacHale, Boole Press, 1993. Page 145.

This chapter presents measurements from MAGICA and MAT2C, in-house implementations that realize the shape inference techniques and optimizations described in this dissertation. The efficacy of the methods was evaluated by collecting metrics relating to shape compositions, memory footprints and execution times over a set of 11 MATLAB programs obtained from different sources. These benchmarks are listed in Table 10.1 along with brief descriptions, their origins, and their sizes in terms of the number of files that constitute their source codes (called *M-files* in MATLAB jargon) and the total number of nonempty noncomment lines in them.

10.1 Benchmark Suite Organization

Programs in the set were organized along the lines of the FALCON benchmark suite [32] in which the main function in a program is invoked from a driver routine.

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

Legend:
❑ Benchmarks involve three-dimensional arrays.
† FALCON MATLAB Compiler Test Suite [32].
✞ OTTER Parallel MATLAB Compiler Test Suite [65].

Table 10.1 Benchmark Suite Description

Typical tasks performed by a driver are the preparation of arguments for an invocation, the display of results from an invocation and the timing of the entire execution. Though the MATLAB-to-C translator MAT2C can handle built-in functions like `disp` and `fprintf` that produce outputs to external files, no support currently exists for the loading of data from external files. Some of the drivers in their original form did read data from external files; these were modified by the inclusion of the loaded data in the driver. This wasn't found to be a problem because the loaded data was confined to only a few scalars. Nevertheless, the current inability to handle MATLAB's `load` built-in function is a limitation of our implementation. Programs that produce data by alternate means, such as by using the MATLAB random number generator `rand`, can be handled by our system.

Two benchmarks in the suite operate on three-dimensional arrays. As far as we know, our ability to handle arrays of arbitrary dimensions is a first because previous research compilers for MATLAB were limited to matrices. In fact, many publicly available benchmarks restrict their data to scalars, vectors and matrices. This is due to early versions of MATLAB only supporting matrices; multidimensional array support was introduced in version series 5, the current version series being 6.

10.2 Platform Specifications

All measurements were taken on a 440 MHz UltraSPARC-IIi workstation running Solaris 7 and having 128MB of main memory. The version of the MATLAB interpreter used was 6.1 (Release 12), while the version of `mcc` (The MathWorks' MATLAB-to-C compiler) used was 2.2.[1] Version 5.1 of the Sun Workshop C compiler was used for back-end compilation by both `mcc` and `mat2c` (the MAT2C program). Version 1.0 of MAGICA, invoked transparently by `mat2c` to gather type information, was executed on version 4.1 of the Mathematica kernel.

The back-end C compiler in both cases was passed the `-xO4` and `-xlibmil`

[1] In July of 2002, latest versions of both—6.5 (Release 13) and 3.0—were announced.

options that turn on a host of global and local optimizations, and inline certain library routines for faster execution. All optimizations offered by `mcc` were turned on. To minimize memory usage, the MATLAB interpreter was always run with the `-nojvm` option. This suppresses the loading of a Java virtual machine that allows a MATLAB session to draw on Java's capabilities.

10.3 Inferred Shape Compositions

Table 10.2 presents numbers pertaining to inferred shape information across the benchmark suite. The "Shapes" column displays the total number of inferred shapes in each benchmark. These include the shapes of the original program variables, as well as the shapes of temporaries that are introduced in the steps that lead up to type determination. For instance, temporaries get introduced during the SSA conversion step that precedes the type determination phase. It additionally includes variables that are introduced by the effective splitting of $\dot{\Phi}(P)(s, t)$ into copies during the SSA inversion phase. The "Explicit Shapes" column indicates the percentage of inferred shapes that are explicit. The column also shows the breakup of these explicit shapes between scalar and nonscalar shapes. Explicit shapes allow translators to generate highly efficient code. For example, if a translator could infer that the variables `a` and `b` in the MATLAB expression `a+b` are scalars, it can generate a simple machine instruction to perform the addition rather than a generalized array addition routine.

All nonexplicit shapes are symbolic. The "Symbolic Copies" column in Table 10.2 displays the percentage of symbolic shapes that are inferred to be identical. This deduction happens due to algebraic simplifications such as those discussed in Chapter 5. Establishing that one symbolic shape is exactly the same as another allows a translator to not only reduce the run-time array shape computation overhead but to also avoid unnecessary shape checks. Table 10.2 also shows that in benchmarks that have symbolic shapes, a usually large percentage of the symbolic shapes can be determined to be identical. This indicates that such inferences have the po-

Benchmark	Shapes	Explicit Shapes			Symbolic Copies (%)	Type Inference Timings (secs)	
		%	Scalar	Nonscalar		Kernel	MathLink
`adpt`	188	42.55	70	10	62.96	33.30	11.20 (25%)
`capr`	233	31.76	66	8	61.64	39.10	15.80 (29%)
`clos`	64	100	21	43	0.00	5.34	9.56 (64%)
`crni`	139	100	64	75	0.00	11.40	10.60 (48%)
`diff`	92	91.30	70	14	62.50	17.70	61.70 (78%)
`dich`	143	100	94	49	0.00	40.50	16.90 (29%)
`edit`	83	28.92	19	5	62.71	4.30	7.00 (62%)
`fdtd`	192	100	38	154	0.00	5.52	10.18 (65%)
`fiff`	88	100	58	30	0.00	5.16	11.04 (68%)
`nb1d`	163	10.43	12	5	64.38	11.40	6.80 (37%)
`nb3d`	118	16.10	14	5	37.37	7.24	5.56 (43%)

Table 10.2 Inferred Shape Compositions

tential to considerably lower shape-related run-time overheads. As far as we know, no previous approach to shape determination can make this kind of an inference.

10.4 Inference Timings

Timing measurements are presented in the last column of Table 10.2. The "Kernel" subcolumn displays the time taken by the Mathematica kernel to arrive at *all* type inferences—intrinsic type, value range and shape—for each benchmark.

The "MathLink" subcolumn records the time taken to transfer the intermediate representation of the input program across to the Mathematica kernel, and to transfer the inferred type information back to the front-end. This data movement, which occurs over a special interprocess communication interface called MathLink, can be expensive. As Table 10.2 shows, in 5 out of 11 benchmarks, the MathLink timings account for over 60% of the *total* type inference times. The MathLink channel has been improving with newer releases and it is possible that the total type inference timings will be closer to that shown in the "Kernels" subcolumn in future versions of Mathematica. Still, considering the quality of the inferences made, we believe that the current timings are acceptable, especially since a typical usage scenario is one where the interpreter is used for prototyping and code development, and an inference-capable compiler is used in the generation of the final production code.

10.5 Storage Reductions

Table 10.3 shows reductions in variable count and corresponding reductions in storage size due to the GCTD algorithm. Entries in the "Static/Dynamic Variable Reduction" column are of the form s/d where s is the number of variables whose array sizes are statically estimable and that get subsumed in another array by the GCTD pass, and where d is the number of variables whose sizes are statically inestimable (thus requiring dynamic allocation) but which can still be statically subsumed within

Benchmark	Static/Dynamic Variable Reduction	Original Variable Count	Storage Reduction (KB)
`adpt`	127/74	271	0.96
`capr`	84/75	301	0.68
`clos`	24/0	46	1216.14
`crni`	73/0	113	4055.85
`diff`	48/1	93	12.77
`dich`	82/0	107	144.90
`edit`	25/21	108	0.21
`fdtd`	111/0	168	4374.61
`fiff`	51/0	77	12712.92
`nb1d`	66/63	235	0.55
`nb3d`	58/54	191	0.59

Table 10.3 Array Storage Coalescing Reductions

another dynamically allocated variable because of the storage-size partial order $\preceq$.

The "Original Variable Count" column indicates the total number of variables in the CFG on entry to the GCTD pass. This includes variables in the program source, as well as temporaries introduced in preceding transformations like the SSA and SO form (see § 9.2.1) conversions. Other kinds of variables that also come under the coalescing regime of the GCTD pass, such as those assigned in symbolic shape tuple and rank expressions, are included in the above count.

Observe that for five benchmarks, d is 0. This is because MAGICA manages to explicitly infer all the shape tuples in them, causing all of their storage to be stack allocated by the GCTD algorithm. The corresponding reduction in bytes due to the coalescing of stack allocated variables is displayed in the "Storage Reduction" column. This figure is thus conservative in the savings in storage that it reflects because it doesn't include reductions due to the coalescing of heap allocated variables. Still, in 4 out of 11 programs, static reductions are seen to be over a megabyte, and close to 13 MB in `fiff`. This is on account of the large coalescent arrays

($\approx 451 \times 451$) that are operated upon by this benchmark. This is also the reason why with coalescing turned on, code generated by `mat2c` for `fiff` runs nearly two orders of magnitude faster than code generated by `mcc` (see Figure 10.8), and why without coalescing, code generated by `mat2c` is nearly six orders of magnitude slower (see Figure 10.9)! The key contributors to the relatively large reductions in variable count are the temporaries introduced as part of the SO form conversion process. Since these temporaries only serve to break long expressions into smaller ones, they have considerable potential for reuse both within and across expressions.

10.5.1 How `mcc` Handles Arrays

The `mcc` compiler relies on run-time type determination for its generated C code. In essence, the approach is to represent every array by a C `struct` called `mxArray`, which besides embedding the contents of the array, has a number of fields that contain meta information such as the array's shape and intrinsic type. These fields are set up at run time when arrays get created, and are examined and modified as conformance checks are performed and arrays evolve. A consequence is that all arrays in the `mcc`-generated C code are allocated on the heap. As a result, memory usage conservation also happens at run time where the library versions of each of the MATLAB operators have the onus of creating and deleting arrays. Array duplication due to copies is also minimized dynamically through sharing and a "copy-on-write" scheme. The way the automated memory management works is that `mxArray` structures created and returned within nested calls of library functions are deallocated immediately after being used. Another consequence of this run-time approach is that an `mxArray` structure, whose size in version 2.2 of `mcc` is 88 bytes, will be allocated for scalars that don't get folded at compile time.

This "library-based" model of compilation isn't unique to MATLAB; it has also been used in compilers for other similar typeless array-based languages like APL.

10.5.2 Dynamic Run-Time Memory Footprints

Figures 10.1 to 10.3 display the dynamic memory footprints of the `clos`, `adpt` and `capr` benchmarks, with some of their inputs set at certain values. These graphs show three improvements typically exhibited by `mat2c` C codes over `mcc` C codes:

1. The maximum storage levels associated with dynamic program data (stack and heap segments of the C code's process image) are often lower.
2. The maximum virtual memory levels (this is the total virtual memory size of a process and includes all of its swapped-out pages, mapped files and devices) as well as the maximum resident set levels (this is the physical memory size ascribed to the process) are usually lower.
3. Because execution times are very often much smaller, their *cumulative* memory usage levels, indicated by the so-called kcore-min value, are also much lower.

