MC2FOR : A MATLAB TO FORTRAN 95 COMPILER

by

Xu Li

School of Computer Science
McGill University, Montréal

April 2014

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Copyright © 2014 Xu Li

Abstract

MATLAB® is a dynamic numerical scripting language widely used by scientists, engineers and students. While MATLAB's high-level syntax and dynamic types make it ideal for fast prototyping, programmers often prefer using high-performance static languages such as FORTRAN for their final distribution. Rather than rewriting the code by hand, our solution is to provide a source-to-source compiler that translates the original MATLAB program to an equivalent FORTRAN program.

In this thesis, we introduce MC2FOR, a source-to-source compiler which transforms MATLAB to FORTRAN and handles several important challenges during the transformation, such as efficiently estimating the static type characteristics of all the variables in a given MATLAB program, mapping numerous MATLAB built-in functions to FORTRAN, and correctly supporting some MATLAB dynamic features in the generated FORTRAN code.

This compiler consists of two major parts. The first part is an interprocedural analysis component to estimate the static type characteristics, such as the shapes of the arrays and the ranges of the scalars, which are used to generate variable declarations and to remove unnecessary array bounds checking in the translated FORTRAN program. The second part is an extensible FORTRAN code generation framework automatically transforming MATLAB constructs to equivalent FORTRAN constructs.

This work has been implemented within the McLab framework, and we evaluated the performance of the MC2FOR compiler on a collection of 20 MATLAB benchmarks. For most of the benchmarks, the generated FORTRAN program runs 1.2 to 337 times faster than the original MATLAB program, and in terms of physical lines of code, typically grows only by a factor of around 2. These experimental results show that the code generated by MC2FOR performs better on average, at the cost of only a modest increase in code size.

Résumé

MATLAB® est un langage de script dynamique très utilisé par les scientifiques, les ingénieurs et les étudiants. La syntaxe de haut niveau et le typage dynamique de MATLAB en font un langage idéal pour faire du prototypage rapide, mais les programmeurs préfèrent souvent utiliser des langages statiques performants comme FORTRAN pour la distribution finale. Au lieu de réécrire le code à la main, notre solution est de proposer un compilateur qui traduit le programme MATLAB original vers un program FORTRAN équivalent.

Dans cette thèse, nous introduisons MC2FOR, un compilateur qui transforme MATLAB vers FORTRAN et surmonte plusieurs difficultés importantes rencontrées durant la transformation, dont celles d'estimer efficacement le type statique de toutes les variables dans un programme MATLAB donné, de trouver une correspondance pour les nombreuses fonctions intégrées de MATLAB vers FORTRAN et de supporter correctement quelques caractéristiques dynamiques de MATLAB dans le code FORTRAN généré.

Le compilateur est constitué de deux parties majeures : la première partie est une analyse interprocédurale qui estime des caractéristiques du type statique, comme la forme des tableaux et les limites des scalaires, qui sont utlilisées pour générer des déclarations de variables et pour supprimer les vérifications de limite de tableaux inutiles dans le programme FORTRAN généré. La deuxième partie est un framework de génération de code extensible qui transforment automatiquement des constructions de MATLAB vers des constructions de FORTRAN équivalentes.

Ce travail a été implementé dans le framework *Mc***LAB**, et nous avons évalué les performances du compilateur MC2FOR sur une collection de 20 programmes MATLAB. Pour la plupart des programmes, le programme FORTRAN généré s'éxécute entre 1.2 et 337 fois plus rapidement que le programme MATLAB original, et en termes de lignes de code, gran-

dit seulement par un facteur de deux. Ces résultats expérimentaux démontrent que MC2FOR est en mesure de générer du code qui performe mieux en moyenne que l'original sans pour autant augmenter de trop sa taille.

Acknowledgements

It's a great pleasure to work with Professor Laurie Hendren. Her positive attitude and high standard in research always help us towards the best solution. I would also like to thank all the other members in the lab, especially Matthieu Dubet, Vineet Kumar, and Ismail Badawi, we help and learn from each other both in work and in life. Since the very beginning of my life, my parents become my first teachers and friends. They guide me how to survive in this wonderful but also tough world and support me to chase my dream without any hesitation. Finally, and also the most importantly, I would like to thank SU Qingyuan, who is my best friend, my girlfriend, my wife and my soulmate. We met as strangers, we fell in love at the first glance, and we support and trust each other more than families. She is the source of my strength to get through all the difficulties in my master program and all the challenges in the future.

Table of Contents

Abstract i

Résumé iii

Acknowledgements v

Table of Contents vii

List of Figures xi

List of Tables xiii

Table of Contents xv

1 Introduction 1

2 Background and Overview 5

2.1 Potential Challenges 6

2.2 Overview of MC2FOR 6

2.3 Components in MC2FOR 8

3 Shape Analysis 11

3.1 Propagating Shapes through MATLAB Built-in Functions 12

3.1.1 Typical Behaviors of Built-in Functions on Shapes 14

3.1.2 Features of the Language 16

3.1.3 Shape Propagation Equation Language 20

3.1.4 Shape Matching Algorithm 30
3.1.5 Summary 34
3.2 Merging Different Shapes 35
3.2.1 Merging Strategy 35
3.2.2 Merging Shapes in Loop Statements 39
3.3 Shape Analysis Result Verification 41

4 Range Value Analysis **43**
4.1 Propagating Ranges through Built-in Functions 49
4.2 Merging Different Range Values 57
4.3 Propagating Shapes through Array Indexing 59
4.3.1 Brief Introduction of Array Indexing in MATLAB 60
4.3.2 For Array Set Statement 64
4.3.3 For Array Get Statement 67

5 Transforming MATLAB to FORTRAN 95 **71**
5.1 Introduction 72
5.1.1 Why FORTRAN 95? 72
5.1.2 Potential Problems 74
5.2 Basic Transformations 75
5.2.1 Types 76
5.2.2 Variable Declarations 76
5.2.3 Built-in Functions 77
5.2.4 Control Flow Statements 83
5.2.5 User-defined Functions 84
5.3 Advanced Problems in Mapping Types 85
5.3.1 For Subscripts in Array Indexing 86
5.3.2 For Loop Range Expressions 86
5.3.3 Assigning Multiple Types to the Same Variable 86
5.4 Array Indexing Transformation 89
5.4.1 For Array Get Statements 89

5.4.2 For Array Set Statements . . . 91
5.4.3 Shortcut Linear Indexing Transformation . . . 92
5.5 Run-time Array Bounds Checking and Variable Resizing . . . 93
5.5.1 For Array Get statements . . . 93
5.5.2 For Array Set statements . . . 94
5.5.3 For Assignment Statements . . . 95

6 Experimental Results **97**
6.1 Description of the Benchmarks . . . 98
6.2 Experimental Results . . . 100
6.3 Analysis of Results . . . 103
6.3.1 MC2FOR and MATLAB Coder vs. MATLAB . . . 103
6.3.2 MC2FOR vs. MATLAB Coder . . . 105
6.3.3 MC2FOR without Checks vs. with Checks . . . 106
6.4 Summary . . . 108

7 Related Work **109**

8 Conclusions and Future Work **111**

Appendices

A Shape Propagation Equation Language **113**
A.1 Tokens . . . 113
A.2 Grammar . . . 114
A.3 Some Shape Equation Examples . . . 117
A.4 Implementation Details . . . 119
A.5 The Aspect File to Detect Array Growth . . . 123

Bibliography **129**

List of Figures

2.1 The overview of MC2FOR. We highlight the boxes which are the contributions of this thesis. 7

3.1 MATLAB code examples of merging two different shapes 36
3.2 The analysis result of MATLAB code examples in Figure 3.1 38
3.3 MATLAB code example of merging shapes with variant dimensions 39
3.4 The profiling results of the benchmark `adapt` by AspectMatlab and shape analysis 42

4.1 MATLAB code example of array growth 44
4.2 MATLAB code example of array growth with non-constant value 45
4.3 Generated FORTRAN code v1.0 for the MATLAB code in Figure 4.2 (b) 46
4.4 Generated FORTRAN code v2.0 for the MATLAB code in Figure 4.2 (b). 48
4.5 MATLAB code example of merging range values in loops 58
4.6 MATLAB code example of out-of-bound array set assignment 60
4.7 Illustration of MATLAB array indexing 62
4.8 MATLAB script example of linear indexing 63
4.9 MATLAB script example of array growth 64

5.1 MATLAB code example of using `mldivide` built-in function 79
5.2 Generated FORTRAN code for the code example in Figure 5.1 81
5.3 FORTRAN module for mapping MATLAB built-in function `mldivide` 81
5.4 MATLAB code example to illustrate function overloading 82
5.5 Generated FORTRAN code for the code example in Figure 5.4 82
5.6 FORTRAN module ones supporting function overloading 83

5.7 if constructs in MATLAB (left) and FORTRAN (right) 84
5.8 for loop constructs in MATLAB (left) and FORTRAN (right) 84
5.9 while loop constructs in MATLAB (left) and FORTRAN (right) 84
5.10 Translating entry point function from MATLAB to FORTRAN 85
5.11 Subscript in MATLAB (left) and FORTRAN (right) 86
5.12 Transformation of for loop constructs in MATLAB to FORTRAN 87
5.13 Using derived data type in FORTRAN to map inconvertible types of the same variable in MATLAB 88
5.14 The function ARRAY_GET3SV 90
5.15 The subroutine ARRAY_SET3SV2 92
5.16 Run-time array bounds checking code for array get statement 94
5.17 Run-time array bounds checking and variable reshape code for array set statement 95
5.18 Illustration of allocating allocatable arrays in user-defined functions 96

6.1 Constant value replacement for the power function in FORTRAN 106

A.1 Example to illustrate shape matching process 121

List of Tables

3.1 Built-in functions of shape propagation equation language 26
3.2 Shape merging relation table 40

4.1 Operators supported by the range value analysis 49

5.1 Mapping MATLAB types to FORTRAN 76
5.2 Mapping MATLAB arithmetic operators to FORTRAN 78
5.3 Mapping MATLAB relational operators to FORTRAN 78
5.4 Mapping MATLAB logical operators to FORTRAN 79
5.5 Directly mapping MATLAB commonly used mathematical built-ins to FORTRAN 80

6.1 Performance comparison 101
6.2 Physical lines of code comparison 104
6.3 MC2FOR without and with checks 107

List of Listings

2.1 MATLAB implementation of Babai algorithm 5
3.1 Shape matching algorithm 1 of 5: function matchShape 31
3.2 Shape matching algorithm 2 of 5: function case.match 31
3.3 Shape matching algorithm 3 of 5: function patternlist.match 32
3.4 Shape matching algorithm 4 of 5: function expression.match 32
3.5 Shape matching algorithm 5 of 5: function outputlist.match 34
3.6 Shape merging strategy 36
3.7 The equals function to check whether the analysis in loops gets to the fixed point 37
4.1 Unary plus operator (+) 52
4.2 Binary plus operator (+) 52
4.3 Unary minus operator (-) 52
4.4 Binary minus operator (-) 53
4.5 Element-wise multiplication operator (.*) 53
4.6 Matrix multiplication operator (*) 53
4.7 Element-wise rdivision operator (./) 54
4.8 Matrix rdivision operator (/) 54
4.9 Natural logarithm operator (log) 54
4.10 Exponential operator (exp) 55
4.11 Absolute value operator (abs) 55
4.12 Colon operator (:) 55
4.13 Range value merging strategy 57
4.14 Range value equals function 57

4.15 Shape analysis for array set statements 65
4.16 Shape analysis for array get statement 68
5.1 Variable declaration in FORTRAN 77
A.1 The aspect file to detect array growth 123

Chapter 1
Introduction

MATLAB [Matb] is a well established language commonly used by engineers, scientists and students. This user community finds MATLAB convenient for prototyping their applications because of MATLAB's flexible syntax, the fact that no static declarations are required, the availability of many high-level array operators, and access to a rich set of built-in functions. However, once the user has developed their prototype application, he/she often wants to move to a more traditional high-performance scientific language such as FORTRAN.

There are two compelling reasons to make such a transition to FORTRAN. Firstly, the user may want high-performance code, which can be freely distributed. If the application has been translated to FORTRAN, then the user may compile the code with any of the numerous high-performance optimizing FORTRAN compilers, including open source compilers like GFortran [GNU13]. Secondly, the prototyped MATLAB code may implement a function which needs to be integrated into an existing system already implemented in FORTRAN. For example, a weather forecasting system may use many different models, and new models must be implemented in FORTRAN for integration into the system.

Given that converting from MATLAB to FORTRAN is a common problem, our goal is to make this easy for programmers by providing MC2FOR, a source-to-source compiler that transforms MATLAB programs to FORTRAN. This compiler enables MATLAB users to move their applications from MATLAB to FORTRAN without the effort and knowledge required of manually rewriting their code in FORTRAN. To be generally useful our compiler needs to: (1) be *easy* to use, (2) produce *efficient* FORTRAN code, and (3) produce *readable*

FORTRAN code.

Although MATLAB's roots are as a simple scripting language to interface with FORTRAN libraries,[1] modern MATLAB has evolved into quite a complex language, with syntax and semantics that have grown somewhat organically. Thus, although there is natural match between many array operations available in MATLAB and FORTRAN, there is actually a large gap between the dynamic nature of MATLAB and the statically-compiled nature of FORTRAN. As one example, in MATLAB there are are no variable declarations, and variables may hold any type, and in fact may hold different types at different program points. Whereas in FORTRAN all variables must be statically declared and must have well-defined types. Thus, to perform an automatic translation, our compiler must implement sophisticated static analyses, including a mechanism to analyze the many built-in functions.

The main contributions of this thesis are as follows:

Identified need/challenges: We have identified the need for a compiler to help programmers transform MATLAB to FORTRAN, and we have identified the main challenges.

Shape Analysis: We have designed and implemented an interprocedural shape analysis that estimates the number and extent of array dimensions, including handling built-in functions via a domain-specific language for expressing shape propagation rules through the functions.

Range Analysis: We have implemented a custom range analysis for MATLAB scalar variables in order to minimize the overhead of inlined array bounds checking and array shape resizing in the generated FORTRAN code.

Code Generation Strategies: We have designed and implemented code generation strategies for mapping the different types of variables, the simple control constructs, and the more difficult aspects of MATLAB.

Tool Implementation and Empirical Evaluation: We have implemented the tool as an open source project (`www.sable.mcgill.ca/mclab/mc2for.html`), and we have evaluated the tool on a suite of 20 benchmarks, showing that we can produce both efficient and compact code.

[1] wwww.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html

The thesis is structured as follows. In Chapter 2, we give the necessary background and overall structure of our compiler. In Chapter 3, we provide a detailed explanation of our shape analysis, including our approach for handling the shape propagation through MATLAB built-in functions. Chapter 4 describes our approach to range analysis, which is used to minimize the inlined array bounds checking and array shape resizing. Chapter 5 introduces the transformation to map the different types of variables, the simple control constructs, and the more dynamic features of MATLAB to FORTRAN. Chapter 6 provides our empirical study of using the compiler on a collection of 20 MATLAB benchmarks. Chapter 7 discusses some related works, and finally we conclude the whole thesis in Chapter 8.

Chapter 2
Background and Overview

MATLAB is widely used to prototype code for algorithms, implement solutions to complicated mathematical problems, and even run simulations for systems. Based on its array and dynamic language nature, MATLAB is especially suitable for solving linear algebra problems. For example, Listing 2.1 shows a MATLAB implementation of a well known linear algebra algorithm, the Babai nearest plane algorithm.

```
1  function z_hat = babai(R,y)
2  % compute the Babai estimation
3  % find a sub-optimal solution for min_z ||R*z-y||_2
4  % R - an upper triangular real matrix of n-by-n
5  % y - a real vector of n-by-1
6  % z_hat - resulting integer vector
7    n=length(y);
8    z_hat=zeros(n,1);
9    z_hat(n)=round(y(n)./R(n,n));
10
11   for k=n-1:-1:1
12     par=R(k,k+1:n)*z_hat(k+1:n);
13     ck=(y(k)-par)./R(k,k);
14     z_hat(k)=round(ck);
15   end
16 end
```

Listing 2.1 MATLAB implementation of Babai algorithm

This algorithm is an approximation to solve the closest vector problem and has pervasive applications in the field of wireless communication. Imagine that we want to transform this MATLAB implementation to FORTRAN- what potential problems may we encounter?

2.1 Potential Challenges

First of all, how should we declare the MATLAB variables in the transformed FORTRAN program? MATLAB is a dynamic scripting language which doesn't need variable declarations (although for readability MATLAB programmers often put some informal type information as comments), while in FORTRAN, to declare an array variable, we need to know at least the type and the number of dimensions of the variable, which means that in order to transform MATLAB to FORTRAN, first we need to find some way to obtain the type and shape information of all the variables in the given MATLAB program. Secondly, assuming that we can correctly declare all the variables, how should we map those built-in functions in MATLAB to FORTRAN? For example, in Listing 2.1, how should we map the `length` function at line 7, the `zeros` function at line 8 and the `round` function at lines 9 and 14. Thirdly, besides these two significant problems, we also need to think about how to map MATLAB constructs to the equivalent constructs in FORTRAN and how we should handle the differences between MATLAB and FORTRAN. For example, in MATLAB the programmer may leave out some of the trailing indices in an array reference, and the missing dimensions will be linearized, while in FORTRAN the number of the indices must be the same as the number of dimensions of the accessed array. Further, how should we map dynamic features such as the MATLAB behaviour that automatically grows an array when a write to that array is out of bounds?

2.2 Overview of MC2FOR

In order to solve these problems, we designed and implemented MC2FOR, as illustrated in Figure 2.1. First, let's focus on the input (top of figure) and output (bottom of figure) of MC2FOR. Note that the user only provides the name of the MATLAB file which is the entry point of the user's program to the compiler. Any other MATLAB files that may be used by

the program should be in the same directory as the entry point function file. If the entry point function has one or more input parameters, then the user should also provide the type and shape information for each of the parameter(s). The MC2FOR compiler then finds all functions reachable directly or indirectly from the entry point, loads the necessary files, and translates all the reachable MATLAB functions to equivalent FORTRAN. The output of the compiler is a collection of FORTRAN files, which can be compiled with any FORTRAN 95-compliant compiler. Thus, from the user's point of view, it is very simple to use MC2FOR.

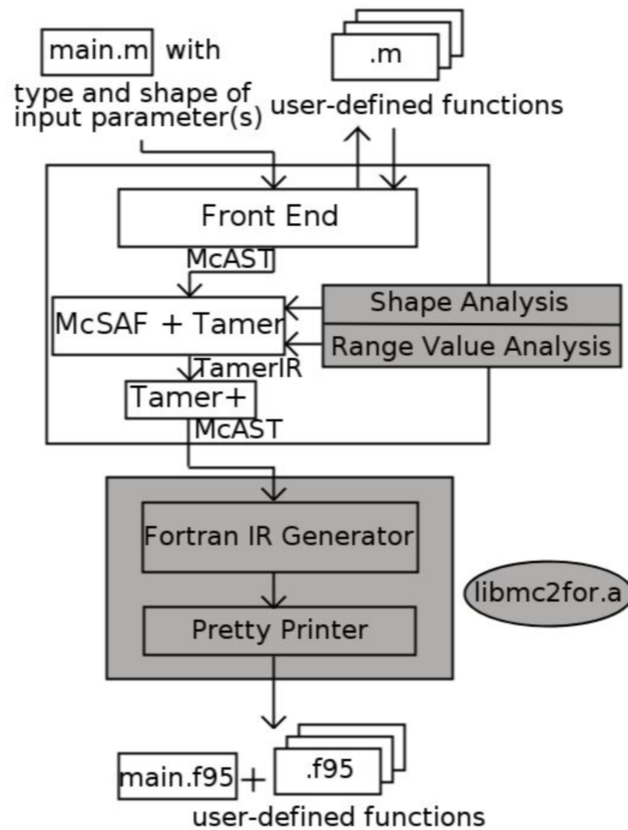

Figure 2.1 The overview of MC2FOR. We highlight the boxes which are the contributions of this thesis.