In the nomenclature of operating systems, a *segment* refers to a contiguous portion of a process's virtual address space. Examples include the text, data and stack segments that hold a process's execution instructions, initialized and uninitialized data (known commonly as the `.bss` section) and run-time stack respectively. A process could have multiple segments of the same kind. For example, in addition to the text segment that corresponds to its binary, it could have another text segment that corresponds to a shared library that gets subsequently loaded during execution.

When a function is invoked in a process, its run-time stack grows by a stack frame. This growth however is only *logical* because it occurs by the adjustment of the stack and frame pointers. It becomes actual when a reference to a virtual address in the run-time stack points to a nonexistent virtual page in the stack segment. Such a reference triggers a page fault and could occur early on, during the inception of the call itself when the various registers that encapsulate the machine's state are saved, or later on, when trailing elements in a large local array are accessed. In effect, this means that the stack segment will initially grow and finally reach a constant level, even though the run-time stack may continually expand and contract.[2]

[2]It should be noted that the stack segment can also shrink in size, but this action is beyond the user-process's direct control and happens due to a special kernel entity called the *page stealer*

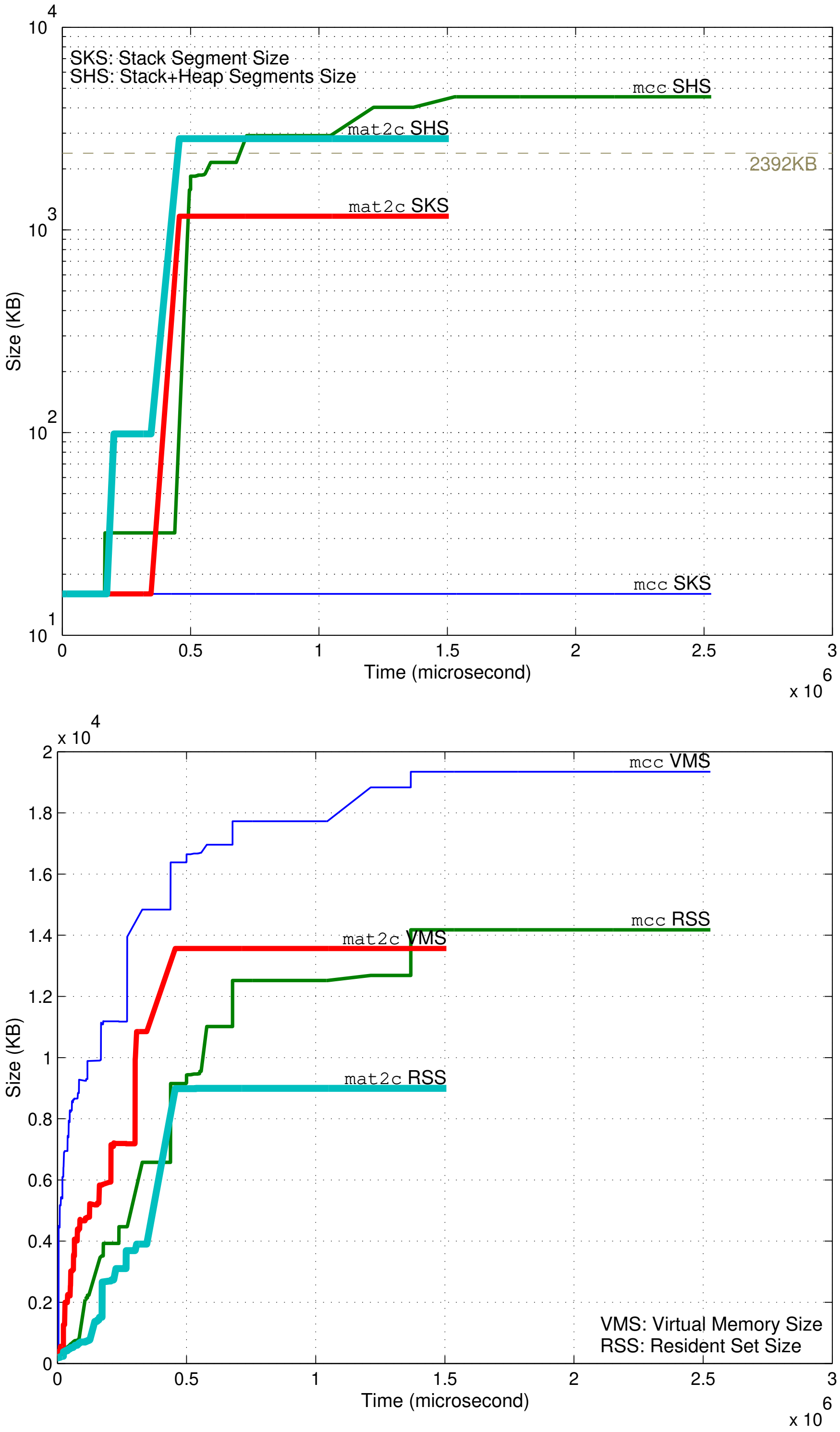

Fig. 10.1 Dynamic Memory Footprints for `clos` with $N = 256$

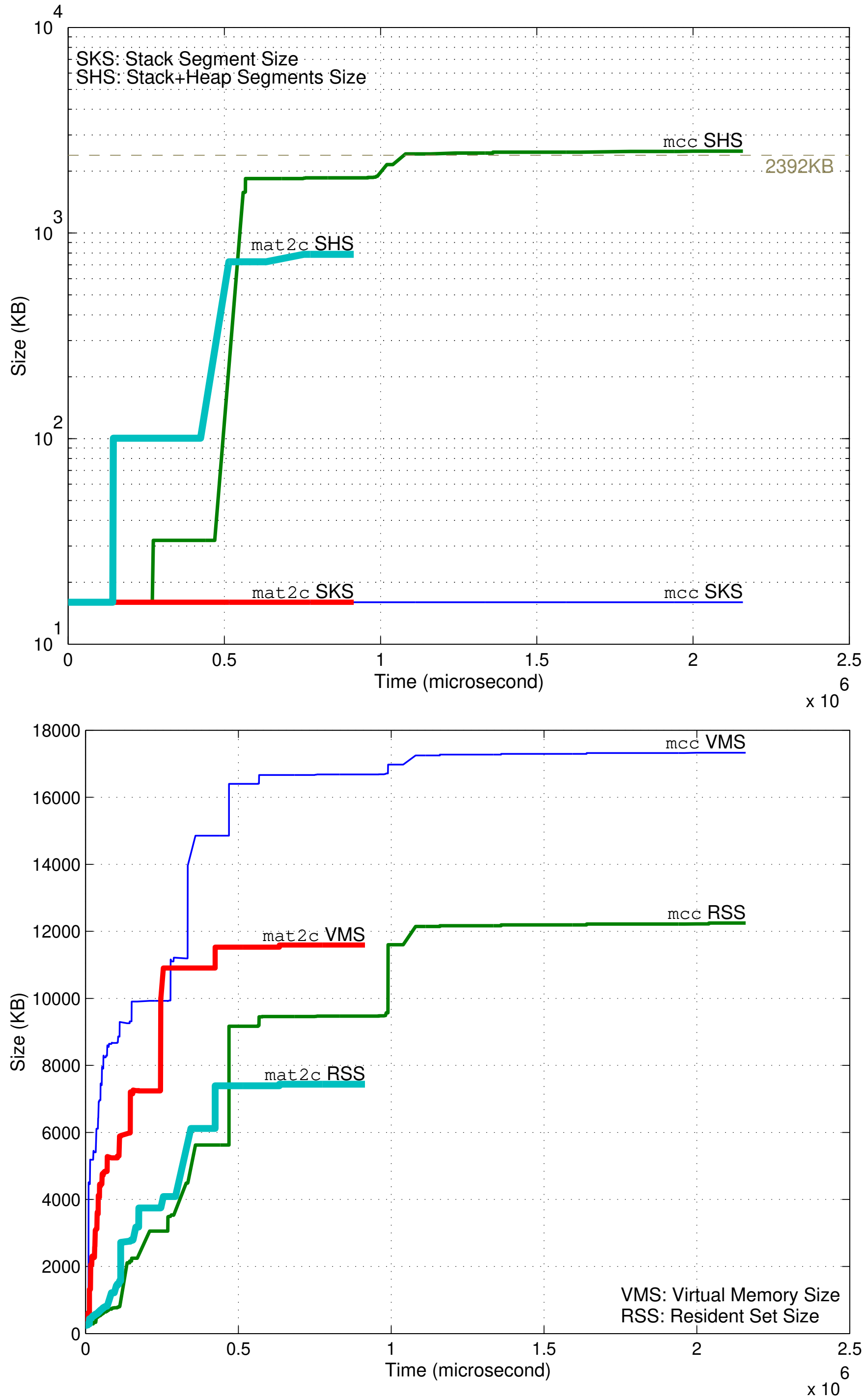

Fig. 10.2 Dynamic Memory Footprints for `adpt` with $tol = 10^{-9}$

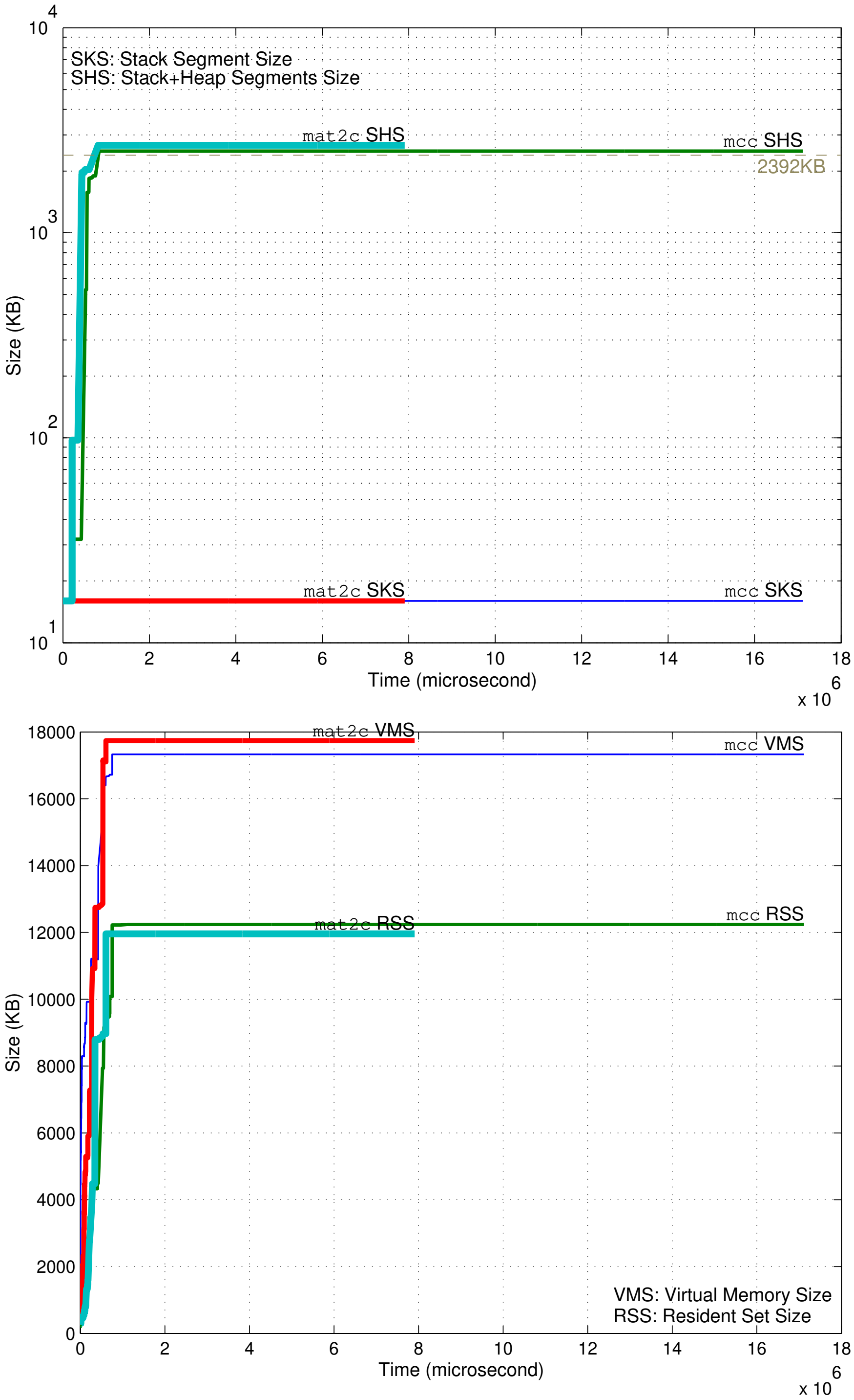

Fig. 10.3 Dynamic Memory Footprints for `capr` with a Random n

When a process begins execution, there is already one stack frame on its run-time stack that contains its initial environment. In a C program, this initial environment would include the `argc` and `argv` parameters of the `main` routine, as well as an array of strings that represents the process's environment variables. Thus, the stack segment, which grows in units of pages, will be initially at least one page in size, which is 8KB on the Solaris 7 UltraSPARC-IIi platform on which we ran our experiments. Further procedure invocations from `main` can cause the stack segment to grow depending on how local variables are allocated and referenced in the invoked functions. Because the `mcc` C codes rely on the heap for all array allocations and use function interfaces only for the passing and allocation of *handles* to these arrays, the high watermark of their run-time stacks shouldn't be expected to be large. Indeed, the dynamic memory footprint graphs show that the `mcc` C codes have a stack segment size that always stays at 16KB. In fact, as can be seen from Figure 10.5, 16KB was the average stack segment size for the `mcc` C codes of all benchmarks.

In the case of `mat2c` C codes, if a shape is statically determinable, the array gets allocated on the stack; this is the reason for the jump in the stack segment plot (labelled "`mat2c` SKS") in Figure 10.1. The jump occurs sometime after entry into the `closure` function from `main`, in an initialization loop when accesses to a 256×256 local double-precision array trigger page faults. However, when a shape isn't statically determinable, the array gets allocated on the heap. Only scalars, and pointers to the allocated arrays, exist on the stack. This is the reason why the stack segment plots for the `mat2c` C codes are also flat at 16KB in Figures 10.2 and 10.3.

In standard expositions of the UNIX operating system, the data segment is considered as comprising three parts: (1) a `.data` section that contains initialized data; (2) a `.bss` section that holds uninitialized data; and (3) a heap section from which memory gets dynamically allocated. In Solaris 7 however, it appears that only the first two parts form the data segment, with the third forming a separate segment by itself [68]. Because the `.data` and `.bss` sections are determined at compile time

daemon when available memory falls below a threshold.

```
% ( truss -u 'libmx:*' -u 'libc:*' \
? MCC/COutput/Benchmarks/MAT2C/drv_closure > /dev/null ) | & \
? sed -n '/mxCreateScalarDouble/,/mxCreateScalarDouble/ p'
                      ...
-> libmx:mxCreateScalarDouble(0x40080000,0x0,0x1e93e0,0x1e92a8)
                      ...
  -> libc:malloc(0x3ffc0, 0x0, 0x0, 0x0)
brk(0x001F90C0)                                       = 0
brk(0x002390C0)                                       = 0
  <- libc:malloc() = 0x1f8a40
                      ...
<- libmx:mxCreateScalarDouble() = 0x1f8a48
                      ...
```

Fig. 10.4 Condensed Internal Call Trace of `mxCreateScalarDouble`

and cannot change in size on the fly, the size of a process's data segment remains fixed during its execution. (The number of such data segments could of course change with time depending on whether the application loads shared objects.)

The highest virtual address in a process's heap space is called its *break value* and can be changed via the `brk` system call or the `sbrk` standard C library function. At the application level, memory can be allocated dynamically using the standard C library's `malloc` function, which is usually built on top of `sbrk` that is in turn built on top of `brk`. When arrays are run-time allocated in `mcc` and `mat2c` C codes, the `malloc` function gets ultimately called. This can actually be verified for any `mcc` C code by using the `truss` tool, as seen in Figure 10.4. The shown command line, issued under `tcsh`, traces calls into the `libmx.so` shared object provided by The MathWorks as part of their MATLAB distribution, as well as calls into `libc.so.1`, the standard C shared object. The former contains the `mx` family of procedures that enable arrays to be created while the latter contains the `malloc` implementation. In the condensed output snippet above, we see a pair of entry/exit points for the `mxCreateScalarDouble` library function, which is used for scalar array creation, as well as the entry/exit points for the `malloc` function that is invoked within `mxCreateScalarDouble`. The displayed invocation instance of malloc is particularly interesting because it grows the process's heap space by requesting the services of the operating system through the brk system call.

When an application is done using a dynamically allocated chunk of memory, it returns it back to the memory allocator using the free standard C library function. The memory allocator puts back the chunk into its free list of available memory blocks from which future memory requisitions through malloc are satisfied. More importantly, it doesn't relinquish the freed chunk to the operating system by lowering the process's break value through the brk or sbrk calls. A justification for this is the principle of locality by which it would be likely that memory requisition patterns would repeat themselves in the future. Rather than lower and later raise back the process's break value—both of which are comparatively expensive operations since they require the intervention of the operating system—the memory allocator keeps around the freed chunk of memory. It is because of this that the sums of the stack and heap segment sizes, shown by the SHS plots in the dynamic memory footprint graphs, form a nondecreasing sequence with time.

Specific trends can also be explained. For instance, in Figure 10.1, the SHS plot for the mat2c C code shows two distinct jumps, one before the same C code's SKS jump and one at around the same time. The first jump takes place due to the part in the mat2c C code that implements MATLAB's clock built-in function. (In all benchmarks, the invocation of the main function in the driver is surrounded by calls to clock to do an "in-file" timing of the invocation.) The mat2c C code implementation of clock uses the services of the standard C library's gettimeofday and localtime functions and it is the latter's use of malloc that causes the initial jump. The second jump, which is more pronounced, happens because of calls into malloc that originate from the dgemm function that implements the Level 3 BLAS operation of multiplying two general double-precision matrices. The dgemm routine used in mat2c C codes was exactly the one used by mcc in its generated C code; this was ensured by linking in the same libmwlapack.so shared library in both cases. The fact that both localtime (in libc.so.1) and dgemm (in libmwlapack.so, which is The MathWorks' implementation of LAPACK) eventually use malloc can once again be verified using truss.[3]

For the adpt benchmark, the two jumps in the SHS plot for the mat2c C code in Figure 10.2 occur for the same reasons discussed above, though in this case, there is one additional factor that contributes to the second jump: the mat2c C code also uses the heap space to allocate arrays in the adapt procedure, invoked from main, on account of a number of array shapes being symbolic in this procedure. The latter is also the reason why the mat2c C code uses dgemm to implement matrix multiplication in the adapt procedure. (The MAT2C translator resorts to using dgemm to implement matrix multiply under two situations: (1) if the array shapes are compile-time constants and the total number of elemental operations exceeds MTIMES_THRESHOLD, or (2) if the array shapes are symbolic.)

10.5.3 Average Run-Time Memory Footprints

Figures 10.5 to 10.7 display the average memory levels of the stand-alone C codes generated by mat2c with the GCTD pass turned on, and by mcc. Reductions in the dynamic program data sizes (stack plus heap) relative to the mat2c C codes are shown as percentages above the bars in Figure 10.5. Relative reductions in other categories of memory are also shown as percentages in Figures 10.6 and 10.7.

10.5.3.1 Average Stack Trends

Figure 10.5 shows four prominent peaks for the average stack segment size of the mat2c C codes for the clos, crni, fdtd, and fiff benchmarks. This is because mat2c allocates all arrays in these benchmarks on the stack. In the other benchmarks, significant percentages of the array shapes were symbolic; this led to storage being primarily allocated on the heap. The exception was dich in which although

[3]truss also revealed that The MathWorks' implementation of clock itself relies on the standard C library's gettimeofday and localtime functions, a revelation that was somewhat reassuring because it reinforced our confidence in our own implementation.