2.3 Components in MC2FOR

Now let us concentrate on the actual structural organization of MC2FOR. The central component driving the compilation process is the *Tamer* [Dub12, DH12b] module. It starts with the entry point function and iteratively discovers all the functions that are directly and indirectly called. For each processed MATLAB function file, the *McLab Front End* [McL] is used to scan and parse the file, generating a high-level intermediate representation (IR), McAST. The analysis and transformation engine, ***McSAF*** [Doh11, DH12a] is then used to transform to a lower-level AST; and to perform initial analyses such as *kind analysis* [DHR11], which determines which identifiers refer to arrays, and which refer to functions.[1] The Tamer then processes the IR into an even lower-level IR, TamerIR, which is more suitable for interprocedural static analysis.

For the purposes of the MC2FOR project, our main new analyses have been implemented in the Tamer's framework. The Tamer's framework, besides providing a low-level IR with well-defined semantic meanings, also provides an extensible interprocedural abstract value analysis framework. In the framework, Tamer already provided some basic MATLAB type characteristics analyses, like simple constant analysis and MATLAB class (mclass) analysis. In order to generate FORTRAN, MC2FOR provides two more important analysis components to the framework, which are the *shape analysis* and the *range value analysis*. The shape analysis computes shape information of all the variables for all program points in a given MATLAB program. The range value analysis extends the basic constant analysis and is used to estimate the range of a scalar variable at each program point. The range value analysis can assist the shape analysis in the case of static array bounds checking.

The TamerIR is in the form of three address code, which is very suitable for static analysis but introduces a lot of temporary variables making the code unreadable. In order to generate readable FORTRAN and other target languages code, there is a restructuring component, *Tamer+*, which aggregates the low-level three address code of TamerIR back to

[1] In MATLAB the syntactic construct `a(i)` can either be an array reference or a function call. In fact, even the reference to the identifier `i` can either be a reference to a variable `i`, or a call to the predefined function `i` which gives the complex value *i*.

the high-level IR of McAST. The obtained type characteristics and the new transformed McAST are then given as inputs to the FORTRAN code generation backend. By traversing the IR, the backend generates an functionally equivalent FORTRAN IR. In this traversing process, MC2FOR solves the problems of mapping built-in functions in MATLAB to FORTRAN, transforming difference between MATLAB and FORTRAN in array indexing and so on. There is also a standalone FORTRAN library, `libmc2for`, shipped together with MC2FOR, which is used to map those built-in functions which have no direct FORTRAN equivalents. Finally, after building the FORTRAN IR, MC2FOR pretty prints the IR into files with corresponding names. One of them maps to the entry point function file and the others map to the user-defined function file(s) used in the program. The resulting FORTRAN programs should be easy to redistribute, since they can be compiled with any FORTRAN 95-compliant compiler (including the open source GFortran). Further, as we show in Chapter 6, the resulting FORTRAN code is often more efficient than the original MATLAB code.

Chapter 3
Shape Analysis

As with other dynamically-typed programming languages, MATLAB programmers do not need to declare the type of variables before defining and then using them in the code. This feature gives the programmers a lot of convenience, allows them to focus on the design of the algorithm, and supports fast prototyping. In contrast, for statically-typed programming languages, like FORTRAN, the programmers must declare the type of variables before defining and then using them in the code. The advantage of variable declaration is that it simplifies the compilation process as the compilers know exactly what types of operators to generate, as well as knowing exactly how much storage to allocate for different variables. To pave the way for translating a given MATLAB program to a FORTRAN 95 program, the first step of MC2FOR is to statically estimate some properties or characteristics[1] of all the variables used in the MATLAB program. With enough knowledge of all the variables, MC2FOR can thus generate the variable declaration section in the converted FORTRAN program.

In this thesis, the estimated characteristics of variables include five components: constant information, mclass information, shape information, range value information and complex information. The constant and range value information are not used directly for type declaration in the converted FORTRAN code, but they are essential to other value analysis components. The mclass, shape, and complex information are all needed for declaring

[1] This is also called value analysis in our research.

variables in the converted FORTRAN code. For example, if variable `var` is a 2-by-3 complex array, then in the converted FORTRAN code, it will be declared as:

```
COMPLEX, DIMENSION(2,3) :: var
```

Intuitively, the shape of an array is defined as the total number of dimensions, also called **rank**, and the size of each dimension, also called **extent**. In MATLAB, even a scalar is represented as a 1-by-1 array. So, the shape analysis is an analysis to estimate the rank and extent of all the variables in a given MATLAB program.

In order to get the shape information of all the variables in a given program, we designed and implemented a shape information flow analysis component, which can be integrated with the Tamer's interprocedural value analysis framework. The component also includes a concise and extensible domain-specific language to write shape propagation equations for built-ins utilizing Tamer's built-in framework. Recall that the Tamer's built-in framework provides us with an extensible framework to handle abstract values propagating through MATLAB built-in functions, and the Tamer's interprocedural value analysis framework takes care of propagating abstract value analysis in a given MATLAB program and merging the different values at the fixed point. Thus, for the shape analysis, we should first design and implement a solution to estimate how do the shapes propagate through built-ins and integrate it into the Tamer's built-in framework, then develop a merging strategy for the shape analysis which is invoked by the Tamer's interprocedural value analysis framework.

In this chapter, we start with the solution to estimate shape information through MATLAB built-in functions in Section 3.1, and then we discuss the problem of how to merge different shape results for control flow statements in Section 3.2, finally, we introduce a solution to assist in assessing the correctness of shape analysis results in Section 3.3.

3.1 Propagating Shapes through MATLAB Built-in Functions

In the shape analysis, we implement a shape propagator object to collect shape information by using Tamer's interprocedural value analysis framework. When the shape propagator goes through a program to analyze the shape information of all its variables, one of

the most difficult problems it encounters is how to statically infer shape information of a built-in function's output argument(s) given the input argument(s). For example, when the propagator encounters the statement `arr = ones(k, l, m)`, if we want to translate this code to FORTRAN, we have to know some shape information of the variable `arr`. The problem is how to statically obtain the shape information of `arr`. Let's assume that `k`, `l` and `m` are all scalars and whose values are 2, 3, 4, respectively. With the knowledge of how the built-in function `ones` works, we know that the variable `arr` will be a 2-by-3-by-4 array, which means that `arr` will have three dimensions and the sizes of each dimension are 2, 3 and 4, respectively. Then, together with other value information of `arr`, like its mclass, we would know to declare the variable `arr` in the generated FORTRAN code. But, the problem is not that simple. What if the assignment expression is `arr = ones(v)` where `v` equals 4 or even `v` is a vector containing two 3s? Again, with the knowledge of how the built-in function `ones` works, we know that the variable `arr` will be a 4-by-4 array or a 3-by-3 array. But there are hundreds of built-in functions in MATLAB, and for each of them, there are multiple rules determining the shape information of output argument(s) based on the shape information or/and other value information, like constant information, of function's input argument(s). So, is there a general solution to solve this problem?

In this section, we propose a concise and extensible domain specific language, the *shape propagation equation language*, to write *shape propagation equations* for each MATLAB built-in functions to describe the behaviors of how the shapes propagate through the built-ins. Besides the shape propagation equation language, we also introduce a matching algorithm whose inputs are the encountered built-in function's input argument(s) and the corresponding shape equation for the function, and whose output is the shape information of the function's output argument(s). We made the shape propagation language as concise as possible, so that it's easy to be remembered and understood; and furthermore, we made it extensible, so that it can be used to implement shape equations for new built-in functions in the future.

We wrote shape equations for most frequently used built-ins and put them together with other existing value propagation equations[2], which can be regarded as a value propagation dictionary for MATLAB built-in functions. By these means, every time the shape propa-

[2]For example, mclass propagation equations and complex propagation equations.

gator encounters a built-in function in a given program, it will fetch the shape equation in the dictionary, and by applying matching algorithm on that equation, it will get the shape information of the output argument(s).

This section is divided into 5 subsections. Subsection 3.1.1 presents the typical behaviors of built-in functions on the shapes. Subsection 3.1.2 introduces the necessary features of the shape propagation equation language used to cover those behaviors in subsection 3.1.1. Subsection 3.1.3 provides a more formal description of the language. Subsection 3.1.4 presents the shape matching algorithm. Subsection 3.1.5 concludes the whole section.

3.1.1 Typical Behaviors of Built-in Functions on Shapes

Before formally introducing the shape propagation equation language and shape propagation equations for built-ins, we summarize the most typical behaviors of how a built-in function propagates shape information of its input and output argument(s). In other words, what shape information of the output arguments will be yielded based on the encountered built-in function's input argument(s). In order to make the language as concise as possible, we only equip the language with enough features to describe all these behaviors. By supporting all the typical behaviors, no matter how many new built-ins are introduced into MATLAB in the future, we still can describe how these new built-in works on shapes by writing shape equations in this language.

To simplify the shape representation in the shape analysis, we use a list of numbers enclosed by a pair of square brackets to represent shape information of an array, and each number in the vector represents the size of corresponding dimension of the array. For example, if `arr` is a 3-by-5 array, we will represent its shape as `[3,5]`[3], in which `3` means that the size of the first dimension of `arr` is 3 and `5` means that the size of the second dimension of `arr` is 5. By studying a lot of MATLAB built-in functions, we summarized the key possible behaviors into four major categories as below.

Based on the shape of input argument(s): The first typical behavior of built-in functions on shape is that the shape of output argument(s) only depends on the shape of input

[3]Please don't be confused with the bibliography reference marks

argument(s). For example, the built-in function `round` takes only one input argument and the shape of output argument is the same as the shape of input argument; moreover, the return shape of some arithmetic built-ins, like `+`, `-`, `.*` and `./`, also only depends on the shape of the input arguments.[4]

Based on numeric values of input argument(s): The shape of output argument(s) of some built-in functions depends on the numeric value of input argument(s) of the functions. For example, the output of built-in function call `true(3)` will be a 3-by-3 array and the output of `ones([1,2,3])` will be a 1-by-2-by-3 array.

Based on optional numbers or strings: The shape of output argument(s) of some built-in functions depends on some optional numbers or character strings in their input argument lists. For example, the return shape of the built-in function `svd`, which is used to compute singular value decomposition of an array, depends on an optional input number argument, `0`, and an optional input string argument, `'econ'`. For instance, if the shape of array `X` is `[3,2]`, after the assignment expression `[U,S,V] = svd(X)`, the shape of U, S and V will be `[3,3]`, `[3,2]` and `[2,2]`, respectively; while, after the assignment expression `[U,S,V] = svd(X,0)` or `[U,S,V] = svd(X,'econ')`, the shape of U, S and V will be `[3,2]`, `[2,2]` and `[2,2]`, respectively.

Sometimes, the optional input arguments may not affect the return shape, like the built-in function call `ones(2,2,'int8')` will return a 2-by-2 array full of 1s in the MATLAB numeric type of int8, while without explicitly putting the character string argument `'int8'`, the default return MATLAB numeric type of the array elements will be double, but the return shape will remain as 2-by-2. In this case, we still need some matching expressions to match this input string argument.

Other cases: The above three categories already cover most behaviors. However, there are still a few special cases in MATLAB. For example, the built-in function `cross`, which is used to get cross product of two vectors or matrices. Besides this function requires that both two input vector or array arguments must have the same shape, it

[4] .* is element-wise multiplication, and ./ is element-wise division

also requires that the vectors must be 3 element vectors or the matrices must have at least one dimension with the size of 3.

3.1.2 Features of the Language

In order to make the shape propagation equation language as concise as possible, we designed this language equipped with small number of necessary features. These features are capable enough to describe all the key possible behaviors of how a MATLAB built-in function propagates the shape information through its input argument(s) to its output argument(s). A more formal introduction of the language and the equations will be given in Subsection 3.1.3.

Shape matching expressions: To handle the behavior based on the shape of input argument(s), there are some shape matching expressions in this language. The shape matching expressions are the basic and essential elements in this language. In detail, we use the `$` symbol, upper-case letters, like `M`, and vector expressions, like `[m,n]` to match the shape of input argument(s). The `$` symbol is used to match an input argument of 1-by-1 shape (or called scalar), upper-case letters are used to match other arrays which are not of 1-by-1 shape, and vector expressions are more specific than the upper-case letters, which has restriction on certain dimensions. For example, for the built-in function `mtimes`, which is the matrix product of two matrices where the number of columns of the first matrix must equal the number of rows of the second matrix, by using shape matching expressions to describe this behavior, it will be:

```
[m,k],[k,n] -> [m,n]
```

This can be considered as an equation to describe how shape information propagates through the function. In this thesis, we call it a *shape propagation equation*. By interpreting this equation, we can get several shape propagation properties of this function, they are:

1. There should be only two input arguments and one output argument.
2. The two input arguments should have two dimensions.

3. If the shape of the first input argument is m-by-k, and the shape of the second input argument is k-by-n, then the shape of the output argument will be m-by-n;
4. Moreover, in this example, the vector expressions also implies a restriction on the shape of inputs. Since, the second dimension of the first input argument and the first dimension of the second input argument are represented in the same lower-case letter, `k`, it requires that the number of columns of the first matrix must equal the number of rows of the second matrix. If they are not equal, in MATLAB, there is going to be a run-time error[5] in execution, and in the shape analysis, the return shape information of this function will be marked as `[shape propagation fails]` which is used to inform the users of the shape analysis that there is a misuse of function `mtimes` in its input MATLAB code, and the shape analysis will carry on to the end of the program. If the matching process succeeds, the first dimension of the output matrix will be the same size as the first dimension of the first input argument, and the second dimension will be the same size as the second dimension of the second input argument.

Assignment expression and function call expressions: To handle the behavior based on the numeric values of input argument(s), there is some functionality to capture the numeric values of the input argument(s). For example, for the built-in function `ones`, which is always used as an array preallocation function and will generate an array full of 1s. The shape propagation equation of this function is:

```
[] -> $ ||
($,n=previousScalar(),add(n))+ -> M
```

In this equation, we use function `previousScalar` to ask for the value of previous matched scalar argument and use function `add` to add `n` into the default return shape array. In this language, we define `n=previousScalar()` as assignment expression and define `previousScalar()` and `add(n)` as function call expressions. By interpreting this equation, it means that there are two cases, separated by a `||` symbol, for this matching process: the first case is that if the input argument is empty, the

[5] In MATLAB R2013a, the error information is inner matrix dimensions must agree.

return shape will be the same as a scalar, which is 1-by-1; the second case is that if the input argument is a list of scalars, and for each scalar, we use the assignment expression `n=previousScalar()` to get the integer value of the previous matched scalar, then we use the function call `add(n)` to add the integer value into a default return shape array, the `+` symbol has the same meaning as in regular expressions, since the first case already covers the empty input argument situation, here we use the `+` symbol to represent the fact that there is at least one scalar to match. After matching all the input arguments, the output shape will be the default return shape array.[6]

String literals: To handle the behavior based on optional strings[7], we also include character string literals as a kind of shape matching expression. Then we can use it to match those optional input string literals arguments. For example, for the built-in function `ones`, now we can extend the shape equation to support the function taking string literals as its input argument, like `ones(3,3,'int8')`. The shape equation to cover this function is:

```
[]|'int8' -> $ ||
($,n=previousScalar(),add())+,'int8'? -> M
```

In this equation, we introduced several new symbols and expressions. In the first case, the `|` symbol is used to separate two shape matching expressions, `[]` and `'int8'`. It means that the input argument can be either empty or a string of `'int8'` and the return shape will be 1-by-1. In the second case, the `?` symbol has the similar meaning as in regular expressions. Since `'int8'?` appears at the last position of the left hand side part of the `->` symbol, it means that at the end of the input argument list, there can be a string literal argument or not.

Assert expressions: To handle the remaining special cases, the language consists of some assert expressions, which is used to assess some conditions. The assert expression will be evaluated, and if its return value is true, the matching process continues, or the

[6]If any upper-case letter, like `M` in this example, does not appear in the shape matching side, it will be used to represent the default return shape array.

[7]Optional numbers in the input argument list will be handled by using the $ symbol.

matching process on current case terminates. Recall the MATLAB built-in function `cross` which requires that two input vectors or matrices must be 3 element vectors or have at least one dimension with the size of 3, besides the requirement that two inputs also must be in the same shape. To describe these requirements, the corresponding shape equation is:

```
M,M,atLeastOneDimEqls(3)->M ||
M,M,$,n=previousScalar(),
k=previousShapeDim(n),isEqual(k,3)->M
```

The assert expression `atLeastOneDimEqls(3)` in the first case is used to check whether there is at least one dimension with the size of `3`. The second case in above equation is used to cover the case where the built-in function call with one extra optional input scalar, like `cross(A,B,DIM)` in which `A` and `B` are two matrices and `DIM` is a scalar to indicate that along which dimension to apply cross product. So based on the equation, we use `$` to match that optional scalar input, and use `n=previousScalar()` to store the scalar's numeric value into `n`, and then we use another assignment expression, `k=previousShapeDim(n)`, to get the corresponding dimension's size, finally, we use another assert expression `isEqual(k,3)` to check whether the corresponding dimension's size equals 3. Since MATLAB is evolving all the time, there may always come out some new built-ins. To handle this challenge, we made the language extensible by allowing programmers add new assert expression functions inside function call CST node.

Other advanced features: There are also some other function calls can be used in the assignment expression. For example, for the built-in function `diag`, which is used to return the main diagonals of the matrices, the shape equation for it is:

```
[m,n],k=minimum(m,n) -> [k,1] ||
[n,1]|[1,n],$,k=previousScalar(),n=add(k) -> [n,n]
```

In this equation example, the equation uses `minimum` to find the minimum value between `m` and `n`.

Moreover, the assignment expression can also be used as an assignment to an indexed upper-case letter, which is used to change the previous matched shape expression information. For example, for the built-in function `median`, which is used to get the median value of a vector or an array, the shape equation for this function is:

```
[1,n]|[n,1] -> $ ||
M,M(1)=1 -> M ||
M,M -> M ||
M,$,n=previousScalar(),M(n)=1 -> M
```

This equation consists of four cases. In the first case, it tries to match a row vector or a column vector, and returns a shape of 1-by-1 as the result. In the second case, we introduced the array indexing assignment expression, `M(1)=1`. This case first tries to match an array using `M` and the match assigns the shape information into `M`, then `M(1)=1` overwrites the first dimension's size of the shape stored in `M` to 1, and finally returns the modified shape `M` as the result. In the third case, it tries to match two same shape matrices and then return the shape of them as the result. For the last case, it tries to match an array and a scalar, then to get the value of that scalar and to store the value into `n`, and the array indexing assignment expression, `M(n)=1`, will set the `n`th dimension's size of the shape stored in `M` to `1`, and finally return the shape stored in `M` as the result.

In summary, the shape propagation equation language supports shape matching expressions, function call expressions, assignment expressions and assert expressions. In the following subsections, we give a detailed introduction of the language and how the matching algorithm works in the shape equation.

3.1.3 Shape Propagation Equation Language

The shape propagation equation language is the language to write shape propagation equations which are used to describe how MATLAB built-in functions behave on shape information of input and output arguments. The tokens and the formal grammar of the language

are listed in Appendix A.1 and A.2. In this subsection, we introduce the general structure and semantics of the constructs in this language, starting with the top-level constructs.

Caselist: Since almost all the MATLAB built-in functions can take several combinations of input arguments, a shape equation of a built-in function is represented as a caselist of at least one case, and the cases are separated by OROR (`||`) symbols. For example,

```
case1 || case2 || case3
```

The formal grammar snippet code for this construct is:

```
caselist
    = case.c
    | case.c OROR caselist.l
    ;
```

The separate cases are evaluated from left to right. If any of them is matched successfully with the shape of input argument(s), the matching process terminates and returns the corresponding shape result.