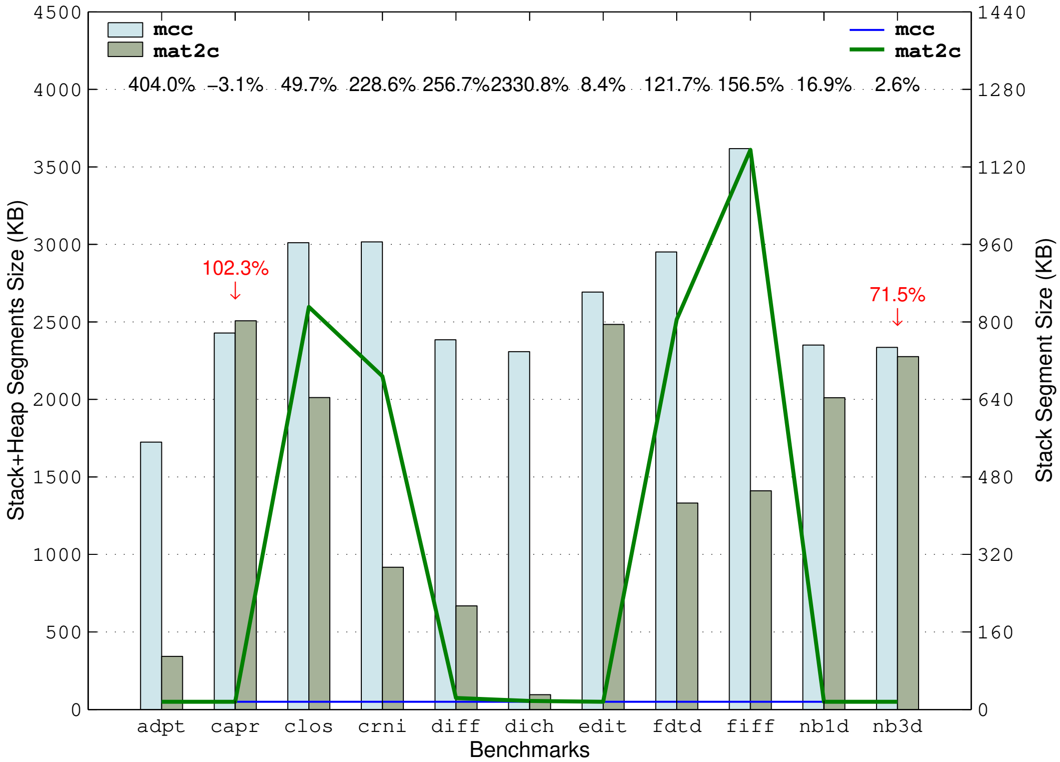

Fig. 10.5 Average Stack, and Stack+Heap Levels

100% of the array shapes were statically inferred, the overall average stack segment size was only about 17.5KB because most of the arrays were small.

10.5.3.2 Average Heap Trends

Figure 10.5 also shows the average stack and heap space sums across all benchmarks. All average memory sizes, be it stack, virtual memory or resident set, were calculated using a weighted time-averaged formula. If m_i was the observed memory size in some small duration Δt_i, then the average memory size M was calculated as

$$M = \frac{\sum_i m_i \Delta t_i}{\sum_i \Delta t_i}. \tag{10.1}$$

To ensure the interception of rapid fluctuations in m_i, Δt_i was made as small as possible. In our measurements, Δt_i was between 250 to 350 microseconds, which was usually about a thousandth to a hundredth thousandth of a C code's run time.

In general, the average dynamic program data sizes of the `mat2c` C codes were smaller than that of `mcc` C codes; the relative reductions were over 20% in 7 out of 11 cases, being over 100% in over half of the cases. In the case of `capr`, though the `mcc` C code fared better by a small margin, it still has a far higher kcore-min value as discussed in § 10.5.3.3 below. (For `capr`, the average dynamic program data sizes for the `mcc` and `mat2c` C codes were 2428.02KB and 2506.75KB.)

10.5.3.3 KCore-Min Reductions

An important point that the average figures don't uncover is the duration of consumption of a memory resource. If processes P_1 and P_2 both consume x kilobytes of memory, P_1 for t seconds and P_2 for $2t$ seconds, then both will have the same time-averaged memory consumption level but clearly P_2 will be the bigger memory hog. To take into consideration the effect of time, UNIX systems use a metric called the *kcore-min* value of a process that is computed as

$$\text{kcore-min} = M \times T,$$

where M is the mean memory size in kilobytes and T is the memory usage duration in minutes. Thus, although the average dynamic program data size of the `mat2c` C code was close to that of the `mcc` C code in `capr` and `nb2d`, the `mat2c` versions have lower kcore-min values because of shorter execution times. The relative kcore-min reductions were 102.3% and 71.5% in these two cases and are shown using arrow markers in Figure 10.5. Though not shown in Figure 10.5, the `mat2c` C codes for all other benchmarks also exhibit kcore-min reductions because besides having lower size averages, they also usually have significantly lower run times (see Figure 10.8).

10.5.4 Overall Memory Levels

To obtain the complete picture, the average virtual memory consumption levels of the `mat2c` and `mcc` C codes, which includes all swapped-out pages, mapped files and devices, was recorded. The measurements are shown in Figure 10.6. The

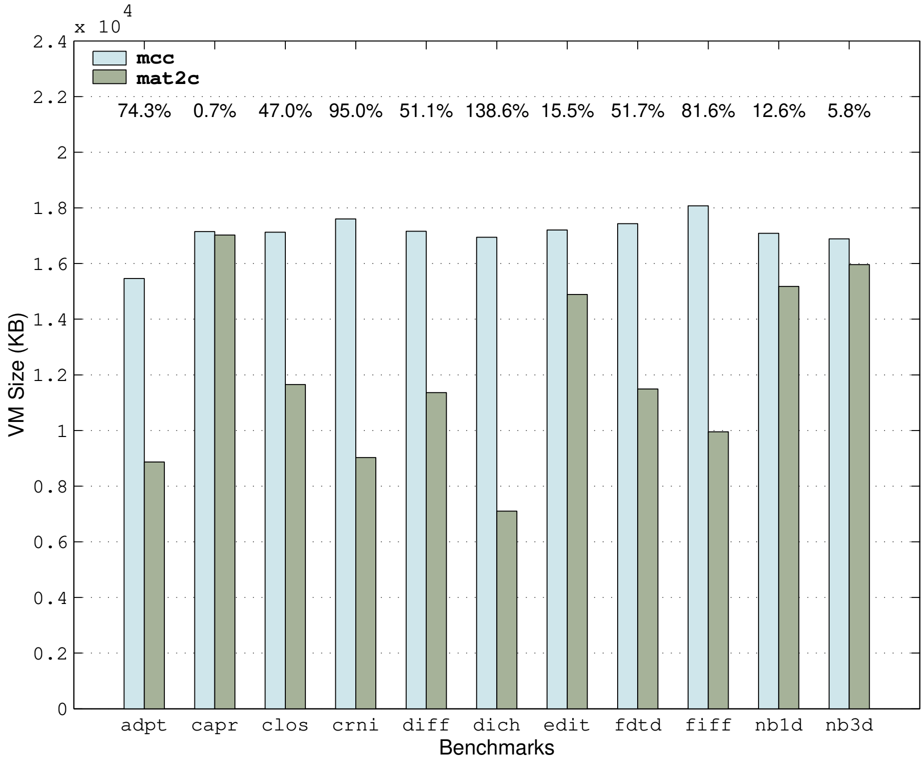

Fig. 10.6 Average Virtual Memory Levels

average resident set sizes (RSS) for all benchmarks are also shown, in Figure 10.7, which describe the amount of physical memory used by a process. The RSS numbers are significant, especially in an embedded setting, because non-resident pages don't task a RAM. Note that the sizes of the compiled images of the `mat2c` and `mcc` C codes also affect both these levels. However, it should be mentioned that the binary image size of a `mat2c` C code is nearly always larger than that of an `mcc` C code. The reason for this is that `mat2c` inlines out most of the language operations (this can be controlled by a pass-file option) whereas `mcc` always translates them to calls into a library. Hence, at the heart of the resource usage differences are the disparate approaches to compilation used by `mat2c` and `mcc`.

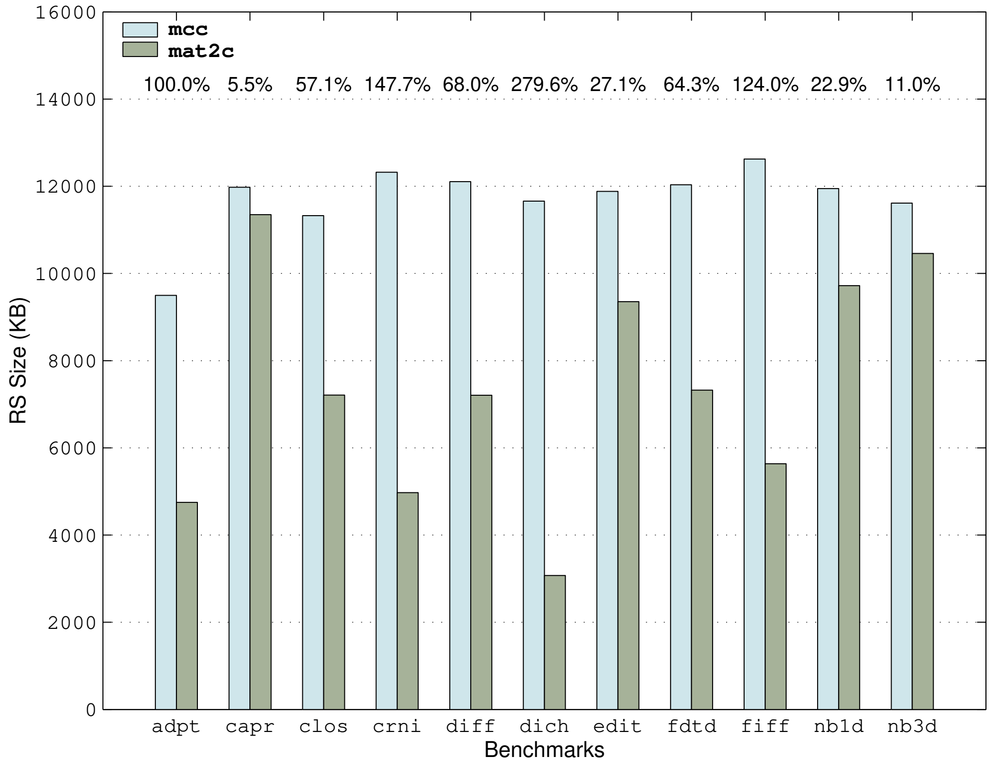

Fig. 10.7 Average Resident Set Levels

10.6 Execution Time Improvements

Figure 10.8 compares execution times of the `mat2c` and `mcc` C codes on a log scale. The figure also shows the times taken by the MATLAB interpreter to execute the benchmarks. Indicated above the bars are the performance speedups of the `mat2c` C codes over the `mcc` C codes. In only one case was the speedup marginal—10% for the `adpt` benchmark. The `adpt` benchmark has a parameter called *tol* that specifies the error tolerance in the quadrature values produced. Decreasing this value decreases the fractional overhead due to type checking, making numerical computation the dominant work in the benchmark. Measurements of run times for `adpt` were taken with *tol* set at 10^{-12}, which is the setting in the FALCON benchmark suite. Because increasing *tol* increases the relative overhead due to type checking, the `mat2c` C code performs much better than the `mcc` C code at higher values of *tol*. In all other benchmarks, the speedups ranged from 30% and 74% in two cases, to over 100% in the remaining. In fact, in 4 out of 11 benchmarks, the speedups were dramatic, being over an order of magnitude.

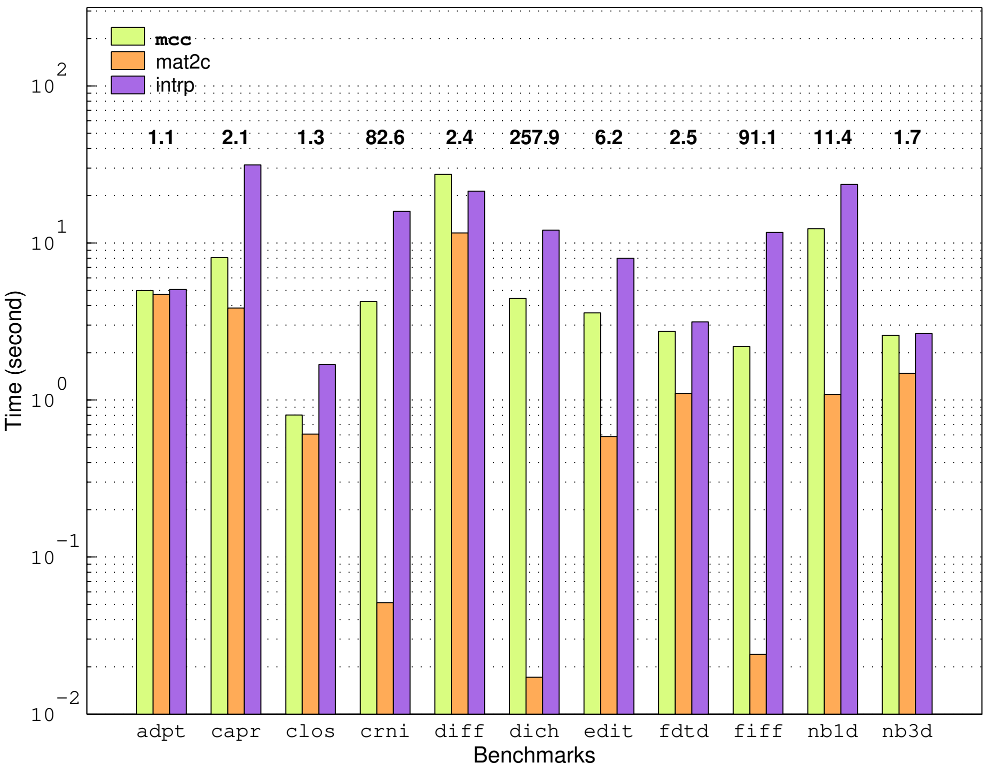

Fig. 10.8 Comparative Execution Times

10.6.1 Impact of the GCTD Algorithm

Figure 10.9 exhibits the influence of the GCTD pass on the execution times of the `mat2c` C codes. All other optimizations offered by the $\mathrm{M^{A}T_{2}C}$ system, like common subexpression elimination, constant folding, dead-code elimination and loop unrolling, were active in both cases. Numbers shown above the bars indicate the relative speedups with and without the GCTD pass. These relative speedups, in the context of the timings displayed in Figure 10.8, show that without it, the `mat2c` C codes would have performed poorly with respect to the `mcc` C codes in 8 out of 11

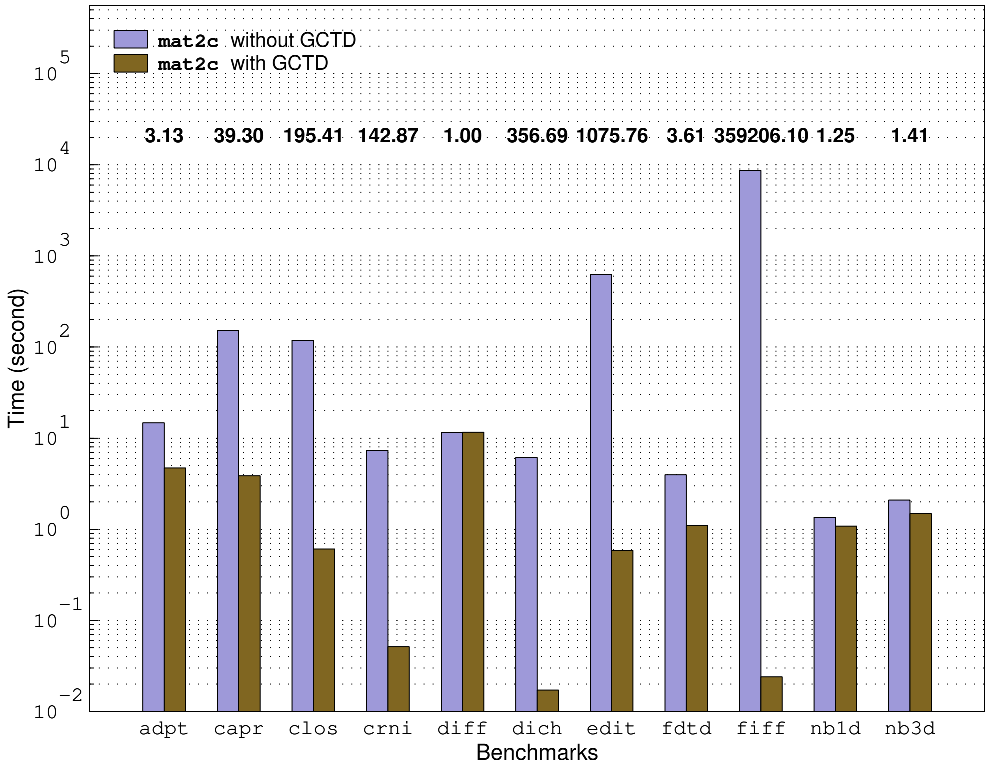

Fig. 10.9 Effect of Coalescing on Execution Times

cases. This demonstrates the pivotal role that the GCTD pass plays in improving performance through the better static management of storage.

10.6.2 JIT Execution on an Intel x86-Based Linux

In the latest release of MATLAB (Release 13), announced in July of 2002 [91], a capability to analyze program types at *run time* was incorporated. This section reports timing measurements from release 13, version 6.5 of MATLAB, and version 3.0 of the `mcc` compiler. The measurements were taken on a Dell Latitude C840, having a 1.60 GHz Intel Pentium 4 Mobile processor, with (slightly over) 512MB of RAM, and running Red Hat Linux 8.0 (kernel version 2.4.18-14).

Prior to MATLAB 6.5, the MATLAB interpreter processed programs in two stages: it translated the source into instructions in an intermediate language called

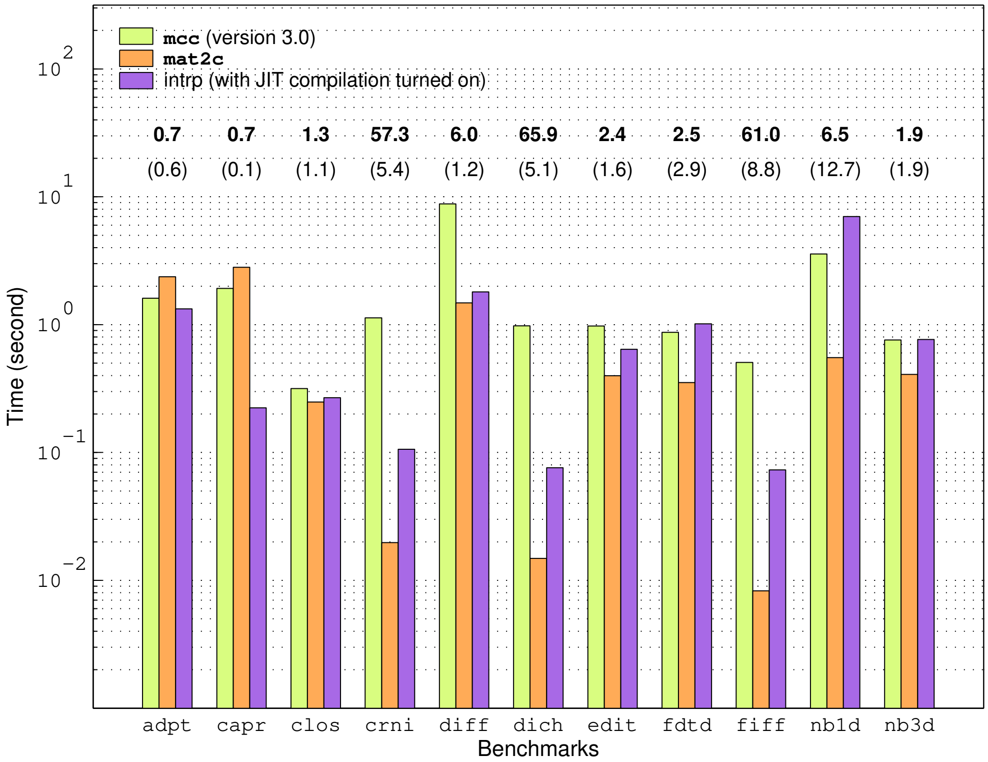

Fig. 10.10 Comparisons with a JIT Compiler on an Intel x86-Based Linux

p-code [66], which were then executed in sequence. Starting from MATLAB 6.5, a "JIT accelerator" has been integrated into the interpreter [90]. It speeds up execution using two methods: just-in-time (JIT) code generation and run-time type analysis. Under JIT code generation (a feature that is on by default in the MATLAB 6.5 interpreter), p-code instructions get converted into native code before execution. Run-time type analysis is used for producing specialized code, validating previously compiled code, and triggering subsequent recompilations if necessary [90].

Figure 10.10 compares the execution times with the JIT feature on, against the execution times of C codes generated using version 3.0 of `mcc` and a port of `mat2c` to Red Hat Linux 8.0. The first important observation is that the runs under the interpreter are now faster than the corresponding runs of the `mcc`-generated executables in 8 out of 11 programs. (Contrast this with Figure 10.8, which shows execution times under Solaris 7 using MATLAB 6.1 and version 2.2 of `mcc`.) The second important observation is that while `mat2c`-generated executables continue to be generally faster than `mcc`-generated executables, the margin of improvement is usually smaller vis-à-vis Figure 10.8. This improvement is shown as speedup numbers in boldface Helvetica font above the bars. In fact, in two cases—`adpt` and `capr`—the `mat2c` C codes exhibit an effective slowdown with respect to the `mcc` C codes. But in one situation, namely `diff`, the `mat2c` C code shows an even higher speedup of 6.0 in relation to the speedup of 2.4 in Figure 10.8.