Case: Each case in the caselist can be divided into two parts, a pattern list side and a shape output list side, separated by an ARROW (->) symbol. All the pattern list expressions will be at the left hand side of the ARROW symbol, and all the shape output list expressions will be at the right hand side of the ARROW symbol. For example,

```
pattern list side -> shape output list side
```

The formal grammar snippet code for this construct is:

```
case
    = patternlist.p ARROW outputlist.o
    ;
```

The pattern list side is evaluated prior to the shape output list side.

Pattern list side: The pattern list side is composed of a list of pattern expressions which are separated by COMMA (,) symbols, and all the expressions are evaluated from left to right. For example,[8]

```
PExp_1 , PExp_2 , ... PExp_n -> shape output list side
```

The formal grammar snippet code for this construct is:

```
patternlist
    = pattern.e
    | pattern.e COMMA patternlist.p
    ;
```

If any expression on the pattern list side fails in the matching process, the matching process for the enclosing case will be terminated and if there are still remaining case(s) in the caselist, the matching process will start from that next case, repeating the matching process again until one case is matched successfully or there isn't any case left in the caselist. If none of the cases in the caselist matches the input argument(s) successfully, it means that there must be some misuse of the built-in function by the MATLAB programmer. Our compiler will issue a warning to the user of the shape analysis.

Pattern expressions: Pattern expressions can be categorized into three different kinds of expressions: shape matching expression, assignment expression and assert expression. The formal grammar snippet code for this construct is:

```
pattern
    = matchExpr.m
    | assignExpr.a
    | assertExpr.a
    ;
```

[8]Inside the example, `PExp` is short for pattern expression.

Among these three expressions, only the shape matching expression is used to match the shape of the input argument(s) and if the matching is successful, the input argument is consumed, which means the matching process will point to the next input argument if there are any left, or go to the shape output list side. The other two expressions, assignment expression and assert expression, are helper expressions in the shape propagation.

Shape matching expression: There are four kinds of symbols which are used to represent shape matching expression: the DOLLAR (`$`) symbol, upper-case letters, vertcat expressions and the ANY (`#`) symbol. Since we use vectors of numbers to represent shapes for arrays, in the grammar, we name shape matching expression as vector expression, so the formal grammar snippet code for shape matching expression is:

```
vectorExpr
    = SCALAR.d
    | UPPERCASE.u
    | ANY.a
    | vertcatExpr.v
    ;
vertcatExpr
    = LSPAREN RSPAREN
    | LSPAREN arglist.al RSPAREN
    ;
arglist
    = arg.a
    | arg.a COMMA arglist.al
    ;
arg
    = scalarExpr.s
    | vectorExpr.v
    ;
```

```
scalarExpr
    = NUMBER.n
    | LOWERCASE.l
    ;
```

Here is the description for all those four kinds of symbols.

1. The `$` symbol: This symbol is used to match input arguments of 1-by-1 shape (also called scalars in our research);
2. Upper-case letters: These symbols are used to match input arguments of arrays which are not of 1-by-1 shape. Since it's almost impossible to need more than 26 different upper-case letters in one shape equation, to make the language concise, it only allows using one letter to match an array's shape, not combination of letters;
3. Vertcat expressions: Vertcat expressions are defined as a list of lower-case letters or numbers enclosed by a pair of square brackets, like `[1,k]` or `[m,2,n]`. Vertcat expressions are also used to match input arguments of arrays, while it may impose more restrictions on the size of certain dimensions;
4. The `#` symbol: In some cases, we may not care about the shape of current input argument. For example, the built-in function `vertcat` is the vertical concatenation of arrays with any dimension. The concatenated arrays must have the same number of columns[9], but don't need to have the same number of rows. The shape equation for this built-in is:

```
$,n=previousShapeDim(1),N=copy($),N(1)=0,
(#,k=previousShapeDim(1),K=copy(#),K(1)=0,
isEqual(K,N),n=add(k))*,N(1)=n -> N
||
M,n=previousShapeDim(1),N=copy(M),N(1)=0,
(#,k=previousShapeDim(1),K=copy(#),K(1)=0,
isEqual(K,N),n=add(k))*,N(1)=n -> N
```

[9] Actually, in MATLAB, if the input arrays of `vertcat` have more than two dimensions, all the dimensions except for the first dimension must be the same size.

This equation probably is the most complicated shape propagation equation which we have ever written. This equation has two cases: the first case is used to match the situation where the first input argument is a scalar; and the second case is used to match the situation where the first input argument is a non-scalar array. The reason why we have two cases for this equation simply is that we want to separate matching a scalar from matching other arrays. So we can just go through the case for matching non-scalar arrays, and the scalar case is just a special case of the non-scalar arrays one. In the non-scalar arrays case, it will start by matching an array, then it fetches the first dimension's size of this array and stores the value into `n`, then it uses `N` to copy the shape in `M` and sets the first dimension of the shape in `N` to 0.[10] The expressions inside the `()*` symbol are used to match any number of input array arguments. For each array, the equation will save the size of its first dimension into `k`, copy the shape of this array to `K`, set the first dimension of the shape in `K` to 0, then by using the function call `isEqual(K,N)`, compare all the dimensions except the first dimension[11] of the shape stored in `K` with the ones stored in `N`. If the result of `isEqual(K,N)` is true, the shape equation will add the sizes of their first dimensions together. The matching process will repeat the expression matching inside `()*` until there is not any input argument left, then the return array will have the same shape as all the input arrays except the first dimension which is the summation of the sizes of all the input arrays' first dimension.

Function call expression: The formal grammar snippet code for this construct is:

```
fnCall
    = ID.i LRPAREN RRPAREN
    | ID.i LRPAREN arglist.al RRPAREN
    ;
```

[10]The reason we copy the shape before we apply computation is for the safeness of the original shape information.

[11]We achieved this goal by setting the first dimensions of them to 0.

Besides the function calls we have already introduced, like `previousScalar`, `add` and `minimum`, there are still several more built-in functions in this language. The complete list of currently implemented built-in functions is given in Table 3.1.

Table 3.1 Built-in functions of shape propagation equation language

Function	Functionality of the function
`previousScalar()`	get the value of previous matched scalar
`previousShapeDim(arg)`	get the size of previous matched shape's `arg`th dimension
`add(arg)`	add `arg` to the default return shape array
`minus(arg1,arg2)`	compute `arg1 - arg2`
`div(arg1,arg2)`	compute `arg1 / arg2`
`minimum(arg1,arg2)`	get the minimum between `arg1` and `arg2`
`copy(arg)`	copy the shape of `arg` to the temporary variable
`numOutput(arg)`	checking whether the number of output arguments equals `arg`
`isEqual(arg1,arg2)`	comparing the value of `arg1` with the one of `arg2`
`atLeastOneDimEqls(arg)`	checking whether there is at least one dimension's size of matched array equals `arg`

Since MathWorks may introduce new MATLAB built-in functions with some new restrictions in the future, we made the shape propagation equation language extensible by allowing users to add new function calls into the language. The user can add new functions inside the function call node in this language, and then implement the new restrictions inside the new functions.

One more thing needed to be clarified is the rules for how all the lower-case and upper-case letters work in the equations. On the pattern list side, for the shape matching expressions, the lower-case or upper-case letters are used to match the shapes of input arguments, and if a letter has already appeared in the equation, when the letter comes again, the matching process requires them to have the same value, if not, the matching fails; for the other expressions, the lower-case or upper-case letter are used to store some values for the purpose of propagating

shapes through. On the output list side, the lower-case or upper-case letters are used to return the shape information stored in them.

Assignment expression: Assignment expression is represented as:

```
lvalue = rvalue
```

Lvalue can be (1) lower-case letters, (2) upper-case letters, (3) # symbol, and (4) indexed upper-case letters or # symbol. The rvalue can be numbers, lower-case letters, other shape matching expressions and function call expressions. The formal grammar snippet code for assignment expression is:

```
assignExpr
    = assignmentLHS.l EQUAL assignmentRHS.r
    ;
assignmentLHS
    = LOWERCASE.l
    | UPPERCASE.u
    | UPPERCASE.u LRPAREN scalarExpr.s RRPAREN
    | ANY.a LRPAREN scalarExpr.s RRPAREN
    ;
assignmentRHS
    = scalarExpr.s
    | vectorExpr.v
    | fnCall.f
    ;
```

Assert expression: Assert expressions are represented by function call expressions only. The formal grammar snippet code for assert expression is:

```
assertExpr
    = fnCall.f
    ;
```

Assert expressions can be put at any place on the pattern list side, and will be evaluated to determine whether the current matching process should con-

tinue. For example, the assert expression `atLeastOneDimEqls(arg)` in Table 3.1 checks whether there is at least one dimension's size of matched array equals `arg`, if not, the current matching process will terminate and start from next case again if there is any case left.

Shape output list side: The shape output list side is very similar to the pattern list side, because both of them are composed of a list of expressions. For example,[12]

```
pattern list side -> OExp_1, OExp_2, ... OExp_n
```

While, in the shape output list side, there is only one kind of expression: the shape matching expression. Although we know that for some built-in functions, the number of output arguments affects the return shape of the functions, we put those assert expressions in the pattern list side, like `numOutput`. So, the formal grammar snippet code for this construct is:

```
outputlist
    = vectorExpr.v
    | vectorExpr.v COMMA outputlist.o
    ;
```

Moreover, the shape matching expressions may have the same representation in the shape output list side and pattern list side, but they work differently in two sides. In the shape output list side, the shape matching expressions are used to return the shape result, not to match the shape of argument(s).

Operators: There are some operators can be used on some pattern expressions or even pattern list introduced above, and most of these operators have the similar meaning as in regular expressions.

The () operator: The parentheses operator will produce a compound expression which is composed of at least one shape matching expression at the first place. It is mostly used to enclose at least one shape matching expression followed by

[12] Inside the example, `OExp` is short for shape output list expression.

assignment and assert expressions, and to work together with other operators to match one or more optional input arguments;

The ? operator: Putting a question mark operator after a shape matching expression or compound expression in the pattern list side means that during the matching process, the preceding expression is optional, and if there is no input argument for this expression to match, it won't be an error;

The + operator: Putting a plus operator after a shape matching expression or compound expression in the pattern list side means that during the matching process, the preceding expression will be evaluated at least one or more times which depends on the number of input argument(s);

The * operator: Putting a star operator after shape matching expression or compound expression in the pattern list side means that during the matching process, the preceding expression may be evaluated one or more times which depends on the number of input argument(s);

The | operator: The choice operator let the shape of input argument(s) matches either the expression before or the expression after the operator;

The '' pair: The single quotation pair encloses some string literals and is only used to match an input string literal argument.

With these operators, the formal grammar snippet code for all the pattern matching expressions is:

```
matchExpr
    = basicMatchExpr.m OR basicMatchExpr.n
    | basicMatchExpr.m QUESTION
    | basicMatchExpr.m MULT
    | basicMatchExpr.m PLUS
    | basicMatchExpr.m
    ;
```

```
basicMatchExpr
    = LRPAREN patternlist.p RRPAREN
    | SQUOTATION ID.i SQUOTATION
    | SQUOTATION LOWERCASE.i SQUOTATION
    | vectorExpr.v
    ;
```

A list of some shape equation examples is attached in Appendix A.3.

3.1.4 Shape Matching Algorithm

After we have the shape propagation equation language and understand the semantics of its constructs, it's time to clearly explain how can we infer shapes through encountered built-in functions by using shape propagation equations.

Recall that when an abstract value propagator encounters a built-in function during its analysis in a given MATLAB program, it will apply the built-in framework. The built-in framework will invoke the matching algorithms in each component[13] in the abstract value analysis. For the shape analysis component, the built-in framework requires us to provide a corresponding matching algorithm to it, which is used to infer the return shape of the built-in functions. The algorithm is achieved by several functions working together, which are in Listing 3.1 to 3.5.

Listing 3.1 shows the shape matching function called by the built-in framework, which returns the final shape result by taking in the encountered built-in's name and its input argument list. Inside the `matchShape` function, the shape analysis iterates over all the cases in the encountered built-in's shape equation to find the first matched case. For each case, the shape analysis invokes the shape matching function of the case construct in Listing 3.2. Inside the shape matching function of the case construct, the shape analysis invokes the shape matching function of the pattern list construct in Listing 3.3, if the pattern list side matches the input argument list successfully, the shape analysis will invoke the shape matching function of the output list construct in Listing 3.5, and finally, return the result to the built-in

[13]Current components include mclass analysis component, complex analysis component, shape analysis component and range value analysis component.

framework. Listing 3.4 shows the shape matching function for all the possible expressions in the pattern list construct.

```
# the matching algorithm entry function
# called by the built-in framework.
function matchShape(builtin, inputArgs)
  get shape equation for the builtin
  caselist = parse the shape equation
  for each case in caselist
    shape_result = case.match(case, inputArgs)
    if shape_result != not_matched_shape
      return shape_result
    end if
  end loop
  # if all the cases do not match inputArgs
  return not_matched_shape
end function
```

Listing 3.1 Shape matching algorithm 1 of 5: function matchShape

```
# match function of case, returns a shape result
function case.match(case, inputArgs)
  get pattern_list of case
  get output_list of case
  matched = pattern_list.match(pattern_list, inputArgs)
  if matched
    shape_result = output_list.match(output_list)
    return shape_result
  else
    # if pattern list side fails in matching inputArgs
    return not_matched_shape
  end if
end function
```

Listing 3.2 Shape matching algorithm 2 of 5: function case.match

```
# match function of pattern list, returns a boolean
# value indicating whether the matching succeeds.
function pattern_list.match(pattern_list, inputArgs)
  # MatchingPosition is regarded as a global variable
  # shared by all these matching functions, to
  # indicate which input argument is being matched.
  MatchingPosition = 0
  for each expression in pattern_list
    matched = expression.match(expression, inputArgs)
    if not matched
      return false
    end if
  end loop
  # the return depends on whether all the expressions
  # succeed in matching all the input arguments.
  return MatchingPosition == inputArgs.length
end function
```

Listing 3.3 Shape matching algorithm 3 of 5: function patternlist.match

```
# match function of expression, return a boolean
# value indicating whether the matching succeeds
function expression.match(expression, inputArgs)
  if the expression is a shape matching expression
    try to match this expression with inputArgs[MatchingPosition]
    if matched
      increment MatchingPosition
      return true
    else
      return false
    end if
  else if it's an assignment expression
    evaluate this expression
    return true
  else if it's an assert expression
    passed = evaluate this expression
    return passed
```

```
# for the case of (exp,...,exp)
else if it's a PARENTHESES compound expression
  for each expression enclosed by the parentheses
    matched = expression.match(expression, inputArgs)
    if not matched
      return false
    end if
  end loop
  return true
# for the case of exp?
else if it's a QUESTION MARK compound expression
  if MatchingPosition < inputArgs.length
    matched = expression.match(expression, inputArgs)
    return matched
  else
    return true
  end if
# for the case of exp+
else if it's a PLUS compound expression
  if MatchingPosition <= inputArgs.length - 1
    repeat
      matched = expression.match(expression, inputArgs)
    until MatchingPosition == inputArgs.length or not matched
    return matched
  else
    return false;
  end if
# for the case of exp*
else if it's a STAR compound expression
  if MatchingPosition < inputArgs.length
    repeat
      mtached = expression.match(expression, inputArgs)
    until MatchingPosition == inputArgs.length or not matched
    return matched
  else
    return true
  end if
```

```
  # for the case of exp|exp
  else if it's an OR compound expression
    matched = first_expression.match(first_expression, inputArgs)
    if not matched
      matched = second_expression.match(second_expression, inputArgs)
      return matched
    else
      return true
    end if
  end if
end function
```

Listing 3.4 Shape matching algorithm 4 of 5: function expression.match

```
# match function of output list, returns a shape result
function output_list.match(output_list)
  shape_result = new an empty shape information list
  for each shape matching expression in output_list
    get shape based on the expression
    add shape to shape_result
  end loop
  return shape_result
end function
```

Listing 3.5 Shape matching algorithm 5 of 5: function outputlist.match

The implementation details of applying this matching algorithm by using the shape propagation equation is given in the Appendix A.4.

3.1.5 Summary

In this section, we introduced a concise and extensible domain specific language, shape propagation equation language, and shape propagation equations written in the language to solve the problem of how to estimate shape information propagating through MATLAB built-in functions. For each built-in function, we defined a shape equation in this language. With the shape equations and a corresponding matching algorithm, every time the shape analysis encounters a built-in function, the analysis can statically infer the function's return

shapes of the output argument(s) by looking up its shape equation and executing the shape matching algorithm.

3.2 Merging Different Shapes

After we solve the problem of how shape information propagates through MATLAB built-in functions, another big problem in the shape analysis is how to merge different shapes for the same variable. Different shapes for the same variable come from the situations where a variable is assigned different shapes on different branches in an if-else statement, or a variable is assigned different shapes in different iterations in a loop statement.

Recall that our abstract value analysis framework guarantees that the shape analysis can go through all the branches or iterations to collect shape information for all the variables, but when a variable has different shapes on different branches in an if-else statement or from different iterations in a loop statement, we need to consider how to merge the different shapes of the variable. The framework requires us to provide a merging strategy and a corresponding `equals` function to check whether the analysis get to the fixed point in a loop statement for the shape analysis component. For example, in Figure 3.1 (a), when the shape propagator analyzes the if-branch, it will infer the shape of variable `arr1` as `[2,2]`; and when it analyzes the else-branch, it will infer the shape of `arr2` as `[2,3]`. The problem is what is the shape of `arr1` at the end of this if-else statement. Moreover, in Figure 3.1 (b), what if the numbers of the shapes' dimensions are different?

In this section, we first introduce a merging strategy and an `equals` function for our shape analysis to handle both the situations of merging shapes with the same or different ranks in Subsection 3.2.1, then we list some wild MATLAB cases of merging different shapes in loop statements and explain the final version of the merging strategy and the `equals` function in Subsection 3.2.2.