The `mat2c` C codes also generally exhibit speedups with respect to the JIT-compiled codes. These speedups have been shown as numbers in parentheses above the bars in Figure 10.10. However, with respect to Figure 10.8, the speedup margins are usually even smaller. The exceptions are `fdtd`, `nb1d` and `nb3d` for which the `mcc` C codes are faster than the MATLAB JIT-compiled codes. It should be borne in mind that while a dynamic compilation system has the potential to produce highly tailored code, increasing the sophistication of such a system can itself have an adverse effect on performance due to increased resource usage footprints, especially on embedded platforms where resources are at a premium.

10.6.2.1 Effect of Input Variation on Run Times

For runs of the `adpt` benchmark in Figure 10.10, the *tol* parameter was set to 10^{-12}. The benchmark calculates quadrature values in a matrix whose size in the first dimension grows inversely with *tol*. Therefore, as mentioned in § 10.6, the overhead due to the type checking work in `adpt` becomes small relative to the numerical work at low *tol* levels, increasing as *tol* is increased. Hence, any benefits arising from the elision of type conformance checks and the ahead-of-time computation of output types should become visible at low values of *tol*. As Figure 10.11 shows, in the vicinity of $tol \approx 10^{-12}$, both the `mcc` C code and the JIT-compiled code outperform the `mat2c` C code because the latter hasn't been optimized for numerical efficiency. But at $tol \approx 10^{-10}$, the trends change as the type checking overhead starts to become

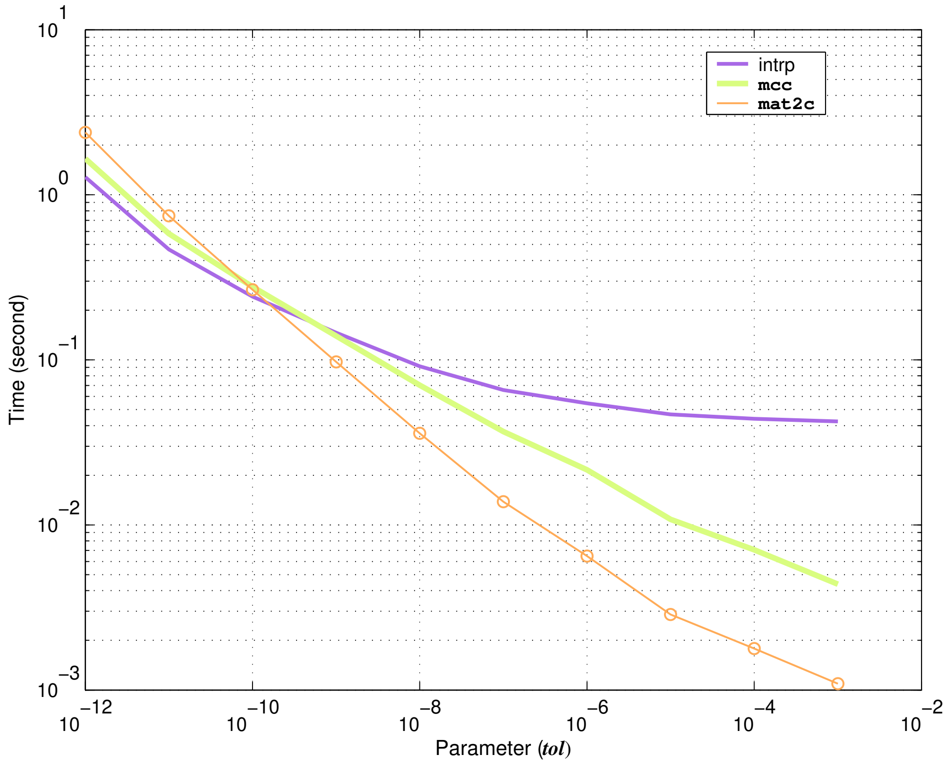

Fig. 10.11 Execution Times for `adpt` under Varying Error Tolerances

significant. Figures 10.12 and 10.13 similarly show the change in the running times of the `nb1d` and `nb3d` benchmarks as the number of bodies n in the simulation is varied. Circles along the `mat2c` curves in Figures 10.11, 10.12 and 10.13 mark points on the abscissae at which the measurements were taken.

10.6.2.2 Effect of Automatic Code Selection on Run Times

On the basis of inferred types, the MAT2C system chooses different implementations to realize a particular MATLAB operation. For example, when two small matrices are multiplied, `mat2c` will either fully unroll the computation or inline it by a standard triply-nested loop implementation. When the matrices multiplied are large, it uses BLAS calls instead. (As mentioned on page 157, the same `dgemm` BLAS routine is used in the translated code as the one used by `mcc`.) Figure 10.14 displays the effect