3.2.1 Merging Strategy

The brief idea of the shape merging strategy is that, first, make sure both shapes have the same number of dimensions by adding 1(s) to the end of the dimension list of the shape

```
1 function foo1(n)
2 if (n>10)
3   arr1 = ones(2,2); % the shape of arr1 will be [2,2].
4 else
5   arr1 = ones(2,3); % the shape of arr1 will be [2,3].
6 end
7 arr2 = arr1; % what is the shape of arr1 and arr2?
8 end
```

(a) Merging two shapes with the same number of dimensions

```
1 function foo2(n)
2 if (n>10)
3   arr1 = ones(2,2); % the shape of arr1 will be [2,2].
4 else
5   arr1 = ones(2,3,2); % the shape of arr1 will be
      [2,3,2].
6 end
7 arr2 = arr1; % what is the shape of arr1 and arr2?
8 end
```

(b) Merging two shapes with the different number of dimensions

Figure 3.1 MATLAB code examples of merging two different shapes

with smaller number of dimensions, and then for each dimension, if the size values are identical in both shapes, use this value as the size of corresponding dimension in the return shape, or mark the size of corresponding dimension as unknown in the return shape. The pseudocode for the merging strategy and the `equals` function are given in Listing 3.6 and 3.7, respectively.

```
1  function merge(shape_a, shape_b)
2    if either shape_a or shape_b is not_matched_shape
3      return not_matched_shape
4    else if either shape_a or shape_b is unmergeable_shape
5      return unmergeable_shape
6    else
7      dim_a = get dimension list from shape_a
8      dim_b = get dimension list from shape_b
9      if dim_a.length > dim_b.length
10       INDEX = dim_b.length
11       repeat
12         # adding trailing 1s to the end of dim_b
```

```
13        add 1 to the end of dim_b
14        increment INDEX
15      until INDEX == dim_a.length
16    else if dim_a.length < dim_b.length
17      INDEX = dim_a.length
18      repeat
19        # adding trailing 1s to the end of dim_a
20        add 1 to the end of dim_a
21      until INDEX == dim_b.length
22    end if
23    INDEX = 1 # initializing to point at the first element in the list
24    dim_new = new an empty dimension list
25    repeat
26      if dim_a[INDEX] == dim_b[INDEX]
27        add dim_a[INDEX] to the end of dim_new
28      else
29        add unknown to the end of dim_new
30      end if
31      increment INDEX
32    until INDEX == dim_a.length
33    shape_result = new shape from dim_new
34    return shape_result
35  end if
36 end function
```

Listing 3.6 Shape merging strategy

```
1  function equals(shape_a, shape_b)
2   if both shape_a and shape_b are not_matched_shape
3     return true
4   else if both shape_a and shape_b are unmergeable_shape
5     return true
6   else
7     dim_a = get dimension list from shape_a
8     dim_b = get dimension list from shape_b
9     if dim_a.length != dim_b.length
10      return false
11    else
12      INDEX = 1 # initializing to point at the first element in the list
```

```
13      repeat
14        # comparing each dimension between two shapes
15        if dim_a[INDEX] != dim_b[INDEX]
16          return false
17        end if
18        increment INDEX
19      until INDEX == dim_a.length
20      return true
21   end if
22 end function
```

Listing 3.7 The equals function to check whether the analysis in loops gets to the fixed point

According to the merging strategy and corresponding `equals` function, the code examples in Figure 3.1 will be analyzed as in Figure 3.2.

```
1 function foo1(n)
2 if (n>10)
3   arr1 = ones(2,2); % the shape of arr1 will be [2,2].
4 else
5   arr1 = ones(2,3); % the shape of arr1 will be [2,3].
6 end
7 arr2 = arr1;
8 % the shape of arr1 and arr2 will both be [2,?].
9 end
```

(a) Merging two shapes with the same number of dimensions

```
1 function foo2(n)
2 if (n>10)
3   arr1 = ones(2,2); % the shape of arr1 will be [2,2].
4 else
5   arr1 = ones(2,3,2); % the shape of arr1 will be
       [2,3,2].
6 end
7 arr2 = arr1;
8 % the shape of arr1 and arr2 will both be [2,?,?].
9 end
```

(b) Merging two shapes with the different number of dimensions

Figure 3.2 The analysis result of MATLAB code examples in Figure 3.1

3.2.2 Merging Shapes in Loop Statements

In the pseudocode of the shape merging strategy and the `equals` function in Listing 3.6 and 3.7, we didn't explain what is the shape of `unmergeable_shape` at lines 4 and 5 in both listings. It is used to represent those shapes that cannot be merged during the shape analysis in MATLAB loop statements. First, let's have a look at some wild features of MATLAB in Figure 3.3.

```
1  function wild1(n)
2  arr1 = [];
3  for i = 1 : n
4    arr1 = [arr1 i];
5    arr2 = ones(arr1);
6    % the arr2 is growing from a scalar
7    % to a matirx of n dimensions
8  end
9  end
```

(a) Merging shapes with variant dimensions example 1

```
1  function wild2(n)
2  for i = 1 : n
3    arr1 = 1 : i;
4    arr2 = ones(arr1);
5    % the arr2 is growing from a scalar
6    % to a matirx of n dimensions
7  end
8  end
```

(b) Merging shapes with variant dimensions example 2

Figure 3.3 MATLAB code example of merging shapes with variant dimensions

In both code examples in Figure 3.3, the array `arr2` grows from a scalar to an array of n dimensions. Recall that the bottom line to generate FORTRAN declaration code for a MATLAB array variable is at least the rank of the array referred by the variable is constant at compile-time. Without knowing the rank of the array referred by the variable, we cannot generate declaration code for the variable. In these two code examples, the dimension of `arr2` depends on the input argument `n`, there is no way to generate FORTRAN declaration code for the variable `arr2`.

Although we cannot generate FORTRAN code for those array variables, we should still handle this kind of case when we encounter it during the shape analysis in a given MATLAB program. The problem is that since the array keeps growing, there will be no fixed point based on our current shape merging strategy and `equals` function in Listing 3.6 and 3.7, the analysis will eventually end up in an infinite loop. In order to hit the fixed point in the shape analysis for arrays which keep growing in a loop statement, we set up an *upper bound counter number* to the iterations which the shape analysis can take at most on merging different shapes for a variable in the loop statement. If the iteration times hits the upper bound counter number, the shape analysis will push the shape of that variable to the `unmergeable_shape`.

Let's go back to the shape merging strategy and `equals` function in Listing 3.6 and 3.7, at lines 4 and 5 in both listings, when the iteration times which the shape analysis takes on merging different shapes for a variable hit the upper bound counter number, the shape analysis will push the shape of this variable to `unmergeable_shape`, then in the following iteration, the `equals` function will inform the shape analysis that the analysis get to the fixed point for the loop statement.

The final shape merging relation is illustrated in Table 3.2. The $\bowtie$ symbol represents the merging operation. The `not_matched` represents the shape not succeeding in matching during the shape analysis. The reason can be either the programming typo in the MATLAB code or the implementation bug in the shape analysis. The `unmergeable` represents the shape of variables may having dependency on themselves in loops and cannot being merged by the shape analysis. Note that a variable with a shape of `unmergeable` is a legal usage in MATLAB. The `ordinary` represents the shape with a concrete dimension list. A shape has a concrete dimension list means that there may be some sizes of some dimensions are unknown, but at least the rank of the shape is known.

Table 3.2 Shape merging relation table

$\bowtie$	not_matched	unmergeable	ordinary
not_matched	not_matched	not_matched	not_matched
unmergeable	not_matched	unmergeable	unmergeable
ordinary	not_matched	unmergeable	ordinary

3.3 Shape Analysis Result Verification

Finally, how can we know the result from the shape analysis is correct? It's definitely not a valid nor an efficient way to examine the correctness of the results by manually checking them line by line. To solve this problem, we use the AspectMatlab compiler [TAH10], a dynamic MATLAB profiling toolkit, to weave some aspects into a given MATLAB program, and then print out the shape information of all the variables (or at least variables we care about) after the execution of the weaved program. Although we can only get shape information of the variables on the execution path, it is still far better than verifying the shape analysis results by manually checking them line by line. We can adjust the value of conditional expressions of those control statements in the given program to make it execute as many branches as possible. Here, in Figure 3.4, is the results comparison between profiling one of our benchmarks[14] by using the AspectMatlab compiler and the shape analysis. The corresponding aspect file is list in Appendix A.5. The `-` symbols in the column of `shape change` in the table (a) means that the shape of the variable never changes during the execution of the program. If the shape changes during the execution of the program, the `shape change` column in table (a) will record its second last time shape information. Based on the results in the figure, we can see that the shape analysis successfully captured the growth of the array `SRmat` statically and mark the shape of `SRmat` as `[?,6]`.

[14]The benchmark `adapt`.

variable name	shape change	final shape
scale	-	[1 1]
a	-	[1 1]
b	-	[1 1]
sz_guess	-	[1 1]
tol	-	[1 1]
i	-	[1 1]
SRmat	[3270 6]	[3271 6]
iterating	-	[1 1]
done	-	[1 1]
h	-	[1 1]
c	-	[1 1]
Fa	-	[1 1]
Fc	-	[1 1]
Fb	-	[1 1]
S	-	[1 1]
SRvec	-	[1 6]
m	-	[1 1]
state	-	[1 1]
n	-	[1 1]
l	-	[1 1]
p	-	[1 1]
SR0vec	-	[1 6]
err	-	[1 1]
SR1vec	-	[1 6]
SR2vec	-	[1 6]
tol2	-	[1 1]
a0	-	[1 1]
b0	-	[1 1]
tol0	-	[1 1]
c0	-	[1 1]
quad	-	[1 1]

(a) The result of AspectMatlab

variable name	shape info
scale	[1 1]
a	[1 1]
b	[1 1]
sz_guess	[1 1]
tol	[1 1]
i	[1 1]
SRmat	[? 6]
iterating	[1 1]
done	[1 1]
h	[1 1]
c	[1 1]
Fa	[1 1]
Fc	[1 1]
Fb	[1 1]
S	[1 1]
SRvec	[1 6]
m	[1 1]
state	[1 1]
n	[1 1]
l	[1 1]
p	[1 1]
SR0vec	[1 6]
err	[1 1]
SR1vec	[1 6]
SR2vec	[1 6]
tol2	[1 1]
a0	[1 1]
b0	[1 1]
tol0	[1 1]
c0	[1 1]
quad	[1 1]

(b) The result of shape analysis

Figure 3.4 The profiling results of the benchmark `adapt` by AspectMatlab and shape analysis

Chapter 4
Range Value Analysis

There are still some remaining problems left in the shape analysis. One of them is how does the shape information propagate through the MATLAB array indexing expressions[1]. The array indexing expression in MATLAB is an array accessed by an index list enclosed in parentheses. The representation of an array indexing expression is:

```
array_name(index_list)
```

The array indexing expressions can exist on both left hand side (which are array indexing set statements) or right hand side (which are array indexing get statements) of assignment expressions, for example:

```
array_name(index_list) = Rvalue or
Lvalue = array_name(index_list)
```

The reason we skip this problem in Chapter 3 is that the shape analysis for MATLAB array indexing expressions is inseparable with the array bounds checking problem in MATLAB. Different from those statically-typed programming languages, like C, C++ or FORTRAN, in which arrays can only be resized by using deallocate and allocate statements manually, an array in MATLAB can automatically grow[2] to a larger size by using out-of-bound index in an array indexing set statement. For instance, in the code example in Figure 4.1, after we

[1] Which exist in the array get statements and array set statements in our Tamer IR.

[2] We define the situation where the size of an array changes to a larger one as the array growth.

```
1 function foo6()
2 arr1 = ones(3,3); % the shape of arr1 is [3,3].
3 arr1(3,4) = 2; % is it an error in MATLAB?
4 end
```

Figure 4.1 MATLAB code example of array growth

define the shape of array variable `arr1` to `[3,3]` at line 2, we try to access it with the index list `(3,4)` and assign `2` to the element in that position. If you are a programmer from the background of a statically-typed programming language, you may think this is definitely incorrect. But, surprisingly, this is allowed in MATLAB. MATLAB allows some array set assignment to grow the shape of the indexed array. Back to the example, the shape analysis will compare the shape of `arr1`, which is `[3,3]`, with the two indices, which are `3` and `4`. After finding that the second index is greater than the size of the array's second dimension, the shape analysis will grow the shape of the array from `[3,3]` to `[3,4]`.[3] Although this may be the simplest case where an array's shape grows by an out-of-bound index, we still can see that we require a valid array bounds checking, so that the shape analysis can determine when an array indexing expression may alter the shape of the result.

Furthermore, what if an array is indexed by a scalar variable without a constant value? For example, in Figure 4.2 (a), `k` may have different value at line 8. The naive solution is that in the corresponding generated FORTRAN code, we declare `arr1` as an allocatable array variable and then inline the run-time array bounds checking code and reallocation code[4] for that array indexing set assignment. What if although the index `k` doesn't have a constant value, neither of the `k`'s possible values exceeds the bounds of `arr1`? For instance, in the code example in Figure 4.2 (b), neither of the possible values of k exceeds the array bounds of the array `arr1`. In Figure 4.3, it is the generated FORTRAN code for the code example in Figure 4.2 (b) with the inlined run-time array bounds checking code and reallocation code. In this situation, inlining run-time array bounds checking code and reallocation code is obviously unnecessary.

To remove unnecessary inlined run-time array bounds checking code and reallocation

[3] A more detailed algorithm for the array growth problem is given in latter section of this chapter.
[4] Which is an overhead at the runtime to the performance of the generated FORTRAN code.

```
function foo7(n)
arr1 = ones(3,3);
if (n>10)
  k = 2;
else
  k = 4;
end
arr1(3,k) = 2; % k may have value of 2 or 4 at runtime.
end
```

(a) The index `k` may exceed the array bounds

```
function foo7(n)
arr1 = ones(3,3);
if (n>10)
  k = 2;
else
  k = 3;
end
arr1(3,k) = 2; % k may have value of 2 or 3 in runtime.
end
```

(b) Neither of the `k`'s possible values will exceed the array bounds

Figure 4.2 MATLAB code example of array growth with non-constant value

code, we extend our constant value analysis to *range value analysis*. Like the shape analysis, the range value analysis is also implemented as a component analysis plugged into the Tamer's abstract value analysis framework. Since the shape analysis may need the result of the range value analysis, we run the range value analysis before the shape analysis. Similarly, the range value analysis depends on the result of the constant value analysis, so we run constant value analysis before the range value analysis. The final order of these three value analyses is: the constant propagation comes first, then the range value analysis and finally the shape analysis.

The range value analysis is an analysis to estimate the value(s) a variable can assume at each point in the program by estimating the minimum and maximum values each variable can reach. The range value of a variable is a pair of values in the *domain of the range values*: the first one represents the minimum possible value, which we call the lower bound; and the second one represents the maximum possible value, which we call the upper bound. Therefore, the range value information of a given variable is represented as:

```
FUNCTION foo7(n)
USE mod_ones
IMPLICIT NONE
DOUBLE PRECISION :: n
DOUBLE PRECISION , DIMENSION(:,:) , ALLOCATABLE :: arr1, arr1_bk
DOUBLE PRECISION :: k
INTEGER(Kind=4) :: arr1_d1max
INTEGER(Kind=4) :: arr1_d2max
INTEGER(Kind=4) :: arr1_d1
INTEGER(Kind=4) :: arr1_d2
! insert runtime allocation.
IF (ALLOCATED(arr1)) THEN
   DEALLOCATE(arr1);
   ALLOCATE(arr1(3,3));
ELSE
   ALLOCATE(arr1(3,3));
END IF
! end
arr1 = ones(3,3);
IF (n .GT. 10.0) THEN
   k = 2;
ELSE
   k = 3;
ENDIF
! insert runtime array bounds check and reallocation.
arr1_d1 = SIZE(arr1, 1);
arr1_d2 = SIZE(arr1, 2);
IF ((3 > arr1_d1) .OR. (k > arr1_d2)) THEN
   IF (ALLOCATED(arr1_bk)) THEN
      DEALLOCATE(arr1_bk);
   END IF
   ALLOCATE(arr1_bk(arr1_d1, arr1_d2));
   arr1_bk = arr1;
   DEALLOCATE(arr1);
   arr1_d1max = MAX(3, arr1_d1);
   arr1_d2max = MAX(INT(k), arr1_d2);
   ALLOCATE(arr1(arr1_d1max, arr1_d2max));
   arr1(1:arr1_d1, 1:arr1_d2) = arr1_bk(1:arr1_d1, 1:arr1_d2);
END IF
! end
arr1(3, INT(k)) = 2.0;
END FUNCTION
```

Figure 4.3 Generated FORTRAN code v1.0 for the MATLAB code in Figure 4.2 (b)

```
<lower bound, upper bound>
```

The domain of the range values is a closed numeric value interval, ordered by including a smallest element (-inf, the range value decreasing to the negative infinity), all the real number elements, and a largest element (+inf, the range value increasing to the positive infinity). Moreover, to support range value analysis through control flow statements, we add two special symbols to real numbers, + and -. We put these two symbols as the superscripts of real numbers, for example, 5^+ or 5^-. You can interpret these symbols as follows. Consider there is a ε, ε is positive and close to 0, where 5^+ means 5+ε and 5^- means 5-ε. Back to the representation of the range values, if the range value of a variable represented as <10,+inf>, it means that at this point of the program, the variable may be any value greater than or equal to 10 to +inf, and if the range value of a variable represented as <10^+,+inf>, it means that at this point of the program, the variable may be any value greater than but not equal to 10 to +inf. Moreover, the lower bound in a range value can only be one of -inf, any real number and any real number with + superscript, and the upper bound in a range value can only be one of +inf, any real number and any real number with - superscript.

In MATLAB, besides scalar variables, a variable can also refer to an array. In order to support the range values of the array variables, we should first figure out what is the meaning of the range value of an array variable. Intuitively, the range value of an array should be the result from merging the range values of all its elements. But, it's much harder to define the range value for an array than for a scalar in MATLAB. For instance, the built-in function rand returns an array containing pseudo-random values drawn from the standard uniform distribution on the open interval (0,1). It's hard to define a more precise range value for it than <0,1>. Moreover, the built-in function svd will produce a vector containing singular values. Defining the return range value for this function needs a complicated computation which involves a lot of overhead during the analysis. To balance between the efficiency of the analysis and its accuracy, we need to support array bounds checking in the shape analysis. Unlike the shape analysis which supports shapes on both scalar variables and array variables, the range value analysis only supports the analysis in the domain of the range values for scalar variables.

By involving the range value analysis, the problem in the above example in Figure 4.2 (b)

can be solved. In detail, after the if-else statement, the possible values of k at line 8 can be either 2 or 3, whose range value is represented as <2,3>. By comparing the upper bound of k's range value, which is 3, with the original size of arr1's second dimension, which is 3, the shape analysis confirm that the index k won't exceed the array bounds of arr1. Therefore, the FORTRAN code generation phase doesn't need to inline the run-time array bounds checking code and reallocation code before this statement, which means that the performance overhead from inlining the run-time checks and reallocations is removed. The corresponding generated FORTRAN code is in Figure 4.4, which is much more readable and efficient than the generated code in Figure 4.3.

```
FUNCTION foo7(n)
USE mod_ones
IMPLICIT NONE
DOUBLE PRECISION :: n
DOUBLE PRECISION , DIMENSION(3,3) :: arr1
DOUBLE PRECISION :: k
arr1 = ones(3,3);
IF (n .GT. 10.0) THEN
   k = 2;
ELSE
   k = 3;
ENDIF
arr1(3, INT(k)) = 2.0;
END FUNCTION
```

Figure 4.4 Generated FORTRAN code v2.0 for the MATLAB code in Figure 4.2 (b).

This chapter is composed of three sections. We start with the equations for propagating range values through certain MATLAB built-in operators (functions) in Section 4.1, then we present a solution to merge different range values in Section 4.2, and finally in Section 4.3, we end the chapter with an algorithm to perform the shape analysis through array indexing expressions by using the result from the range value analysis, as well as the static array bounds checking inside the analysis.

4.1 Propagating Ranges through Built-in Functions

All of the currently supported MATLAB built-in operators (functions) in the range value analysis are listed in Table 4.1. These twelve built-in operators are the most commonly used built-in operators in MATLAB programs and the range values propagating through these operators are easily inferred.

Table 4.1 Operators supported by the range value analysis

unary plus operator (+)	binary plus operator (+)
unary minus operator (−)	binary minus operator (−)
element-wise multiplication operator (`.*`)	matrix multiplication operator (`*`)
element-wise rdivision operator (`./`)	matrix rdivision operator (`/`)
natural logarithm operator (`log(x)`)	exponential operator (`exp(x)`)
absolute value operator (`abs(x)`)	colon operator (`:`)

We implement a *range value propagation function* for each of above operators to illustrate how the range values propagate through them. Before introducing all the range value propagation functions, we need to define some operations on the values in the range value domain. The order of values in the range value domain is: the smallest value, `-inf`, is smaller than any other values, then the real numbers and the real numbers with the superscripts of + or − are in the same order as they are in the algebra, finally, the largest value, `+inf`, is greater than any other values. With the definition of the order of the values in the range domain, we have:

min: find the minimum value in all the given values according to the order given above.

max: find the maximum value in all the given values according to the order given above.

equals (==): `-inf` only equals `-inf`; `+inf` only equals `+inf`; the equals operation on real numbers and real numbers with superscripts has the same meaning as in the algebra.

unary +: keep the original value.

unary -: change the signs of the original values. For example, `-(-inf)` goes to `+inf`, and $-(10^-)$ goes to -10^+.

+ (plus): if any operand is `-inf` (`+inf`), the result will be `-inf` (`+inf`); if neither of the operands is `-inf` nor `+inf`, the + operator follows the rule as:
$x^- + y^-$, $x^- + y$ or $x + y^-$, where x and y are real numbers[5] $\Rightarrow (x+y)^-$;
$x^+ + y^+$, $x^+ + y$ or $x + y^+ \Rightarrow (x+y)^+$;
$x + y \Rightarrow (x+y)$;
when + applies on real numbers, the result will be the same as in the algebra.

- (minus): if the first operand is `-inf` (`+inf`), the result will be `-inf` (`+inf`); if the second operand is `-inf` (`+inf`), the result will be `+inf` (`-inf`); if neither of the operands is `-inf` nor `+inf`, the - operator follows the rule as:
$x - y^+$, $x^- - y^+$ or $x^- - y \Rightarrow (x-y)^-$;
$x - y^-$, $x^+ - y^-$ or $x^+ - y \Rightarrow (x-y)^+$;
$x - y \Rightarrow (x-y)$;
when - applies on real numbers, the result will be the same as in the algebra.

$\times$ (times): The $\times$ operations on the values in the domain of the range values are a little bit complicated and it follows the rule as:
`-inf` $\times\, x$ or $x \times$ `-inf`, $x < 0 \Rightarrow$ `+inf`;
$, x = 0 \Rightarrow$ `0`;
$, x > 0 \Rightarrow$ `-inf`;
`+inf` $\times\, x$ or $x \times$ `+inf`, $x < 0 \Rightarrow$ `-inf`;
$, x = 0 \Rightarrow$ `0`;
$, x > 0 \Rightarrow$ `+inf`;
if neither of the operands is `-inf` nor `+inf`, the $\times$ operator follows the rule as:
$x^+ \times y \Rightarrow (x \times y)^s$, where s is the sign of y;
$x^+ \times y^+ \Rightarrow (x \times y)^s$, where s is the sign of the result of $x+y$;
$x \times y^+ \Rightarrow (x \times y)^s$, where s is the sign of x;
$x^- \times y \Rightarrow (x \times y)^s$, where s is the opposite sign of y;
$x^- \times y^- \Rightarrow (x \times y)^s$, where s is the opposite sign of the result of $x+y$;
$x \times y^- \Rightarrow (x \times y)^s$, where s is the opposite sign of x;
$x^+ \times y^-$ or $x^- \times y^+ \Rightarrow (x \times y)^s$, where s is the sign of the result of $x-y$;

[5] Assuming that all the following x and y are real numbers.

$x \times y$, x and y are real numbers $\Rightarrow (x \times y)$;
when $\times$ applies on real numbers, the result will be the same as in the algebra.

÷ (divide): As with the × operations, the ÷ operations are also a little bit complicated and follow the rule as:
`-inf` $\div x$, `x` $< 0 \Rightarrow$ `+inf`;
, `x` $> 0 \Rightarrow$ `-inf`;
$x \div$ `-inf`, `x` $< 0 \Rightarrow 0^+$;
, `x` $> 0 \Rightarrow 0^-$;
`+inf` ÷ `x`, `x` $< 0 \Rightarrow$ `-inf`;
, `x` $> 0 \Rightarrow$ `+inf`;
$x \div$ `+inf`, `x` $< 0 \Rightarrow 0^-$;
, `x` $> 0 \Rightarrow 0^+$;
if neither of the operands is `-inf` nor `+inf`, the ÷ operator follows the rule as:
$x^+ \div y \Rightarrow (x \div y)^s$, where s is the sign of y;
$x^+ \div y^+ \Rightarrow (x \div y)^s$, where s is the sign of the result of $y - x$;
$x \div y^+ \Rightarrow (x \div y)^s$, where s is the opposite sign of x;
$x^- \div y \Rightarrow (x \div y)^s$, where s is the opposite sign of y;
$x^- \div y^- \Rightarrow (x \div y)^s$, where s is the sign of the result of $x - y$;
$x \div y^- \Rightarrow (x \div y)^s$, where s is the sign of x;
$x^+ \div y^- \Rightarrow (x \div y)^s$, where s is the sign of the result of $x + y$;
$x^- \div y^+ \Rightarrow (x \div y)^s$, where s is the opposite sign of the result of $x + y$;
$x \div y$, x and y are real numbers $\Rightarrow (x \div y)$;
when ÷ applies on real numbers, the result will be the same as in the algebra.

log: log of the values in the range domain greater than 0 but not `+inf` goes to the same result as in algebra; log of `+inf` goes to `+inf`.

exp: exp of `-inf` (`+inf`) goes to `0` (`+inf`); if the operand is neither `-inf` nor `+inf`, the result will be the same as in algebra.

In the following listings from 4.1 to 4.12, there is the pseudocode of the range value propagation functions for the built-in operators in Table 4.1. In the pseudocode, the arith-

metic operations on the values in the range value domain follow the rules listed above. Note that the result from colon operator is a vector, not a scalar. Although we only support range value analysis for scalar variables, since colon operator is used frequently in array indexing in MATLAB and not hard to infer the range value result from the operator, we decide to also support range value analysis for it.

```
function range_value_unary_plus(operand_a)
  if operand_a has a known range value
    <a,b> = get range value pair from operand_a
    return <a,b>
  else
    return unknown
  end if
end function
```

Listing 4.1 Unary plus operator (+)

```
function range_value_binary_plus(operand_a, operand_b)
  if both operand_a and operand_b have known range values
    <a,b> = get range value pair from operand_a
    <c,d> = get range value pair from operand_b
    return <a+c,b+d>
  else
    return unknown
  end if
end function
```

Listing 4.2 Binary plus operator (+)

```
function range_value_unary_minus(operand_a)
  if operand_a has a known range value
    <a,b> = get range value pair from operand_a
    return <-b,-a>
  else
    return unknown
  end if
end function
```

Listing 4.3 Unary minus operator (-)

```
function range_value_binary_plus(operand_a, operand_b)
  if both operand_a and operand_b have known range values
    <a,b> = get range value pair from operand_a
    <c,d> = get range value pair from operand_b
    return <a-d,b-c>
  else
    return unknown
  end if
end function
```

Listing 4.4 Binary minus operator (-)

```
function range_value_ew_multiply(operand_a, operand_b)
  if both operand_a and operand_b have known range values
    <a,b> = get range value pair from operand_a
    <c,d> = get range value pair from operand_b
    return <min(a×c, a×d, b×c, b×d),max(a×c, a×d, b×c, b×d)>
  else
    return unknown
  end if
end function
```

Listing 4.5 Element-wise multiplication operator (.*)

```
function range_value_mat_multiply(operand_a, operand_b)
  if both operand_a and operand_b have scalar values
    return range_value_ew_multiply(operand_a, operand_b)
  else
    return unknown
  end if
end function
```

Listing 4.6 Matrix multiplication operator (*)

```
function range_value_ew_rdivide(operand_a, operand_b)
  if both operand_a and operand_b have known range values
    <a,b> = get range value pair from operand_a
    <c,d> = get range value pair from operand_b
    return <min(a÷c, a÷d, b÷c, b÷d),max(a÷c, a÷d, b÷c, b÷d)>
  else
    return unknown
  end if
end function
```

Listing 4.7 Element-wise rdivision operator (./)

```
function range_value_mat_rdivide(operand_a, operand_b)
  if both operand_a and operand_b have scalar values
    return range_value_ew_rdivide(operand_a, operand_b)
  else
    return unknown
  end if
end function
```

Listing 4.8 Matrix rdivision operator (/)

```
function range_value_natural_log(operand_a)
  if operand_a has a known range value
    <a,b> = get range value pair from operand_a
    if a > 0 # if a < 0, the result will be complex number.
      return <log(a),log(b)>
    else
      return unknown
    end if
  else
    return unknown
  end if
end function
```

Listing 4.9 Natural logarithm operator (log)

```
function range_value_exp(operand_a)
  if operand_a has a known range value
    <a,b> = get range value pair from operand_a
    return <exp(a),exp(b)>
  else
    return unknown
  end if
end function
```

Listing 4.10 Exponential operator (exp)

```
function range_value_abs(operand_a)
  if operand_a has a known range value
    <a,b> = get range value pair from operand_a
    if b < 0
      return <-b,-a>
    else if a > 0
      return <a,b>
    else
      return <min(0, abs(a), b), max(abs(a), b)>
    end if
  else
    return unknown
  end if
end function
```

Listing 4.11 Absolute value operator (abs)

```
function range_value_colon(operand_a, operand_b)
  if both operand_a and operand_b have known range values
    <a,b> = get range value pair from operand_a
    <c,d> = get range value pair from operand_b
    return <min(a,c), max(b,d)>
  else return unknown
  end if
end function
```

Listing 4.12 Colon operator (:)

Besides above twelve built-in operators in Table 4.1, the range value of a variable can also be updated from control flow statements, like if-else, for loop and while loop statement, in MATLAB. For example, if the conditional expression in an if clause is `var < 5`, we can confidently infer that in the if branch, the upper bound of the variable `var` is smaller than `5`, which can be represented as <something,5^->. More precisely speaking, to support range value analysis through control flow statements, we need to support range value analysis for five MATLAB built-in relational operators: less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=) and equal to (==). Note that for the range value analysis, we only consider the cases where one of the operand is a variable and the other operand is a constant, because if both operands are constant, there is no need to know the range value of a constant, and if both operands are scalar variables, actually, they are comparing their run-time values when executing the program.

less than: If the left hand side is a scalar variable, update the upper bound of the variable to the value of the right hand side constant minus ε; if the right hand side is a scalar variable, update the lower bound of the variable to the value of the left hand side constant plus ε.

less than or equal to: If the left hand side is a scalar variable, update the upper bound of the variable to the value of the right hand side constant; if the right hand side is a scalar variable, update the lower bound of the variable to the value of the left hand side constant.

greater than: If the left hand side is a scalar variable, update the lower bound of the variable to the value of the right hand side constant plus ε; if the right hand side is a scalar variable, update the upper bound of the variable to the value of the left hand side constant minus ε.

greater than or equal to: If the left hand side is a scalar variable, update the lower bound of the variable to the value of the right hand side constant; if the right hand side is a scalar variable, update the upper bound of the variable to the value of the left hand side constant.

equal to: Update both the lower and upper bound of the scalar variable to the value of the constant.

4.2 Merging Different Range Values

With the same reason as the shape analysis, the range value analysis also needs to provide a merging strategy and an `equals` function to the abstract value analysis framework. Intuitively, the merging strategy and the `equals` function can be achieved by applying simple comparisons between the range values of the variables needed to be merged or compared, something like in Listing 4.13 and Listing 4.14.

```
function merge(range_a, range_b)
  if either range_a or range_b is unknown
    return unknown
  else
    <a, b> = get range value pair from range_a
    <c, d> = get range value pair from range_b
    return <min(a,c),max(b,d)>
  end if
end function
```

Listing 4.13 Range value merging strategy

```
function equals(range_a, range_b)
  if both range_a and range_b are unknown
    return true
  else if both range_a and range_b have range value pairs
    <a, b> = get range value pair from range_a
    <c, d> = get range value pair from range_b
    if a == c and b == d
      return true
    end if
  else return false
  end if
end function
```

Listing 4.14 Range value equals function

As in the shape analysis, we also need to handle how to merge different range values in the loop statements. In Figure 4.5, it's a simple code example to demonstrate the problem of merging different range values in a while loop. In the example, since we don't know how many iterations the while loop will take, we cannot determine the upper bound of the variable `a`, and neither the lower bound nor the upper bound of the variable `b`. Recall

```
function foo8(n)
a = 5; % the range value of a is <0,0>
b = 5; % the range value of b is <5,5>
while (n > 10)
  a = a * 5;
  % the upper bound of a cannot be fixed.
  b = b * -5;
  % neither the lower nor the upper
  % bound of b can be fixed.
  n = n - 1;
end
end
```

Figure 4.5 MATLAB code example of merging range values in loops

the upper bound counter number used to stop the shape analysis ending in an infinite loop in analyzing loop statements, we also add a similar-functionality counter number inside the range value analysis. Different from the shape analysis, since the range value has two bounds, we set an upper bound counter numbers for each bound. For each bound, if the iteration times which the range value analysis takes to merge different range values of that bound hit the counter number, the range value of that bound will be pushed to `-inf` or `+inf` respectively. Back to the example in Figure 4.5, the range value information of `a` and `b` after the while loop will be `<5,+inf>` and `<-inf,+inf>` based on this solution. Since in Section 4.1, we list the operations on the values in the range value domain, after involving `-inf` and `+inf`, the merging function in Listing 4.13 and the equals funciton in Listing 4.14 are still working.

4.3 Propagating Shapes through Array Indexing

The array bounds checking is crucial for most modern compilers. Without valid array accesses, the execution result of a program is obviously unsound. For the same reason, we also equip our MC2FOR compiler with an appropriate array bounds checking strategy. The array bounds checking in our compiler consists of two phases: one is the compile-time or static array bounds checking; and the other one is the run-time or dynamic array bounds checking. The static array bounds checking is achieved during the abstract value analysis phase, strictly speaking, during the shape analysis phase. Because the array growth problem, the array bounds check and the shape analysis for array indexing expressions are inseparable. The dynamic array bounds checking is achieved by inlining the run-time array bounds checking code blocks in the generated FORTRAN programs. The inlined run-time array bounds checking code will definitely introduce some overhead during the execution of the generated FORTRAN programs. Recall the generated FORTRAN code in Figure 4.3 and Figure 4.4, the goal of the static array bounds checking is to remove those unnecessary run-time array bounds checking code as much as possible.

Based on the fact that our abstract value analysis framework is built upon the Tamer's simplified three address IR, the array indexing expressions only exist in two kinds of statements: array set statements and array get statements. For array indexing expressions in both array set statements and array get statements, the static array bounds checking will try to determine whether the index is valid, in other words, whether the index is in the array bounds, neither smaller than the lower bound[6] nor greater than the upper bound of the array. The only difference between the static array bounds checking on two statements is the reaction when the checks find an out-of-bound array indexing. The out-of-bound array indexing is definitely a run-time error in an array get statement, while in an array set statement, it may be not a run-time error. The MATLAB will always first try to grow the original array according to the out-of-bound index (or indices) and the shape of original array before throwing an exception. For example, in the code example in Figure 4.6 (a), the shape of `arr` will be `[5,5]` after line 2, although the index `(5,10)` obviously exceeds

[6]Mostly, the lower bound of an array is 1.

```
1 function foo9()
2 arr = ones(5,5);
3 arr(5,10) = 5;
4 end
```

(a) An out-of-bound array indexing growing the original array

```
1 function foo9()
2 arr = ones(5,5);
3 arr(50) = 5;
4 end
```

(b) An out-of-bound array indexing throwing run-time error

Figure 4.6 MATLAB code example of out-of-bound array set assignment

the upper bound of the array, the array indexing will grow the shape of the array to `[5,10]` at line 3. While for the code example in Figure 4.6 (b), the array indexing expression at line 3 will not grow the original array, since the MATLAB interpreter does not have enough information to resize `arr` based on its current shape and the only index, `50`. In this case, MATLAB will throw a run-time exception.[7] Based on these two examples, we can imply that the MATLAB interpreter will always try to resize the original array when there is an out-of-bound array indexing in an array set statement before throwing an exception.

4.3.1 Brief Introduction of Array Indexing in MATLAB

Before showing the algorithm of the shape analysis for the array indexing expressions, we need a brief introduction of the array indexing in MATLAB. Indexing into an array is a means of selecting a subset of elements from the array and the index of an array is also called the subscript. According to our experience and a MATLAB online documentation[8], the valid subscripts in MATLAB can be divided into five categories: (1) scalar literals, (2) scalar variables, (3) vector expressions, (4) vector variables and (5) colon notation (`:`). In Figure 4.7, there is some MATLAB code to illustrate how these five kinds of subscripts

[7] The exception for this out-of-bound indexing is "`In an assignment A(I)= B, an array A cannot be resized.`"

[8] Array Indexing in MATLAB, by Steve Eddins and Loren Shure, MathWorks, http://www.mathworks.com/company/newsletters/articles/matrix-indexing-in-matlab.html

work in accessing MATLAB arrays. Note that the special built-in operator `end` at line 13 and 25 is used to return the last position of the indexed array. The single colon notation (`:`) in a subscript position at line 34 is shorthand for `1:end` and is often used to select an entire row or column.

Besides various kinds of subscripts, the relation between the number of the indices and the rank of the accessed array is also very interesting. In MATLAB, the number of the indices doesn't need to be equal to the rank of the accessed array as in some programming languages, like FORTRAN. In some circumstances, it is legal to access an array with indices whose number is less than or even greater than the rank of the accessed array.

For the case where the number of the indices is less than the rank of the accessed array, the MATLAB interpreter will use the last index in the index list to perform linear indexing on the remaining dimensions of the accessed array. For example, in Figure 4.8 (a), the array `arr` is a 2 dimensional array and there is only one subscript. In this case, the only index `3` is regarded as the last index and all the two dimensions of `arr` is regarded as the remaining dimensions. The MATLAB interpreter will iterate `arr` in the column-major order to apply linear indexing. Accessing the array with the subscript `3` at line 6 returns `3`, because `3` is the value of the third element of the array in column-major order. Another example is in Figure 4.8 (b). In this example, the array `arr` has three dimensions and is accessed by the index list `(2,3)` at line 10. In this example, the number of the indices is less than the rank of `arr`. The interpreter will use the last index `3` to proceed the linear indexing on `arr`'s remaining dimensions, which are the second and third dimensions. Since the third element of the remaining dimensions has the value of `3`, the array indexing with `(2,3)` returns `3`.

When the number of the indices is greater than the rank of the accessed array, if this array access is in an array get statement, it's definitely a run-time error , but if it is in an array set statement, the MATLAB interpreter first tries to grow the original array according to the extra index (or indices), if the endeavor fails, the interpreter throws a run-time error. For example in Figure 4.9 (a), at line 2, the number of the indices `(2,2,2)`, which is 3, is greater than the rank of the array `arr`, which is only 2. Since this array accessing is on the left hand side of the assignment, the MATLAB interpreter first tries to grow the original array. For this case, the interpreter succeeds in resizing the array, so the resized array becomes a 2-by-2-by-2 array. While in Figure 4.9 (b), at line 2, although the array

```
1  % constructing a vector
2  >> v = [2 4 6 8 10 12 14 16];
3  % the subscript is a scalar literal
4  >> v(3)
5  ans =
6        6
7  % the subscript is a scalar variable
8  >> idx = 3;
9  >> v(idx)
10 ans =
11       6
12 % the subscript is the special end operator
13 >> v(end)
14 ans =
15       16
16 % the subscript is a vector expression
17 >> v([1 2 3])
18 ans =
19       2 4 6
20 % the subscript is a vector expression
21 >> v(1:3)
22 ans =
23       2 4 6
24 % the subscript is a vector expression
25 >> v(1:end)
26 ans =
27       2 4 6 8 10 12 14 16
28 % the subscript is a vector variable
29 >> idxv = 1:3;
30 >> v(idxv)
31 ans =
32       2 4 6
33 % the subscript is a colon notation
34 >> v(:)
35 ans =
36       2 4 6 8 10 12 14 16
```

Figure 4.7 Illustration of MATLAB array indexing

```
1  >> arr = [[1; 2; 3] [4; 5; 6] [7; 8; 9]]
2  arr =
3       1 4 7
4       2 5 8
5       3 6 9
6  >> arr(3)
7  ans =
8       3
```

(a) Linear indexing example a

```
1  >> arr = ones(3,3,3);
2  >> arr(2,:,:) = [[1; 2; 3] [4; 5; 6] [7; 8; 9]];
3  >> arr(2,:,:)
4  ans(:,:,1) =
5       1 2 3
6  ans(:,:,2) =
7       4 5 6
8  ans(:,:,3) =
9       7 8 9
10 >> arr(2,3)
11 ans =
12      3
```

(b) Linear indexing example b

Figure 4.8 MATLAB script example of linear indexing

accessing is on the left hand side of the assignment, the MATLAB interpreter cannot succeed in resizing the array `arr` with the extra index colon notation. But, at line 4, the extra index `1:3` gives the interpreter more information about how the programmer wants to resize the array, and the interpreter succeeds in resizing the array `arr`.