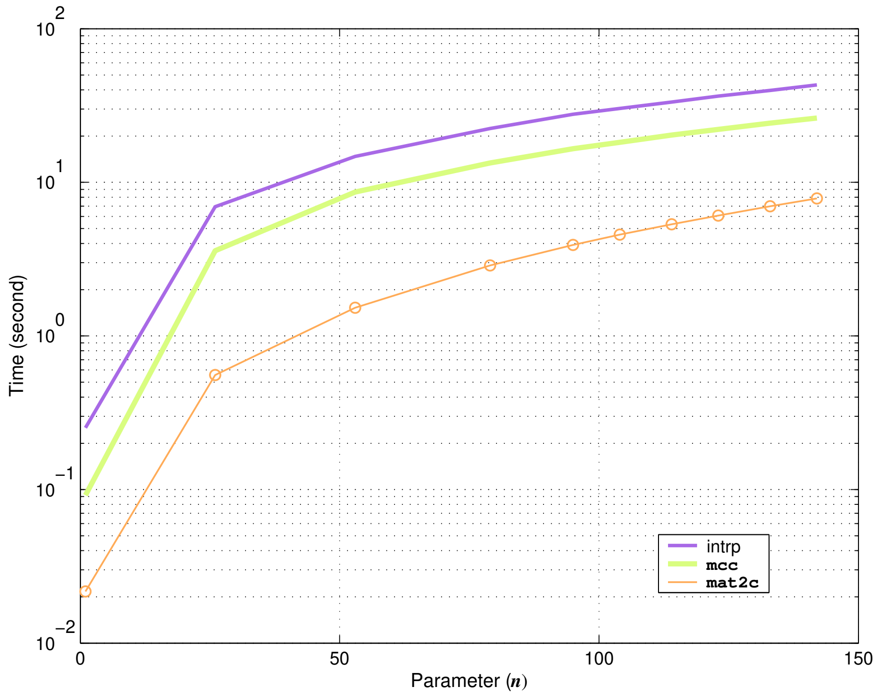

Fig. 10.12 Execution Times for `nb1d` under Varying Simulated Bodies

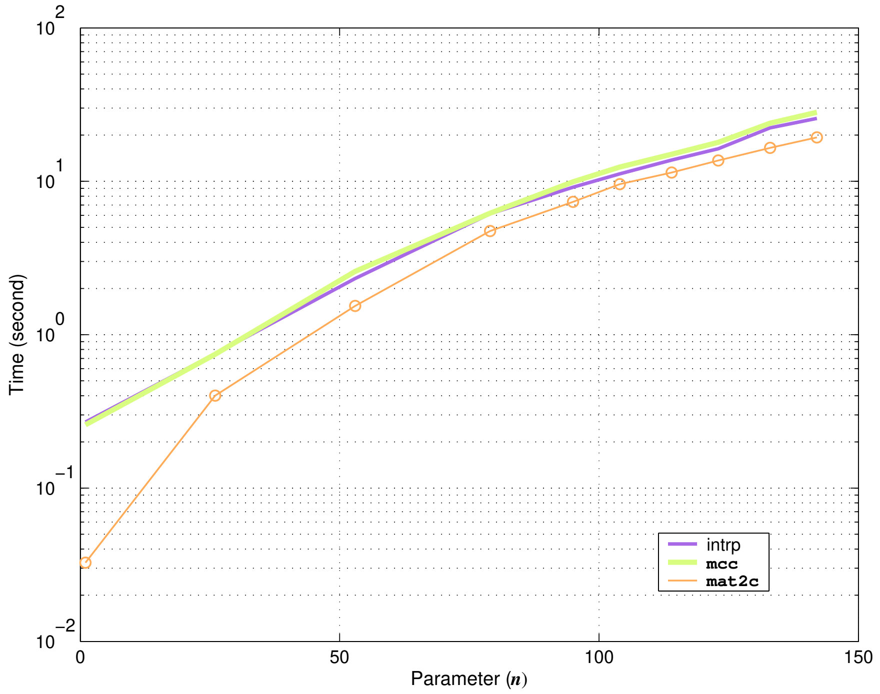

Fig. 10.13 Execution Times for `nb3d` under Varying Simulated Bodies

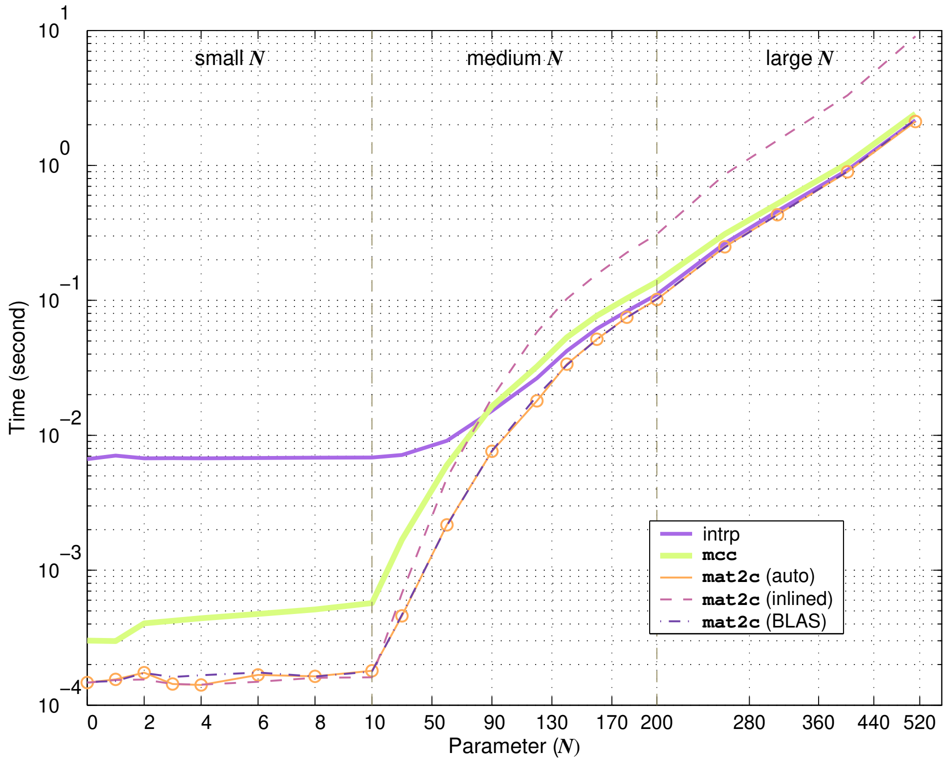

Fig. 10.14 Execution Times for `clos` under Automatic Code Selection

on the `clos` benchmark of choosing different implementations for the matrix multiplication operation. The benchmark has a matrix multiplication kernel in which an $N \times N$ adjacency matrix is repeatedly multiplied with itself $\log_2(N)$ times (see Figure 8.1 on page 115). Figure 10.14 shows three `mat2c` curves: one where matrix multiplication is always replaced by an inlined triply-nested loop implementation, another where the replacement is always with BLAS calls, and the third, marked as `auto`, where code is automatically selected. In the `auto` case, unrolling was used when N was at most 2, inlining was used when $2 < N \leq 5$, and BLAS was used when $N \geq 6$. (The "unrolling threshold" can be specified by the user through a pass file. The threshold for BLAS selection has been hard-coded in the current version of `mat2c`.) The most interesting observation is that under the standard loop implementation of matrix multiplication, the code begins to perform poorly (relative to the `mcc` and JIT-compiled codes) beyond $N \approx 90$. Observe also that below $N \approx 90$, the `mcc` C code fares better than the JIT-compiled code. This is perhaps due to the cost of dynamic compilation offsetting gains from run-time type determination.

Chapter 11

Conclusions

We know more about the dark side of the Moon than where we're going. We simply don't know what we'll find.
— CLIVE ROBERTS (marine scientist)
Comment before embarking on a multinational expedition to explore some of the deep-water regions of the Pacific Ocean.
As quoted in "Delight in Ugly Creatures," BBC News, July 1 2003.

The main impediment to efficiently compiling MATLAB and other dynamically typed array-based languages is the lack of explicit type information. A number of strategies to overcome this have been proposed in the past, ranging from "hand-holding" the compilation process through the insertion of type information in the form of comments, to static inference mechanisms that attempt to infer and propagate explicit type information where possible, and that resort to run-time schemes in situations when that information isn't explicitly compile-time ascertainable. The disadvantage with the former is that it doesn't encompass legacy code and also sacrifices the language's user-friendliness and attractiveness. The disadvantage with the latter is an overly conservative solution when array extents are compile-time unknowns or when program libraries are to be separately compiled (see 2.3.1.1).

11.1 Contributions

The chief departure of this work from previous efforts is that by operating on shapes symbolically, it deduces valuable information even when the shapes *per se* may be statically unknown, and it does so without expecting any kind of manual intervention, language modification or user-guided preprocessing of the source codes. The proposed approach is based on an algebraical modelling of the shape semantics of the language. The main contributions of this thesis can be divided into five parts:

1. An algebraic modelling of a typeless array-based programming language's shape semantics. The chief attraction of this modelling is that it enables precise characterizations that are independent of an array's dimensionality.
2. A systematic rewriting methodology that leverages the characterizations to symbolically evaluate an array expression's shape. At the heart of this technique is a novel symbolic evaluator that runs in $O(R\|s\|^3 \log \|s\|)$ time, where $\|s\|$ is the number of nodes in a rooted ordered tree representation of a shape term s, and R is an upper bound on an array's canonical rank.
3. A shape-tracking optimization that detects equivalent shapes, enabling a compiler or interpreter to eliminate redundant shape checks, and to generate specialized code even when shapes may not be compile-time knowns.
4. A preallocation optimization that determines the set of all possible shapes that an expression computed within a loop may assume.
5. An interference graph-based storage coalescing optimization that reuses array storage across MATLAB expressions. The algorithm consists of two phases the first of which uses the classic notion of interference to form classes of variables that don't compete for storage and the second of which decomposes those classes on the basis of program variable types and control flow. Unique aspects of these phases are the consideration of interference due to operator semantics, its resolution using types, and the use of symbolic type information in the decomposition of color classes. Rather than clumping together all arrays having symbolic shapes within a color class, which was observed to be a poor storage management policy, the second phase uses a partial order to facilitate better spatial reuse characteristics.

11.2 Directions for Future Work

There appear to be at least three areas for further research: (1) the use of symbolic techniques to enhance the inference process, (2) improved storage organization in the face of tacit array typing so as to alleviate memory footprint overheads, (3) the use of dynamic compilation techniques so as to better tailor the generated code.

11.2.1 Decision Diagrams

An important component of the shape inference framework is the efficient representation and manipulation of algebraic expressions. It would be interesting to investigate the applicability and integration of an Algebraic Decision Diagram (ADD) [10] or a Hybrid Decision Diagram (HDD) [25] for this purpose. Used extensively in the realms of formal verification, logic synthesis and hardware testing, HDDs and ADDs have canonicalization and equivalence detection properties, attributes that are necessary in the algebraic processing steps of our framework.

11.2.2 Lessening Memory Footprints

As discussed in Chapter 9, efficient storage management through the use of inferred type information can produce a significant gain in the performance of MATLAB codes. While the GCTD algorithm showed how this could be done even when the types may not be explicitly known at compile time, the algorithm is nevertheless nonoptimal. The simplest example, shown in Figure 11.1, is an interference graph that consists of three nodes A, B and C representing variables with identical intrinsic types, whose corresponding storage sizes, assuming all are statically estimable, are say 4, 2 and 3 units respectively. If the only edge in the interference graph is between A and B, its chromatic number is 2. However, the aggregate of the coalesced storages will differ depending on which minimal coloring is actually used—if B and C share the same color, the aggregate will be 7 units, whereas if A and C share the same color, the aggregate will be 6 units (see Figure 11.2). Thus, arriving at an optimal

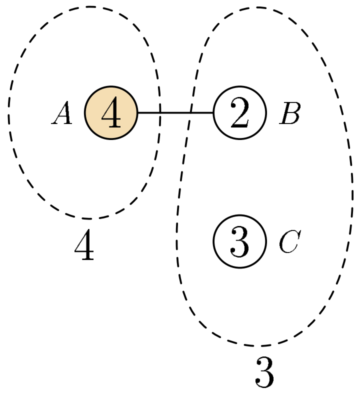

Fig. 11.1 Suboptimal Coloring

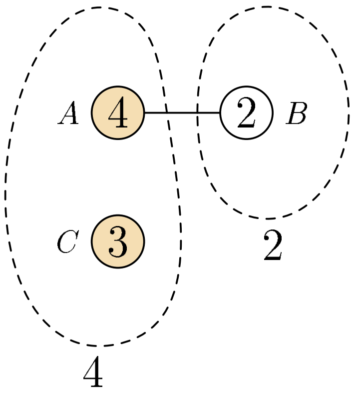

Fig. 11.2 Optimal Coloring

solution to the problem even in the simpler case of all array shapes being compile-time knowns would require an exploration of all possible colorings, a point that Fabri had also noted [38]. However, our experiments demonstrate that even with a conservative approach to the problem like ours, considerable savings in storage space and improvements in execution performance can be achieved.

Another avenue for research would be to see how garbage collection methods [57] could be combined with static allocation so as to achieve enhanced storage reuse.

11.2.3 Dynamic Compilation

Recently, Serrano et. al [86] presented a hybrid compilation model for Java that combined static and dynamic techniques. Their approach provided the benefits of static compilation (like incurring the cost of extensive "off-line" optimizations ahead of production runs) without sacrificing compliance with Java's dynamic features.

A similar technique was used in a dynamic incremental compiler for APL [49]. The idea was to generate code that matched a variable's type attributes on a particular execution run. The generated code was secured against changes in the run-time bindings by the inclusion of a "signature code" block [49] that contained details of the assumptions used in a compilation run. These were checked against on subsequent execution runs to either invalidate or continue using the previously compiled code. In the case of invalidation, code for the new situation was produced automatically by invoking the dynamic compiler. The premise was that such invalidations were occasional, and that more likely than not, a program's type attributes remained unchanged across executions. Such approaches may hold promise in MATLAB.

It should be noted that the inference system described in this thesis isn't limited to a purely static situation—it could also be used in a dynamic compilation setting. In fact, there is greater potential to obtain even more precise inferences because the initial information available to the inference process, such as the argument types at the call site of some function, will itself be more precise.

Bibliography

[1] Philip S. Abrams. *An APL Machine*. PhD thesis, Stanford University, February 1970. Available as Technical Report SLAC–114 from the Stanford Linear Accelerator Center.

[2] ACM SIGACT-SIGPLAN. *Proceedings of the 5th ACM SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1978.

[3] ACM SIGPLAN. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, June 2002.

[4] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *FORTRAN 90 Handbook*. McGraw-Hill Book Company, New York City, NY 10023, USA, 1992.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Computer Science Series. Addison-Wesley Publishing Company, Inc., Redwood City, CA 94065, USA, 1988.

[6] George Almási. *MaJIC: A MATLAB Just-In-Time Compiler*. PhD thesis, University of Illinois at Urbana-Champaign, June 2001.

[7] George Almási and David A. Padua. MAJIC: Compiling MATLAB for Speed and Responsiveness. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* [3], pages 294–303.

[8] Ed Anderson, Zhaojun Bai, Christian H. Bischof, L. Susan Blackford, James Demmel, Jack J. Dongarra, Jeremy Du Croz, A. Greenbaum, Sven Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Software, Environments, and Tools. Society for Industrial and Applied Mathematics, third edition, 1999.

[9] Arthur Anger, editor. *Proceedings of the ACM SIGAPL Conference on Array Processing Languages*. Association for Computing Machinery, May 1979.

[10] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic Decision Diagrams and their Applications. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*, pages 188–191. IEEE Computer Society Press, November 1993.

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

[12] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA 02061, USA, January 1993.

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

[14] Milissa Benincasa, Richard Besler, Diane Brassaw, and Ralph L. Kohler Jr. Rapid Development of Real-Time Systems Using RTExpress. In Sahni [85], pages 594–599.

[15] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[16] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring Heuristics for Register Allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 275–284. Association for Computing Machinery, July 1989.

[17] Timothy Budd. *An APL Compiler*. Springer-Verlag, Inc., New York City, NY 10010, USA, 1988.

[18] Timothy A. Budd. An APL Compiler for a Vector Processor. *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984.

[19] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6(1):47–57, January 1981.

[20] Arun Chauhan and Ken Kennedy. Optimizing Strategies for Telescoping Languages: Procedure Strength Reduction and Procedure Vectorization. In *Proceedings of the 15th ACM SIGARCH International Conference on Supercomputing*, pages 92–102, June 2001.

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

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

[23] Wai-Mee Ching and Alex Katz. An Experimental APL Compiler for a Distributed Memory Parallel Machine. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 59–68, November 1994.

[24] Frederick Chow and John Hennessy. Register Allocation by Priority-Based Coloring. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 222–232, June 1984.

[25] Edmund M. Clarke, M. Fujita, and X. Zhao. Hybrid Decision Diagrams: Overcoming the Limitations of MTBDDs and BMDs. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, pages 159–163. IEEE Computer Society Press, November 1995.

[26] Daniel Cociorva, Gerald Baumgartner, Chi-Chung Lam, P. Sadayappan, Jagannathan Ramanujam, Marcel Nooijen, David E. Bernholdt, and Robert Harrison. Space-Time Tradeoff Optimization for a Class of Electronic Structure Calculations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* [3], pages 177–186.

[27] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press, Cambridge, MA 02142, USA, 1995.

[28] Ron Cytron and Jeanne Ferrante. What's in a Name? The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

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

[30] Luiz Antônio De Rose. *Compiler Techniques for MATLAB Programs.* PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

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

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

[33] Nachum Dershowitz and David A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 9, pages 535–610. Elsevier Publishing, Sara Burgerhartstraat 25, Amsterdam, The Netherlands, 2001.

[34] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[35] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[36] Murray Eisenberg and Howard A. Peele. A Survey of "APL Thinking". *APL Quote Quad*, 21(2):5–8, December 1990.

[37] A. P. Eršov. Reduction of the Problem of Memory Allocation in Programming to the Problem of Coloring the Vertices of Graphs. *Doklady Akademii Nauk SSSR*, 142:785–787, January 1962. English translation in *Soviet Mathematics*, Vol., 3, No., 1, July 1962, pages 163–165.

[38] Janet Fabri. Automatic Storage Optimization. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 83–91, August 1979.

[39] Adin D. Falkoff. The Future of APL. *APL Quote Quad*, 16(2):6–13, 23–24, December 1985.

[40] Alan R. Feuer and Narain H. Gehani. A Comparison of the Programming Languages C and Pascal. *ACM Computing Surveys*, 14(1):73–92, 1982.

[41] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[42] Leonidas J. Guibas and Douglas K. Wyatt. Compilation and Delayed Evaluation in APL. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles of Programming Languages* [2], pages 1–8.

[43] Malay Haldar, Anshuman Nayak, Abhay Kanhere, Pramod G. Joisha, U. Nagaraj Shenoy, Alok Choudhary, and Prithviraj Banerjee. MATCH Virtual Machine: An Adaptive Run-Time System to Execute MATLAB in Parallel. In *Proceedings of the 2000 International Conference on Parallel Processing*, pages 145–152, August 2000.

[44] J. Roger Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[45] Parry Husbands, Charles Lee Isbell Jr., and Alan Edelman. MITMatlab: A Tool for Interactive Supercomputing. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, March 1999.

[46] Kenneth E. Iverson. Notation as a Tool of Thought. *Communications of the ACM*, 23(8):444–465, August 1980.

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

[48] Kathleen Jensen and Niklaus Wirth. *Pascal—User Manual and Report*. Springer-Verlag New York, Inc., New York City, NY 10010, USA, 1974.

[49] Ronald L. Johnston. The Dynamic Incremental Compiler of APL\3000. In Anger [9], pages 82–87.

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

[51] Pramod G. Joisha and Prithviraj Banerjee. Correctly Detecting Intrinsic Type Errors in Typeless Languages such as MATLAB. In *Proceedings of the ACM SIGAPL Conference on Array Processing Languages*, pages 6–21. Association for Computing Machinery, June 2001.

[52] Pramod G. Joisha and Prithviraj Banerjee. MAGICA: A Software Tool for Inferring Types in MATLAB. Technical Report CPDC–TR–2002–10–004, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, October 2002.

[53] Pramod G. Joisha and Prithviraj Banerjee. Implementing an Array Shape Inference System for MATLAB Using Mathematica. Technical Report CPDC–TR–2002–10–003, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, October 2002.

[54] Pramod G. Joisha and Prithviraj Banerjee. Static Array Storage Optimization in MATLAB. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 294–303. Association for Computing Machinery, June 2003.

[55] Pramod G. Joisha and Prithviraj Banerjee. The MAGICA Type Inference Engine for MATLAB. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 121–125. Springer-Verlag, April 2003.

[56] Pramod G. Joisha, U. Nagaraj Shenoy, and Prithviraj Banerjee. An Approach to Array Shape Determination in MATLAB. Technical Report CPDC–TR–2000–10–010, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, October 2000.

[57] Richard Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Inc., New York City, NY 10158, USA, 1997.

[58] Marc A. Kaplan and Jeffrey D. Ullman. A General Scheme for the Automatic Inference of Variable Types. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles of Programming Languages* [2], pages 60–75.

[59] Harry Katzan Jr. *FORTRAN 77.* Computer Science Series. Van Nostrand Reinhold Company, New York City, NY 10020, USA, 1978.

[60] Brian W. Kernighan. Why Pascal is Not My Favorite Programming Language. Technical Report 100, Bell Laboratories, April 1981.

[61] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, NJ 07458, USA, 1988.

[62] Kevin Knight. Unification: A Multidisciplinary Survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[63] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance FORTRAN Handbook.* Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA 02142, USA, 1994.

[64] S. S. Lavrov. Store Economy in Closed Operator Schemes. *Zhurnal vychislitel'noi matematiki i matematicheskoi fiziki*, 1(4):687–701, 1961. English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics*, Vol., 1, No., 3, 1962, pages 810–828.

[65] Alexey Malishevsky. Implementing a Run-Time Library for a Parallel MATLAB Compiler. Master's thesis, Oregon State University, April 1998.

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

[67] The MathWorks, Inc. *MATLAB Compiler*, September 2000. User's Guide (Version 2.1).

[68] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Architecture.* Sun Microsystems Press, Palo Alto, CA 94303, USA, 2001.

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

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

[71] Vijay Menon and Anne E. Trefethen. MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 1–18, November 1997.

[72] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[73] Igor Z. Milosavljević and Marwan A. Jabri. Automatic Array Alignment in Parallel MATLAB Scripts. In *Proceedings of the 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing*, pages 285–289. IEEE Computer Society Press, April 1999.

[74] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, MA 02142, USA, 1996.

[75] David V. Moffat. Conformant Arrays and Strong Typing. In *Proceedings of the ACM Conference*, pages 161–163. Association for Computing Machinery, November 1981.

[76] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, CA 94104, USA, 1997.

[77] Anshuman Nayak, Malay Haldar, Abhay Kanhere, Pramod G. Joisha, U. Nagaraj Shenoy, Alok Choudhary, and Prithviraj Banerjee. A Library-Based Compiler to Execute MATLAB Programs on a Heterogeneous Platform. In *Proceedings of the 13th IASTED International Conference on Parallel and Distributed Computing Systems*, August 2000.

[78] Patricia Pineo and Mary Lou Soffa. A Practical Approach to the Symbolic Debugging of Parallelized Code. In *Proceedings of the 5th International Conference on Compiler Construction*, pages 339–356, April 1994.

[79] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[80] Raymond P. Polivka and Sandra Pakin. *APL: The Language and Its Usage*. Automatic Computation Series. Prentice-Hall, Inc., Englewood Cliffs, NJ 07458, USA, 1975.

[81] Michael J. Quinn, Alexey Malishevsky, Nagajagadeswar Seelam, and Yan Zhao. Preliminary Results from a Parallel MATLAB Compiler. In Sahni [85], pages 81–87.

[82] Research Systems, Inc. *Building IDL Applications*, October 2002. IDL Version 5.6.

[83] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[84] Harry J. Saal and Zvi Weiss. A Software High-Performance APL Interpreter. In Anger [9], pages 74–81.

[85] Sartaj Sahni, editor. *Proceedings of the 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing*. IEEE Computer Society Press, April 1998.

[86] Mauricio J. Serrano, Rajesh Bordawekar, Samuel P. Midkiff, and Manish Gupta. Quasi-Static Compilation for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages – (to appear), October 2000.

[87] Paul L. Springer. Matpar: Parallel Extensions for MATLAB. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1191–1195. CSREA (Computer Science Research, Education, and Applications) Press, July 1998.

[88] Sun Microsystems, Inc. *Programming Interfaces Guide*, May 2002. Part No: 806–4125–10.

[89] Aaron M. Tenenbaum. *Type Determination in Very High-Level Languages*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, October 1974. Report NSO–3.

[90] Accelerating MATLAB: The MATLAB JIT-Accelerator. At `http://www.mathworks.com/mld_accel`.

[91] The MathWorks Announces Release 13 with Major New Versions of MATLAB and Simulink. At `http://www.mathworks.com/company/pressroom/index.shtml/article/332`.

[92] The MathWorks-MATLAB Compiler. At `http://www.mathworks.com/products/compiler/index.shtml`.

[93] J. P. Tremblay and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*. Computer Science Series. McGraw-Hill, Inc., New York City, NY 10121, USA, 1975.

[94] Christoph Walther. Many-Sorted Unification. *Journal of the Association for Computing Machinery*, 35(1):1–17, 1988.

[95] Jim Weigang. An Introduction to STSC's APL Compiler. In *Proceedings of the ACM SIGAPL Conference on Array Processing Languages*, pages 231–238. Association for Computing Machinery, May 1985.

[96] Eric W. Weisstein. Eric Weisstein's World of Mathematics. At `http://mathworld.wolfram.com/HilbertMatrix.html`, 1999.

[97] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–27, November 1998.

[98] Clark Wiedmann. Steps Toward an APL Compiler. In Anger [9], pages 321–328.

[99] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Inc., Champaign, IL 61820, USA, fourth edition, 1999.

[100] Youtao Zhang and Rajiv Gupta. Data Compression Transformations for Dynamically Allocated Data Structures. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 14–28, April 2002.

Vita

Pramod G. Joisha has been a graduate student in the Department of Electrical and Computer Engineering at Northwestern University, Illinois, since 1996. He received the B'Tech degree (Honors) in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur, in 1996, and the MS degree in Electrical and Computer Engineering from Northwestern University in 1998. His Masters defense was on a thesis titled "Compilation of Regular HPF Programs Using a Linear Algebra Framework."

After his final doctoral defense, he will be joining the Programmer Productivity Research Center at Microsoft Research as a researcher, starting fall of 2003. His research interests include parallel computing, high performance compilers, and programming languages.