In the following two subsections, we present the shape analysis for both array set statements and array get statements. Inside the shape analysis, we integrate the static array bounds checking to handle the potential array growth problem for the array indexing expressions in array set statements. Since MATLAB requires that subscript indices must be real positive integers or logicals, the bottom line to proceed static array bounds checking is that the shape of the accessed array is exactly known, in other words, the sizes of all the dimensions of the array are exactly integer numbers, and both bounds of the range values

```
1  >> arr = ones(2,2);
2  >> arr(2,2,2) = 0
3  arr(:,:,1) =
4        1 1
5        1 1
6  arr(:,:,2) =
7        0 0
8        0 0
9  >> size(arr)
10 ans =
11       2 2 2
```

(a) Succeed in resizing the array with extra index

```
1  >> arr = ones(2,2);
2  >> arr(2,2,:) = rand(2,2)
3  Assignment has more non-singleton rhs dimensions than
      non-singleton subscripts
4  >> arr(2,2,1:3) = [4 5 6];
5  >> size(arr)
6  ans =
7        2 2 3
```

(b) More examples of resizing the array with extra index

Figure 4.9 MATLAB script example of array growth

of the indices are exactly real[9] numbers. If these two requirements cannot be met, we have to leave the array bounds checking to the inlined FORTRAN code.

4.3.2 For Array Set Statement

The array set statement is one kind of assignment statement where the array accessing is at the left hand side of the assignment. This statement is used to reset one or more elements selected by the subscript(s) in the array. Based on the description of the array indexing in last subsection, the indices can be narrowed down to three different kinds: (1) colon notation, (2) having a scalar value, or (3) having a vector value. And for each kind of index, the relation between the number of the indices and the rank of the accessed array has three possibilities: (1) less than, (2) equal to, or (3) greater than. So the shape analysis

[9] MATLAB can cast real number index to integer number index implicitly.

should appropriately handle all these nine combination possibilities of the indices for a given array set statement. Moreover, when accessing an array with an out-of-bound index in an array set statement, the out-of-bound array indexing may grow the accessed array, which means the shape information of the accessed array may be changed. If the out-of-bound array indexing cannot grow the shape of the array, which corresponds to a run-time error, the shape analysis should inform the abstract value analysis framework to mark the current flow set as `nonviable`[10]. The pseudocode of the shape analysis algorithm for array set statements is in Listing 4.15.

```
function handle_array_set_stmt(array, index_list)
 # for array set statements, the shape analysis
 # result returns to the indexed array.
 arr_dim_ls = get dimension list of the array
 POS = 1 # initialing to point at the first index in the index_list
 for each index in the index_list
   # case 1: colon notation
   if the index is a colon notation
     if POS > arr_dim_ls.length
       # out-of-bound index with colon notation
       mark the current flow_set as nonviable
     else if POS == index_list.length and POS < arr_dim_ls.length
       # proceed linear indexing
       # won't resize the shape of the array
     else
       # the ordinary case
       # won't resize the shape of the array
     end if
   # case 2: scalar value
   else if the index has a scalar value
     if POS > arr_dim_ls.length
       # out-of-bound index with a scalar
       range_val = get range value of index_list[POS]
       if range_val.lower_bound < 1
         mark the current flow_set as nonviable
       else
         add range_val.upper_bound to the end of arr_dim_ls
       end if
```

[10] In the Tamer's abstract value analysis framework, the `nonviable` flow sets represent the non-reachable code (for statements after errors, or non-viable branches).

```
    else if POS == index_list.length and POS < arr_dim_ls.length
      # proceed linear indexing
      range_val = get the range value of the index
      if range_val.lower_bound < 1
        mark the current flow_set as nonviable
      else
        num = get the number of elements in the remaining dimensions
        if range_val.upper_bound > num
          mark the current flow_set as nonviable
        else
          # won't resize the shape of the array
        end if
      end if
    else
      # the ordinary case
      range_val = get range value of the index
      if range_val.lower_bound < 1
        mark the current flow_set as nonviable
      else
        if range_val.upper_bound > arr_dim_ls[POS]
          # out-of-bound index with a scalar
          arr_dim_ls[POS] = range_val.upper_bound
        else
          # won't resize the shape of the array
        end if
      end if
    end if
  # case 3: vector value
  else if the index has a vector value
    if POS > arr_dim_ls.length
      # out-of-bound index with a scalar
      range_val = get range value of the index
      if range_val.lower_bound < 1
        mark the current flow_set as nonviable
      else
        add range_val.upper_bound to the end of arr_dim_ls
      end if
    else if POS == index_list.length and POS < arr_dim_ls.length
      # proceed linear indexing
      range_val = get range value of the index
      if range_val.lower_bound < 1
        mark the current flow_set as nonviable
```

```
        else
          num = get the number of elements in the remaining dimensions
          if range_val.upper_bound > num
            mark the current flow_set as nonviable
          else
            # won't resize the shape of the array
          end if
        end if
      else
        # the ordinary case
        range_val = get range value of the index
        if range_val.lower_bound < 1
          mark the current flow_set as nonviable
        else
          if range_val.upper_bound > arr_dim_ls[POS]
            # out-of-bound index with a scalar
            arr_dim_ls[POS] = range_val.upper_bound
          else
            # won't resize the shape of the array
          end if
        end if
      end if
    end if
    increment POS
  end loop
  shape_result = new shape from arr_dim_ls
  return shape_result
end function
```

Listing 4.15 Shape analysis for array set statements

4.3.3 For Array Get Statement

The array get statement is another kind of assignment statement where the array accessing is at the right hand side of the assignment. This statement is used to assign one or more elements selected by the subscript(s) in the array to the variable on the left hand side. The array indexing expressions in the array get statements cannot resize the accessed array, but since FORTRAN doesn't have built-in array bounds checking for array indexing, we have to either perform array bounds checking in the compile-time or inline run-time array

bounds checking code in the generated FORTRAN code. The compile-time array bounds checking is achieved via the shape analysis for array get statements. The same as the array set statement, the shape analysis handles properly all those nine combinations of the indices introduced in last subsection and if the out-of-bound array indexing occurs, the shape analysis informs the abstract value analysis framework to mark the current flow set as `nonviable`. The pseudocode of the shape analysis algorithm for array get statements is in Listing 4.16.

```
function handle_array_get_stmt(target, array, index_list)
  # for array get statements, the shape analysis result
  # returns to the left hand side target variable.
  new_dim_ls = new a dimension list
  arr_dim_ls = get dimension list of the array
  POS = 1 # initializing to point at the first index in the index_list
  for each index in the index_list
    # case 1: colon notation
    if the index is a colon notation
      if POS > arr_dim_ls.length
        # out-of-bound index with colon notation
        mark the current flow_set as nonviable
      else if POS == index_list.length and POS < arr_dim_ls.length
        # proceed linear indexing
        # won't resize the shape of the indexed array
        # but has to set the dimension size for target
        num = get the number of elements in the remaining dimensions
        add num to the end of new_dim_ls
      else
        # the ordinary case
        # won't resize the shape of the indexed array
        # but has to set the dimension size for target
        add arr_dim_ls[POS] to the end of new_dim_ls
      end if
    # case 2: scalar value
    else if the index has a scalar value
      if POS > arr_dim_ls.length
        # out-of-bound index with a scalar
        mark the current flow_set as nonviable
      else if POS == index_list.length and POS < arr_dim_ls.length
        # proceed linear indexing
        range_val = get range value of the index
```

```
    if range_val.lower_bound < 1
     mark the current flow_set as nonviable
    else
     num = get the number of elements in the remaining dimensions
     if range_val.upper_bound > num
      mark the current flow_set as nonviable
     else
      # won't resize the shape of the array
     end if
    end if
    # to set the dimension size for target
    add 1 to the end of new_dim_ls
   else
    # the ordinary case
    range_val = get range value of the index
    if range_val.lower_bound < 1
     mark the current flow_set as nonviable
    else
     if range_val.upper_bound > arr_dim_ls[POS]
      # out-of-bound index with a scalar
      mark the current flow_set as nonviable
     else
      # won't resize the shape of the array
     end if
    end if
    # to set the dimension size for target
    add 1 to the end of new_dim_ls
   end if
  # case 3: vector value
  else if the index has a vector value
   if POS > arr_dim_ls.length
    # out-of-bound index with a scalar
    mark the current flow_set as nonviable
   else if POS == index_list.length and POS < arr_dim_ls.length
    # proceed linear indexing
    range_val = get range value of the index
    if range_val.lower_bound < 1
     mark the current flow_set as nonviable
    else
     num = get the number of elements in the remaining dimensions
     if range_val.upper_bound > num
      mark the current flow_set as nonviable
```

```
          else
            # won't resize the shape of the array
          end if
        end if
        # to set the dimension size for target
        add unknown to the end of new_dim_ls
      else
        # the ordinary case
        range_val = get range value of the index
        if range_val.lower_bound < 1
          mark the current flow_set as nonviable
        else
          if range_val.upper_bound > arr_dim_ls[POS]
            # out-of-bound index with a scalar
            mark the current flow_set as nonviable
          else
            # won't resize the shape of the array
          end if
        end if
        # to set the dimension size for target
        add unknown to the end of new_dim_ls
      end if
    end if
    increment POS
  end loop
  return the shape from new_dim_ls
end function
```

Listing 4.16 Shape analysis for array get statement

Chapter 5
Transforming MATLAB to FORTRAN 95

After solving the problem of the shape analysis through the array indexing expressions, we finally pave the way for transforming MATLAB to FORTRAN 95. Together with other abstract value characteristics, like mclass and complex information, our MC2FOR compiler is able to declare all the variables of a given MATLAB program in the generated FORTRAN code, which is crucial for a statically-typed language. Besides the variable declaration, we should also map those types in MATLAB to the corresponding types in FORTRAN. For example, in MATLAB, there are `double`, `int8`, `int16`, `int32`, `char` and some other types; how to map them into FORTRAN intrinsic types? Furthermore, there are also a lot of constructs mapping issues. The original intention of designing MATLAB was to make this language a sort of dynamic version of FORTRAN, but after evolving more than 30 years, the syntax of MATLAB has become quite different from the syntax of FORTRAN and even some syntax or features look quite wild from the point of view of a FORTRAN programmer.

In this chapter, we start with a brief review of the history of MATLAB in Section 5.1, including elaborating the reason why we pick FORTRAN 95 instead of any other version of FORTRAN and presenting some potential issues when translating MATLAB to FORTRAN. In Section 5.2, we list some basic transformations from MATLAB to FORTRAN, like mapping built-in types, built-in functions or operators, control flow statements, main entry point functions and user-defined functions. Besides those basic transformations, we also discuss some advanced mapping problems, like mapping types for subscripts in the array indexing expressions, variables in the loop range expressions and assigning multiple types to the

same variable in Section 5.3, transforming flexible array indexing in MATLAB to more rigorous array indexing in FORTRAN in Section 5.4 , and finally inlining the run-time array bounds checking and variable's shape resizing code in Section 5.5.

5.1 Introduction

According to the article "The Origins of MATLAB" [Molb] written by the creator of MATLAB, Clever Moler, MATLAB is designed by him to give his students the access to LINPACK and EISPACK[1] without them having to learn FORTRAN. We may believe the original intention of designing MATLAB is to make it a dynamic or script version of FORTRAN, which means that the programmers can have access to the FORTRAN libraries without declaring types for all the used variables and remembering long input argument lists when calling subroutines from the libraries. In 2000, according to the article "MATLAB Incorporates LAPACK" [Mola] written by Clever Moler, MATLAB was rewritten to use a newer set of FORTRAN libraries for matrix manipulation, LAPACK. For these reasons, MATLAB has similar syntax to FORTRAN. Based on above facts, we picked FORTRAN as the target language of translating MATLAB. In this section, we further explain the reason why we choose FORTRAN 95 among any other version of FORTRAN as our target language and list some potential problems when translating MATLAB to FORTRAN.

5.1.1 Why FORTRAN 95?

In the versions of FORTRAN prior to 90, like 66 and 77, there are many features not supported, i.e. operator overloading, derived (structured) data types, dynamic memory allocation and so on, which are necessary to translate MATLAB to FORTRAN. Therefore, we directly start the comparison from FORTRAN 90 and other following versions. Here, first we list some features in FORTRAN 90, which are essential for translating MATLAB to FORTRAN.

[1] These two software libraries are written in FORTRAN for performing numerical linear algebra on digital computers.

- Inline comments;
- Ability to operate on arrays (or array sections) as a whole, the greatly simplifying math and engineering computations;
- Recursive procedures;
- Operator overloading;
- Derived (structured) data types;
- Dynamic memory allocation;
- New and enhanced intrinsic procedures.

Then FORTRAN 95 is a minor version, mostly to resolve some outstanding issues from the FORTRAN 90 version. Some new features which are also useful for converting MATLAB to FORTRAN are:

- Add a number of extensions, notably from the high performance FORTRAN specification, which add constructs supporting parallel computing;
- Clearly defined that `ALLOCATABLE` arrays are automatically deallocated when they go out of scope;
- Several features noted in FORTRAN 90 to be "obsolescent" were removed from FORTRAN 95, which makes the language more concise and cleaner.

After FORTRAN 95, there are also two new versions of FORTRAN, which are FORTRAN 2003 and FORTRAN 2008. These two versions provide new features to support object-oriented programming, enhancements of old features, interoperability with C and also parallel execution. Here is a brief list of some features in FORTRAN 2003 and 2008.

- Object-oriented programming support: type extension and inheritance, polymorphism, dynamic type allocation, and type-bound procedures, providing complete support for abstract data types;

- Derived type enhancements;
- Input/output enhancements;
- Procedure pointers (the same as function pointers in C);
- Interoperability with the C programming language;
- Enhanced integration with the host operating system: access to command line arguments, environment variables, and processor error messages.
- Coarray FORTRAN, a parallel execution model;
- The `DO CONCURRENT` construct, for loop iterations with no inter-dependencies;
- The `BLOCK` construct, containing declarations of objects with construct scope.

Since our static analysis framework and code generation only focus on the numeric computation aspect of MATLAB, we may not consider the object-oriented programming aspect of MATLAB for now. There is also the fact that most current FORTRAN compilers are only fully compliant with the FORTRAN 90/95 version. Although some FORTRAN compilers can support a significant number of FORTRAN 2003 and FORTRAN 2008 features, the programs written in FORTRAN 90/95 will be more stable than the ones written in FORTRAN 2003 and FORTRAN 2008.

Based on all these considerations, we decide to choose FORTRAN 95 as the standard version of our generated FORTRAN code. However, with the improvement of most FORTRAN compilers in the future, we may pick a more recent version of FORTRAN as the version of our generated FORTRAN code, and we may use the new features in those FORTRAN versions to support more features in MATLAB, for instance, using `DO CONCURRENT` construct to execute for loop iterations in parallel, or accessing to command line arguments to support data visualization using a third-party software, like *gnuplot*.

5.1.2 Potential Problems

The original intention of designing MATLAB is to make this language a kind of dynamic version of FORTRAN, but with more than thirty years evolvement of the language and its

dynamic language nature, the syntax of MATLAB becomes quite different from the syntax of FORTRAN and even some syntax or features looks quite wild from the point of view of a FORTRAN programmer. For example, in FORTRAN, the number of the indices must be equal to the rank of the accessed array; while in MATLAB, as we introduced in Subsection 4.3.1, the number of the indices can be less than, equal to or greater than the rank of the accessed array. We have to consider how to transform the various kinds of array indexing in MATLAB to the more rigorous way in FORTRAN. Built-in functions in MATLAB are overloaded, which means we should also provide the similar features in the generated FORTRAN code. Since MATLAB is a dynamically-typed programming language, variables can be reused referring to different-type values or even different-shape arrays. But in FORTRAN, variables are statically declared only once and cannot be changed to another type within their scopes in the program. Moreover, removing features noted in FORTRAN 90 as "obsolescent" from FORTRAN 95 makes the language more concise and cleaner, but this leads to some new problems when translating MATLAB to FORTRAN. For example, before FORTRAN 95, the compiler allows implicit type coercions, like `DO` statements can use `REAL` or `DOUBLE PRECISION` variables in range expressions which actually expect `INTEGER` variables. This implicit type coercion is an important feature in weakly-typed languages, like MATLAB. After removing this feature from FORTRAN 95, the compiler requires us to explicitly cast `REAL` and `DOUBLE PRECISION` variables to `INTEGER` variables in the range expressions. These are just some of potential problems we may encounter when we transform MATLAB to FORTRAN. We hope that this will give the reader some idea of the problems to be solved. In the remaining of this chapter, we start with some basic and straightforward transformations and then go to handle more advanced problems like we mentioned above.

5.2 Basic Transformations

In this section, we list some basic and straightforward transformations from MATLAB to FORTRAN 95, like mapping types and generating variable declarations in Subsection 5.2.1 and 5.2.2, mapping built-in operators or functions in Subsection 5.2.3, transforming control flow statements in Subsection 5.2.4, and mapping user-defined functions in Subsection 5.2.5.

5.2.1 Types

In Table 5.1, the left column is the primitive data types in MATLAB, and the right column is the corresponding intrinsic types in FORTRAN.

Table 5.1 Mapping MATLAB types to FORTRAN

Primitive Data Types in MATLAB	**Types in FORTRAN**
double	DOUBLE PRECISION
single	REAL
int8	INTEGER(KIND=1)
int16	INTEGER(KIND=2)
int32	INTEGER(KIND=4)
int64	INTEGER(KIND=8)
char	CHARACTER
logical	LOGICAL
complex	COMPLEX

Besides these primitive data types, the MC2FOR compiler also supports `cell arrays` in MATLAB. We use `derived data type`[2] in FORTRAN to map `cell arrays` in MATLAB.

5.2.2 Variable Declarations

In the generated FORTRAN code, we translate MATLAB scalar variables to scalar variables with the corresponding types and MATLAB non-scalar array variables to multidimensional arrays. If the shape of an array variable is not a constant shape, we declare this variable as an `ALLOCATABLE` array variable in FORTRAN. In Listing 5.1, there are some FORTRAN code snippets to illustrate variable declarations in FORTRAN. In the code snippets, the variable `var_1` is used to map an integer scalar variable in MATLAB, the variable `var_2` is used to map a double-type varied-shape 2-dimensional array variable, and the variable `var_3` is used to map a single-type 3-by-4-sized array variable.

[2] Also known as union type and similar to the struct in C/C++.

```
INTEGER(KIND=4) :: var_1
DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: var_2
REAL, DIMENSION(3,4) :: var_3
```

Listing 5.1 Variable declaration in FORTRAN

5.2.3 Built-in Functions

In Table 5.2, 5.3, 5.4 and 5.5, we list some mappings of arithmetic operators, relational operators, logical operators and commonly used mathematical built-ins from MATLAB to FORTRAN. Note that the left division (`.\`) in MATLAB can be supported by swapping the operands of the operator and replacing the operator with the right division operator (./), and the colon operator in MATLAB can be supported by using an implied `DO` loop in an array constructor in FORTRAN. For example, the expression `var = 1 : 10` in MATLAB will be converted to `var = (/I, I=1, 10/)` in FORTRAN. Some of the arithmetic operators, like complex conjugate transpose, matrix right and left division, and matrix power do not have directly-mapped operators in FORTRAN. The reason we list them in the table is that they are commonly used in MATLAB programs and we must provide a valid solution to handle them if we want to convert MATLAB programs to FORTRAN. Moreover, since there are only a small subset of MATLAB built-ins can be directly mapped in FORTRAN comparing to the large amount of MATLAB built-ins, we have to make sure that the solution is also available to handle other built-ins with no direct mappings.

For the built-in function in MATLAB without a directly mapped intrinsic function in FORTRAN, we leave the built-in function's name in the generated FORTRAN code as the same as it is in MATLAB, and then implement a separate module containing a FORTRAN function with the same function signature doing the same work as it does in MATLAB. For example, in Figure 5.1, there is a MATLAB function computing linear equations using the built-in operaotr \, whose corresponding built-in function is `mldivide` and which is also in Table 5.2. The generated FORTRAN code for this example is in Figure 5.2 and Figure 5.3. Since there is no direct FORTRAN intrinsic function to map \ operator (`mldivide` function) in MATLAB, we provide a module which contains a user-defined FORTRAN function with

Table 5.2 Mapping MATLAB arithmetic operators to FORTRAN

MATLAB **arithmetic operators**	FORTRAN **arithmetic operators**
+ (addition or unary plus)	+
- (subtraction or unary minus)	-
.* (multiplication)	*
./ (right division)	/
. \ (left division)	swap operands and use right division
: (colon operator)	use implied DO loop in array constructors
. ^ (power)	**
.' (transpose)	TRANSPOSE
' (complex conjugate transpose)	no direct mapping
* (matrix multiplication)	MATMUL
/ (matrix right division)	no direct mapping
\ (matrix left division)	no direct mapping
^ (matrix power)	no direct mapping

Table 5.3 Mapping MATLAB relational operators to FORTRAN

MATLAB **relational operators**	FORTRAN **relational operators**
< (less than)	.LT.
<= (less than or equal to)	.LE.
> (greater than)	.GT.
>= (greater than or equal to)	.GE.
== (equal to)	.EQ.
~= (not equal to)	.NE.

the same function signature as the `mldivide` function and this user-defined function will do the same work as the function `mldivide` does in MATLAB. For this example, since there is a same-functionality FORTRAN subroutine, `DGESV`[3], in LAPACK library, we implement this user-defined function as a wrapper function by calling the FORTRAN subroutine `DGESV` in LAPACK library. Note that not all the MATLAB built-in functions can find a corresponding FORTRAN subroutine in LAPACK library, like `ones` and `zeros`, so sometimes we have to implement user-defined functions by ourselves.

[3]LAPACK subroutines solving the system of linear equations for long-precision real.

Table 5.4 Mapping MATLAB logical operators to FORTRAN

MATLAB logical operators	**FORTRAN logical operators**
`&` and `&&`	.AND.
`|` and `||`	.OR.
`~`	.NOT.
xor	.XOR.
any	ANY
all	ALL
bitand	IAND
bitor	IOR
bitcmp	NOT
bitxor	IEOR

```
function linear()

A = [3.1, 1.3, -5.7; 1.0, -6.9, 5.8; 3.4 7.2 -8.8];
B = [-1.3; -0.1; 1.8];
X = A \ B;

end
```

Figure 5.1 MATLAB code example of using `mldivide` built-in function

In summary, to support a built-in function in MATLAB without a directly-mapped intrinsic FORTRAN function, instead of inlining the same-functionality FORTRAN code into the generated program, the MC2FOR compiler will leave a "hole" with the same function signature as the function in MATLAB in the transformed code and requires someone to fill up the "hole" by implementing a user-defined FORTRAN function with the same signature and doing the same job as the function in MATLAB.[4] Furthermore, if another user has a better implementation for that "hole", the only thing the user need to do is removing the old FORTRAN function for that "hole" and recompiling the programs with the new one. By using this solution, whenever there comes a new MATLAB built-in function, the code generation will leave a "hole" for that function and requires a separate FORTRAN mod-

[4]We have already implemented some FORTRAN functions to map some built-in functions in MATLAB, and put them in a user-define FORTRAN library `libmc2for` which is shipped with the compiler.

Table 5.5 Directly mapping MATLAB commonly used mathematical built-ins to FORTRAN

MATLAB **built-ins**	**FORTRAN intrinsic functions**
sum	SUM
ceil	CEILING
floor	FLOOR
mod	MODULO
rem	MOD
round	NINT
fix	INT
sin	SIN
asin	NOT
sinh	SINH
cos	COS
acos	ACOS
cosh	COSH
tan	TAN
atan	ATAN
tanh	TANH
exp	EXP
log	LOG
log10	LOG10
sqrt	SQRT
abs	ABS
conj	CONJG
min	MIN
max	MAX
numel	SIZE
size	SHAPE

```
PROGRAM linear
USE mod_mldivide
IMPLICIT NONE
DOUBLE PRECISION, DIMENSION(3,3) :: A
DOUBLE PRECISION, DIMENSION(3,1) :: B,X

A(1,:) = [3.1, 1.3, -5.7];
A(2,:) = [1.0, -6.9, 5.8];
A(3,:) = [3.4, 7.2, -8.8];

B(1,1) = -1.3;
B(2,1) = -0.1;
B(3,1) = 1.8;

X = mldivide(A,B);

END PROGRAM
```

Figure 5.2 Generated FORTRAN code for the code example in Figure 5.1

```
MODULE mod_mldivide

CONTAINS

  FUNCTION mldivide(A,B)
  IMPLICIT NONE
  DOUBLE PRECISION, DIMENSION(:,:) :: A
  DOUBLE PRECISION, DIMENSION(:,:) :: B
  DOUBLE PRECISION, DIMENSION(SIZE(B,1),SIZE(B,2)) :: mldivide
  INTEGER, DIMENSION(SIZE(B)) :: pivot
  INTEGER :: ok

  CALL DGESV(SIZE(A,1), SIZE(B,2), A, SIZE(A,1), pivot, B, SIZE(B,1),ok)
  mldivide = B;

  END FUNCTION mldivide

END MODULE mod_mldivide
```

Figure 5.3 FORTRAN module for mapping MATLAB built-in function `mldivide`

ule to fulfill that "hole", which means that we don't need to change any code in the code generation framework to handle a new MATLAB built-in function.

MATLAB supports function overloading by argument list, i.e., we can call multiple functions with the same name but a different number of arguments, or arguments of different types. For example, we can pass any number of indices to the built-in function `ones` as in Figure 5.4. To map this features from MATLAB to FORTRAN, we use the `INTERFACE` construct in FORTRAN. The generated FORTRAN code for the example function in Figure 5.4 is in Figure 5.5 and 5.6. In this way, the generated FORTRAN code will be quite similar to the original input MATLAB code.

```
function test_ones()

A = ones(3);
B = ones(3,4);
C = ones(3,4,5);

end
```

Figure 5.4 MATLAB code example to illustrate function overloading

```
PROGRAM test_ones
USE mod_ones
IMPLICIT NONE
DOUBLE PRECISION, DIMENSION(3,3) :: A
DOUBLE PRECISION, DIMENSION(3,4) :: B
DOUBLE PRECISION, DIMENSION(3,4,5) :: C

A = ones(3);
B = ones(3,4);
C = ones(3,4,5);

END PROGRAM
```

Figure 5.5 Generated FORTRAN code for the code example in Figure 5.4

```
MODULE mod_ones
INTERFACE ones
MODULE PROCEDURE ones_1, ones_2, ones_3, ones_i ! may be more
END INTERFACE ones

CONTAINS

FUNCTION ones_1(x)
IMPLICIT NONE
DOUBLE PRECISION, INTENT(IN) :: x
DOUBLE PRECISION, DIMENSION(INT(x),INT(x)) :: ones_1
ones_1 = 1.0
END FUNCTION ones_1

FUNCTION ones_2(x,y)
IMPLICIT NONE
DOUBLE PRECISION, INTENT(IN) :: x, y
DOUBLE PRECISION, DIMENSION(INT(x),INT(y)) :: ones_2
ones_2 = 1.0;
END FUNCTION ones_2

FUNCTION ones_3(x,y,z)
IMPLICIT NONE
DOUBLE PRECISION, INTENT(IN) :: x, y, z
DOUBLE PRECISION, DIMENSION(INT(x),INT(y),INT(z)) :: ones_3
ones_3 = 1.0;
END FUNCTION ones_3

END MODULE mod_ones
```

Figure 5.6 FORTRAN module ones supporting function overloading

5.2.4 Control Flow Statements

The mappings of if-else, for loop and while loop constructs from MATLAB to FORTRAN are kind of straightforward. We list them in Figure 5.7, 5.8 and 5.9, respectively. Although in computer science, subroutines[5] are also considered as one kind of flow control statements, we leave the transformation of the subroutines to the Subsection 5.2.5.

[5]The terminology for subroutines varies; they may be routines, procedures, functions or methods (especially if they belong to classes or type classes). In MATLAB, the subroutines are the user-defined functions.

```
if logical_expression
  statements
[elseif logical_expression
  statements]
else
  statements
end
```

```
IF (logical_expression) THEN
  statements
[ELSE IF (logical_expression) THEN
  statements]
ELSE
  statements
END IF
```

Figure 5.7 if constructs in MATLAB (left) and FORTRAN (right)

```
for var = start[ : step] : stop
  statements
end
```

```
DO var = start, stop[, step]
  statements
END DO
```

- In FORTRAN (right block), `var` should be an integer variable;
- `start` is the initial value `var` is given;
- `stop` is the final value;
- and `step` is the increment by which `var` is changed. If it's omitted, unity is assumed.

Figure 5.8 for loop constructs in MATLAB (left) and FORTRAN (right)

```
while logical_expression
  statements
end
```

```
DO WHILE (logical_expression)
  statements
END DO
```

Figure 5.9 while loop constructs in MATLAB (left) and FORTRAN (right)

5.2.5 User-defined Functions

For now, our toolkit only accepts MATLAB function files, no script files, as the input to the front end. Functions in MATLAB can return (1) no value, (2) one value, or (3) multiple values. Generally speaking, if a user-defined function file has only one return value, we map it to the user-defined function file in FORTRAN; and if a user-defined function file returns no value or multiple values, we map it to the user-defined subroutine file in FORTRAN.

Since in MATLAB every function file can be the main entry point program file, we map the main entry point function file in MATLAB to the main program file in FORTRAN. If the

main entry point function file has input values, we use the FORTRAN 95 subroutine `GETARG` to get input arguments for the main programs. At the left hand side of Figure 5.10, there is a MATLAB entry point function with one integer input argument. On the right hand side, it is the generated FORTRAN code. Inside the FORTRAN code, we use the FORTRAN 95 intrinsic subroutine `GETARG` to read one input from the command line and assign the value of the input to the corresponding variable.

```
% MATLAB entry point function
function entry_point(arg_1)
...
end
```

```
! Fortran code generation
PROGRAM entry_point
! declaration section
INTEGER :: arg_1
INTEGER :: int_tmpvar
CHARACTER(10) :: arg_buffer
...
! main program starts
int_tmpvar = 0;
arg_buffer = '0000000000';
DO int_tmpvar = 1 , IARGC()
  CALL GETARG(int_tmpvar,
      arg_buffer);
  IF ((int_tmpvar == 1)) THEN
    READ(arg_buffer, *) arg_1
  END IF
END DO
...
END PROGRAM entry_point
```

Figure 5.10 Translating entry point function from MATLAB to FORTRAN

5.3 Advanced Problems in Mapping Types

MATLAB is a weakly-typed programming language because types are implicitly converted. That's why you can use a `double` value subscript to index an array in MATLAB, even though MATLAB requires that subscript indices must either be real positive integers or logicals. But FORTRAN is a strongly-typed programming language, types should be converted explicitly. For example, in FORTRAN, the array indices must be in the `INTEGER` type, and in order to

achieve more efficiency, DO statements cannot use REAL and DOUBLE PRECISION variables since FORTRAN 95.

5.3.1 For Subscripts in Array Indexing

Since FORTRAN doesn't support type coercion and requires that the indices of the array accessing must be in the type of INTEGER, the MC2FOR compiler uses the FORTRAN intrinsic function INT to explicitly convert the types of the indices of the accessed array to INTEGER in the generated code. Here is an example in Figure 5.11.

```
% MATLAB code snippet
arr = [1,2,3,4,5,6];
arr(3.0) = 6;
```

```
! converted FORTRAN code snippet
arr = [1,2,3,4,5,6];
arr(INT(3.0)) = 6;
```

Figure 5.11 Subscript in MATLAB (left) and FORTRAN (right)

5.3.2 For Loop Range Expressions

In MATLAB, recall the for loop construct in Figure 5.8. The expression after the keyword **for**, which is `var = start[ : step] : stop`, is called the range expression. The variable `var` can be in the `double` type and the values of `start`, `step` and `stop` can also be in the `double` type. While after FORTRAN 95, the `var`, `start`, `stop` and `step` must be in the type of INTEGER. So we have to transform the loop constructs in MATLAB one more step than directly transformed in Figure 5.8. The final version of transforming for loops in MATLAB to FORTRAN is in Figure 5.12.

5.3.3 Assigning Multiple Types to the Same Variable

We know that in FORTRAN and other statically-typed programming languages, after the declaration of a variable, we can only assign values of the declared type to the variable. While in MATLAB, a same variable can be assigned with values of different types. One solution to map this feature in MATLAB to FORTRAN is that we use static single assignment

```
for var = start[ : step] : stop
 statements
end
```

(a) For loop construct in MATLAB

```
! if the range expression has step
var = start;
step_new =
    INT((stop-start)/step+1);
DO var_new = 1, step_new
 statements
 var = var + step;
END DO
```

```
! if the range expression has no
    step, unity is assumed
var = start;
step_new = INT((stop-start+1);
DO var_new = 1, step_new
 statements
 var = var + step;
END DO
```

(b) Transformed for loop construct in FORTRAN

Figure 5.12 Transformation of for loop constructs in MATLAB to FORTRAN

form (SSA form) to rename the variable with multiple definitions to different variables, which ensures that each variable is assigned exactly once. But this solution will be very expensive and introduce extra overhead. For example, if all the definitions of the variable are in the same type, for the code generation, we don't want to split this variable according to different use-def chains. In our transformation framework, we choose to rename variables only when it's necessary.

The condition to trigger renaming a variable is that this variable has different type informations in different DU-UD webs. If the variable has different type in a certain DU-UD web, we transform this variable to a derived-data type variable in FORTRAN. In the generated derived-data type, there are several data fields which exactly match those possible types of the variable in the MATLAB program. Besides those data fields, there is a character string data field, named token, which serves to indicate which field is currently active. In Figure 5.13, the left hand side is a piece of code example in MATLAB, and the right hand side is the corresponding generated FORTRAN code.

In Figure 5.13, you may have already noticed that since `a` has different types, the generated FORTRAN code for the operations involving `a` also has different possibilities. In other words, to translate MATLAB statement, `b = a + 3;`, at line 9, we inlined run-time checking code and different statements from line 17 to line 23 in the generated FORTRAN

```
% MATLAB code snippet
if ...
  a = int8(15);
  ...
else
  a = 15.0;
  ...
end
b = a + 3;
...
```

```
! Fortran code snippet
TYPE a
  CHARACTER(10) token
  INTEGER(KIND=1) val_int8
  DOUBLE   val_double
END TYPE a
! generated fortran code
IF ... THEN
  a%val_int8 = int8(15);
  a%token = 'int8';
  ...
ELSE
  a%val_double = 15.0;
  a%token = 'double';
  ...
END IF
IF (a%token .EQ. 'int8') THEN
  b%val_int8 = a%val_int8 + 3;
  b%token = 'int8';
ELSE
  b%val_double = a%val_double + 3;
  b%token = 'double';
END IF
...
```

Figure 5.13 Using derived data type in FORTRAN to map inconvertible types of the same variable in MATLAB

code.

One more thing to notice, actually the ideal type or data structure to map this variable in MATLAB should be something similar to the union type in C/C++, because, at any time, there can be only one type for that variable. But there is no such a data type in FORTRAN, and the derived-data type in FORTRAN is more likely to the struct type in C/C++, which means, comparing with the union type, we may sacrifice some extra storage to support this feature in MATLAB.

5.4 Array Indexing Transformation

As we mentioned in Subsection 4.3.1, in MATLAB the number of the subscripts in an array indexing can be less than, equal to or greater than the rank of the indexed array. But for FORTRAN, it requires that the number of the indices in an array indexing must be equal to the rank of the indexed array. So we need to figure out a strategy to transform the array indexing in MATLAB to a more rigorous way of array indexing in FORTRAN. In our MC2FOR compiler, we designed a strategy named *rigorous indexing transformation* to transform the array indexing in MATLAB to a more rigorous way of array indexing in FORTRAN. Strictly speaking, the rigorous indexing transformation is only for the case where the number of the subscripts in an array indexing is less than the rank of the accessed array, or we can say the linear indexing case, because if the number of the subscripts in an array indexing is equal to the rank of the accessed array, the subscript list will be the same in both MATLAB and FORTRAN, and if the number of the subscripts in an array indexing is greater than the rank of the accessed array, after inlining the array shape resizing code before the array indexing statement, the indexing will also be the same in both languages, since the number of the subscripts in the array indexing now is equal to the rank of the reshaped array.

5.4.1 For Array Get Statements

For the linear indexing transformation, in the generated code, we implement some user-defined FORTRAN functions to proceed the indexing work and return the same result as the indexing does in MATLAB. The names of the FORTRAN functions are based on a naming convention. For the array indexing in the array get statement, the name of the function starts with `ARRAY_GET`. Then we add the rank of the accessed array to the end of the name. After that, for each index in the index list, if the index is a colon notation, we add `C` to the end of the name of the function; if the index has vector value, we add `V` to the end of the name of the function; and if the index has scalar value, we add `S` to the end of the name of the function. When there is no index left in the index list, the name of the transformation function is done. For example, if the accessed array `arr` is a `3`-dimensional

array and the index list is `(scalar_var,vector_var)`, the transformation function name will be `ARRAY_GET3SV`. The input argument list of a transformation function starts with the name of the accessed array. Then for each index in the index list, we add the index itself to the end of the list. When there is no index left in the index list, the input argument list of the transformation function is done. Continue with the above example, the mapping of `var = arr(scalar_var,vector_var)` in MATLAB will be `var = ARRAY_GET3SV(arr, scalar_var, vector_var)` in the generated FORTRAN code.

Moreover, the FORTRAN function `ARRAY_GET3SV` can be used to process all the cases where the array's rank is `3`, the first index has a scalar value and the second index has a vector value. The implementation of the function `ARRAY_GET3SV` in our MC2FOR compiler is in Figure 5.14.

```
FUNCTION ARRAY_GET3SV(array, idx_1, idx_2)
IMPLICIT NONE
DOUBLE PRECISION, DIMENSION(:,:,:) :: array
INTEGER :: d2, d3
DOUBLE PRECISION :: idx_1
DOUBLE PRECISION, DIMENSION(:,:) :: idx_2
DOUBLE PRECISION, DIMENSION(1,SIZE(idx_2)) :: ARRAY_GET3SV
INTEGER :: i, idx_new_1, idx_new_2

d2 = size(array,2);
d3 = size(array,3);
idx_new_2 = 1;
DO i = INT(idx_2(1,1)), INT(idx_2(1,SIZE(idx_2)))
  idx_new_1 = i - d2 * (idx_new_2-1);
  IF (idx_new_1 > d2) THEN
    idx_new_2 = idx_new_2 + 1;
    idx_new_1 = idx_new_1 - d2;
  END IF
  ARRAY_GET3SV(1,i) = array(INT(idx_1),idx_new_1,idx_new_2);
END DO
END FUNCTION ARRAY_GET3SV
```

Figure 5.14 The function `ARRAY_GET3SV`

5.4.2 For Array Set Statements

The array set statements assign new values to some of the elements in the original arrays. Instead of using functions, we implemented user-defined FORTRAN subroutines to perform both the array indexing and assigning work. The naming convention of the transformation subroutine for the array set statements is similar to the one for the array get statements with a small update. The name starts with `ARRAY_SET`, then followed by the rank of the accessed array. After that, for each index in the index list, if the index is a colon notation, we add `C` to the name of the function; if the index has vector value, we add `V` to the name of the function; and if the index has scalar value, we add `S` to the name of the function. When there is no index left in the index list, we add the rank of the right hand side value or variable to the end of the name of the function, then the name of the transformation subroutine is done. For example, if the accessed array is a 3-dimensional array, the index list is `(scalar_var,vector_var)` and the right hand side variable is a vector variable, the transformation subroutine name will be `ARRAY_SET3SV2`. The input argument list of the transformation subroutine starts with the name of the accessed array. Then for each index in the index list, we add the index itself to the end of the list. When there is no index left in the index list, we add the name of the right hand side variable or value to the end of the list, then the input argument list of the subroutine is done. Back to the above example, the statement `arr(scalar_var,vec_var1)= value` in MATLAB will be mapped to `CALL ARRAY_GET3SV2(arr, scalar_var, vec_var1, value)` in the generated FORTRAN code.

The same as the transformation functions for the array get statements, the transformation subroutines for the array set statements are also reusable. The subroutine `ARRAY_SET3SV2` can be used to all the cases where the accessed array is 3-dimensional, the first index has a scalar value, the second index has a vector value and the rank of the right hand side is 2-dimensional. The implementation of the subroutine `ARRAY_SET3SV2` in our MC2FOR is in Figure 5.15.

```
SUBROUTINE ARRAY_SET3SV2(array, idx_1, idx_2, value)
IMPLICIT NONE
DOUBLE PRECISION, DIMENSION(:,:,:) :: array
DOUBLE PRECISION :: idx_1
DOUBLE PRECISION, DIMENSION(:,:) :: idx_2
DOUBLE PRECISION, DIMENSION(:,:) :: value
INTEGER :: d2, d3
INTEGER :: i, idx_new_1, idx_new_2

d2 = size(array,2);
d3 = size(array,3);
idx_new_2 = 1;
DO i = INT(idx_2(1,1)), INT(idx_2(1,SIZE(idx_2)))
  idx_new_1 = i - d2 * (idx_new_2-1);
  IF (idx_new_1 > d2) THEN
    idx_new_2 = idx_new_2 + 1;
    idx_new_1 = idx_new_1 - d2;
  END IF
  array(INT(idx_1),idx_new_1,idx_new_2) = value(1,i);
END DO
END SUBROUTINE ARRAY_SET3SV2
```

Figure 5.15 The subroutine `ARRAY_SET3SV2`

5.4.3 Shortcut Linear Indexing Transformation

For some cases where both the shape of the accessed array and the subscript(s) are exactly known, the compiler can proceed a linear indexing transformation in compile-time instead of inlining a special-named function call as we mentioned in previous two subsections, it will use the transformed subscripts as the new subscripts. For example, in MATLAB, programmers always prefer to use one subscript to access a column or a row vector. While in FORTRAN, since we declare all the vectors as two dimensional arrays, we have to transform this linear indexing in MATLAB to regular array indexing in FORTRAN. The linear indexing transformation for this case is simple. We only need to add "1" to the first place or the second place of the index list based on whether the array is a row vector or a column vector. The more complicated cases are where the accessed arrays are with two or more dimensions. For example, the array `A` is with the shape of 3-by-4, the linear array indexing `A(5)` will be transformed to `A(2,2)` in the generated FORTRAN code.

5.5 Run-time Array Bounds Checking and Variable Resizing

For the array get and set statements which we cannot applied static array bounds checking in the compile-time, the MC2FOR compiler will insert some run-time array bounds checking code and shape resizing code before the array indexing expressions in the generated FORTRAN code.

5.5.1 For Array Get statements

FORTRAN doesn't have array bounds checking for the array indexing.[6] If the index exceeds the upper bound of the corresponding dimension of the accessed array, the compiler will not raise an exception and return whatever value on that address. But MATLAB has built-in array bounds checks, so for array get statements, we add run-time array bounds checking code before the array get statements. If the number of the indices in an array indexing equals the rank of the accessed array, our MC2FOR compiler will add the array bounds check code before the array get statement, and the code is simply comparing each index in the index list with the size of corresponding dimensions. If any index exceeds the upper bound of the accessed dimension of the array, the program terminates and raise an out-of-bound array indexing exception. For example, in Figure 5.16, (b) is the converted FORTRAN code for the array get statement example in (a).

For the case where the number of the indices in an array indexing is less than the rank of the accessed array, recall that we use a FORTRAN transformation function to map this kind of array get statement in MATLAB, our MC2FOR compile will put the run-time array bounds checking code inside the transformation function.

For the case where the number of the indices in an array indexing is greater than the rank of the accessed array, unless the extra index has the scalar value of 1 or all the extra indices have the scalar values of 1, our MC2FOR compiler will throw an out-of-bound indexing exception.

[6] Although third-party FORTRAN compilers may support some options to enable the generation of run-time checks, for example, the **-fcheck='bounds'** options of GFortran.

```
% MATLAB code snippet
target_var = array(index_list);
```

(a) Array get statement example in MATLAB

```
! converted FORTRAN code snippet
array_d1 = SIZE(array,1);
array_d2 = SIZE(array,2);
... ! if array has more dimensions.
IF first_index > array_d1 or second_index > array_d2 [or ...] THEN
  STOP "ABNORMALY: out-of-bound indexing exception."
END IF
target_var = array(index_list);
```

(b) Converted FORTRAN code for the code snippet in Figure 5.16 (a)

Figure 5.16 Run-time array bounds checking code for array get statement

5.5.2 For Array Set statements

The run-time array bounds checking for the array get statements is a little bit more complicated than the one for the array set statements discussed in Subsection 5.5.1, because the accessed array may be grown by the out-of-bound array indexing in MATLAB. To map this dynamic feature in MATLAB to the generated FORTRAN code, besides the run-time array bounds checking code, our MC2FOR compiler also inlines the run-time deallocate and allocate code for the array set statements, which is in case of the accessed array is grown by the out-of-bound array indexing. The steps to do run-time array bounds checking and variable reshape are listed in order as below.

1. Back up the current size of the accessed array;

2. Compare each index in the index list with the size of the corresponding dimensions of the accessed array, if the out-of-bound indexing occurs, go to the following step, if not, go to the end of the algorithm;

3. Back up the accessed array;

4. Deallocate the array and reallocate the array with the new shape;

5. Copy the backup value back to the reshaped array, the end.

In Figure 5.17, (a) is an array set statement code snippet in MATLAB, and (b) is the generated FORTRAN code with the run-time array bounds checking code for the code snippet in (b).

```
array(p, 5) = value;
```

(a) Array set statement code snippet in MATLAB

```
! step 1, back up the current size of the accessed array
array_d1 = SIZE(array, 1);
array_d2 = SIZE(array, 2);
! step 2, compare the indices with the dimensions
IF ((p > array_d1) .OR. (5 > array_d2)) THEN
  ! step 3, back up the accessed array with a temporary array
  IF (ALLOCATED(array_bk)) THEN
     DEALLOCATE(array_bk);
   END IF
   ALLOCATE(array_bk(array_d1, array_d2);
   array_bk = array;
   ! step 4, deallocate and reallocate the accessed array
   DEALLOCATE(array);
   array_d1max = MAX(p, array_d1);
   array_d2max = MAX(5, array_d2);
   ALLOCATE(array(array_d1max, array_d2max));
   ! step 5, copy the backup value back to the reshaped array
   array(1:array_d1, 1:array_d2) = array_bk(1:array_d1, 1:array_d2);
END IF
array(INT(p), 5) = value;
```

(b) Converted FORTRAN code snippet for the code snippet in Figure 5.17 (a)

Figure 5.17 Run-time array bounds checking and variable reshape code for array set statement

5.5.3 For Assignment Statements

When assigning to a variable, the right hand side can be summarized into three categories: (1) another variable, (2) a directly-mapped built-in function or operator, or (3) a not-directly-mapped built-in function. The first case is a simple array assignment, if both the array variables are not allocatable arrays, they should be have the same shape; if the left hand side array variable is an allocatable array, it will be first allocated with the same shape of the right hand side array variable, and then each element of the right hand side array will

be assigned to the corresponding element of the left hand side array. The second case is similar to the first case. For the third case, the allocation is done by the user-defined FORTRAN function. Recall the MATLAB built-in function `ones`. When using `ones` to assign to an allocatable array in FORTRAN, the allocation is achieved in the user-defined module `mod_ones`, as illustrated in Figure 5.18 (a) and (b). So based on above reasons, there is no need to inline any extra code to perform run-time array bounds checking or shape resizing for the assignment statements.

```
PROGRAM test_ones
USE mod_ones
IMPLICIT NONE
DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: A

A = ones(3,4);

END PROGRAM
```

(a) FORTRAN program to allocate storage using a user-defined function

```
MODULE mod_ones
INTERFACE ones
MODULE PROCEDURE ones_1, ones_2, ones_3, ones_i ! may be more
END INTERFACE ones

CONTAINS
! omit some lines of code
FUNCTION ones_2(x,y)
IMPLICIT NONE
DOUBLE PRECISION, INTENT(IN) :: x, y
DOUBLE PRECISION, DIMENSION(INT(x),INT(y)) :: ones_2
ones_2 = 1.0;
END FUNCTION ones_2
! omit some lines of code
END MODULE mod_ones
```

(b) FORTRAN module `mod_ones` to allocate storage for allocatable arrays

Figure 5.18 Illustration of allocating allocatable arrays in user-defined functions

Chapter 6
Experimental Results

In this chapter, we demonstrate some experiments on a set of 20 MATLAB benchmarks to evaluate the performance of our MC2FOR compiler.

First we compare the running time of (1) the original benchmarks under MATLAB 2013a, which is the standard software and environment to run the MATLAB code, and (2) the GFortran-compiled FORTRAN programs generated by MC2FOR for the benchmarks. We also examine the running time of (3) the benchmarks under Octave [oct], which is an open source software and environment for numerical computations and mostly compatible with MATLAB, and (4) the C programs generated by MATLAB Coder [Mata] for reference. In order to get a measurable execution time, for each benchmark we adjusted a scale number to control the problem size, including the number of iterations and the size of arrays, to make the benchmark to run approximately 20 seconds under MATLAB. Then we use the same scale number to run the same benchmark under Octave, FORTRAN and MATLAB Coder.

Besides the running time, another factor to evaluate the performance of our compiler is the size of the code generated by MC2FOR for the input MATLAB benchmarks. So we list the physical lines of code (LOC) of the original MATLAB benchmarks and the generated FORTRAN code by MC2FOR. We also list the LOC of the generated C code by MATLAB Coder for reference.

In the following sections of this chapter, we first provide a brief description of the 20 MATLAB benchmarks in Section 6.1, then show the experimental results in Section 6.2 and

give a detailed discussion of the results in Section 6.3, and finally sum up the whole chapter in Section 6.4.

6.1 Description of the Benchmarks

The set of benchmarks for the experiments is acquired from a variety of sources, most of them come from related projects, like FALCON and OTTER projects, Chalmers University of Technology[1], "The MathWorks' Central File Exchange"[2] and ACM CALGO library[3]. A brief description of the benchmarks is given here.

- *adpt* is an implementation to find the adaptive quadrature using Simpson's rule. This benchmark features an array whose size cannot be predicted before compilation. [Numerical Methods]
- *arsm* is a simulation of AR process. This benchmarks involves several kinds of array constructions, like from MATLAB built-in functions `randn`, `ones`, `eye`, `zeros` and `horzcat`, and most of the operations are focused on arrays. [ACM CALGO library]
- *bbai* is an implementation of a well known linear algebra algorithm, the Babai nearest plane algorithm. This algorithm is an approximation to solve the closest vector problem. This benchmark has simple array read and scalar operations. [McLab]
- *bubl* is an implementation of the standard bubble sort algorithm. This benchmark contains nested loops and consists of many array read and write operations. [McLab]
- *capr* is an implementation of computing the capacitance of a transmission line using finite difference and Gauss-Seidel method. It's a loop-based program that involves basic scalar operations on two small-sized arrays. [Chalmers EEK 170]
- *clos* is an implementation to calculate the transitive closure of a directed graph. It contains matrix multiplication operations between two 450-by-450 arrays. [Otter]

[1] http://www.elmagn.chalmers.se/courses/CEM/
[2] http://www.mathworks.com/matlabcentral/fileexchange
[3] http://calgo.acm.org/

- *crte* is the front part of a simulated maximum likelihood statistical regression. It first creates a random matrix X and optimizes the columns to maximize the minimum euclidean distance between points. This benchmarks features a lot of array read operations and basic scalar computations. [McLab]

- *crni* is an implementation of the Crank-Nicholson solution to the heat equation. This benchmark involves some elementary scalar operations on a 2300-by-2300 array. [Numerical Methods]

- *dich* is an implementation of the Dirichlet solution to Laplace's Equation. It's also a loop-based program which involves basic scalar operations on a small-sized array. [Numerical Methods]

- *diff* is an implementation to calculate the diffraction pattern of monochromatic light throught a transmission grating for two slits. This benchmark also features an array whose size is increased dynamically like the benchmark *adpt*. [Appelbaum (MUC)]

- *edit* is an implementation to find the edit distance between the source string s1 and the target string s2, so it involves computations on strings of characters. [Castro (MUC)]

- *fdtd* is an implementation to apply the Finite Difference Time Domain (FDTD) technique on a hexahedral cavity with conducting walls. This benchmark features array read and write operations on 3-dimensional arrays. [Chalmers EEK 170]

- *fft* is an implementation to compute the discrete fourier transform for complex data. The benchmark has nested loops, basic array read operations and simple scalar operations. [Numerical Recipes]

- *fiff* is an implementation of finite-difference solution to the wave equation. It's a loop-based program which involves basic scalar operation on a 2-dimensional array. [Numerical Methods]

- *lgdr* is an implementation to evaluate the normalized, orthogonormal legendre polynomials, and also the first derivative and second derivative of the polynomials. This

benchmark has simple array read and write operations and basic scalar operations. [ACM CALGO library]

- *mbrt* is an implementation to compute mandelbrot set with specified number elements and number of iterations. This benchmark contains basic scalar operations on complex data. [McLab]
- *nb1d* is an implementation to simulate the gravitational movement of a set of objects. It involves computations on vectors inside nested loops. [Otter]
- *nb3d* is also an implementation to simulate the gravitational movement of a set of objects. This benchmark differs from the benchmark *nb1d* is that this benchmark involves operations on 3-dimensional arrays and there are very complicated high-level array read and write operations, for example, using a 2-dimensional array and a colon notation to index another 2-dimensional array to get a 3-dimensional array. [Otter]
- *scra* is an implementation to produce a reduced-rank approximation to a matrix, which uses the benchmark *spqr* to compute the SPQR factorizations of A and A'. This benchmarks contains basic array read operations and simple scalar operations. [ACM CALGO library]
- *spqr* is an implementation to compute a pivoted semi-QR decomposition of an m-by-n matrix A. This benchmarks contains basic array read operations and simple scalar operations. [ACM CALGO library]

6.2 Experimental Results

All the programs were executed on a machine with Intel(R) Core(TM) i7-3930k CPU @ 3.20GHz x 12 processor and 16 GB memory running GNU/Linux (3.2.0-26-generic #41-Ubuntu). The MATLAB version is R2013a, the GNU Octave version is 3.2.4, the generated FORTRAN code is compiled with the GFortran compiler of GCC version 4.6.3 with level 3

optimization, and the profiler tools are GNU gprof[4] and gcov[5].

In Table 6.1, we list the execution time of different benchmarks under MATLAB, Octave, FORTRAN (with array bounds checking) and MATLAB Coder.

Table 6.1 Performance comparison

Benchmarks	MATLAB	**Octave**		**FORTRAN (checked)**		MATLAB **Coder**	
	(sec)	(sec)	Slowdown	(sec)	Speedup	(sec)	Speedup
adpt	20.25	156.9	7.7	5	4.1	-	-
arsm	20.28	125.9	6.2	0.32	63.4	0.4	50.7
bbai	20.25	102	5.0	0.55	36.8	4.8	4.2
bubl	20.6	5047	245	1.55	13.3	9.6	2.1
capr	20.29	5900	290.8	1.94	10.5	6.1	3.3
clos	20.57	675	32.8	507	0.04	19.6	1.0
clos+	20.57	675	32.8	17.3	1.2	19.6	1.0
crte	20.54	-	-	0.32	64.2	-	-
crni	20.39	3852	188.9	1.9	10.7	2.4*	8.5
dich	20.48	11325	553.0	5.3	3.9	2.7	7.6
dich+	20.48	11325	553.0	2.1	9.8	2.7	7.6
diff	20.16	1152	57.1	4.8	4.2	-	-
edit	20.77	423	20.4	0.21	98.9	0.9*	23.1
fdtd	20.38	26.6	1.3	2.3	8.9	3.5	5.8
fft	18.71	2854.1	152.5	2.73	6.9	8.1	2.3
fiff	20.48	8301	405	0.92	22.3	0.3	68.3
fiff+	20.48	8301	405	0.52	39.4	0.3	68.3
lgdr	20.2	72.6	3.6	0.06	336.7	0.04	505
mbrt	20.21	1381	68.3	3.3	6.1	3.57	5.7
nb1d	20.57	55.8	2.7	1.08	19.0	2.45	8.4
nb3d	20.36	24.1	1.2	39	0.5	61.1	0.3
scra	20.2	76.9	3.8	0.48	42.1	0.61*	33.1
spqr	20.58	88.4	4.3	0.46	44.7	0.67*	30.7

In the column labeled "Benchmarks", for some benchmarks we add a superscript "+" symbol to the name of the benchmark, which means for that benchmark, we made some

[4] http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html
[5] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

manual optimization to the generated FORTRAN program and record the running time under FORTRAN again. The "-" in the columns under Octave for the benchmark *crte* means that this benchmark cannot be run under Octave for the reason of not supporting the MATLAB built-in function **randi**. The "-" in the columns under MATLAB Coder indicates that the corresponding MATLAB benchmarks cannot be compiled or run under MATLAB Coder for some reasons, for example, the dynamically growing arrays. The "*" symbol on the right shoulder of the data in the column of "(sec)" under MATLAB Coder means that the benchmarks can be compiled and run under MATLAB Coder with some conditional modification on the benchmarks, like adding preallocation statements for all the arrays before they are used in the array writes.

From the results in Table 6.1, we can see that for most benchmarks, after converting to FORTRAN by MC2FOR, there is a performance speedup which varies from 1.2 to 337,[6] besides two exceptions, the benchmarks *clos* and *nb3d*. The generated FORTRAN program for *clos* is running about 25 times slower than the benchmark running under MATLAB; the generated FORTRAN program for *nb3d* is running about 2 times slower than the benchmark running in MATLAB. The detailed explanation is given in the following section. The benchmarks running under Octave are significantly slower than under MATLAB which varies from 1.2 to 553. One reasonable explanation is that Octave has no just-in-time (JIT) accelerator and purely interprets the program. Most of the benchmarks can be transformed by MATLAB Coder to C programs and the running time of the C programs is similar to the running time of the generated FORTRAN program. During the experiment, we also found that there are at least two features of MATLAB which are currently not supported by MATLAB Coder. Detailed discussion of the experiment result is given in the following section.

In Table 6.2, we list the LOC of the original MATLAB benchmarks, the generated FORTRAN programs by MC2FOR and the generated C programs by MATLAB Coder. The LOC here also includes the lines of whitespace and comments. First of all, the numbers of the LOC of the generated FORTRAN and C are, as expected, larger than the number of the LOC of the original MATLAB benchmark, since there are extra inlined lines of code for the variable declarations and code to support some dynamic features of MATLAB, like dynamically

[6] We highlight the cells in which the speedup is above 1.

growing arrays[7] and array bounds checking. Because of the similar syntax between MATLAB and FORTRAN and the fact our MC2FOR generate FORTRAN based on a readable IR, for most of the benchmarks, the number of the LOC of the generated FORTRAN is within or around a factor of 2 the size of the number of the LOC of the original MATLAB benchmark. While, the number of the LOC of the C code generated by the Coder is significantly larger than the number of the LOC of the original benchmarks. After going through the generated C code, we found that for each benchmark the Coder always generates some extra files with the name of the entry point function adding the suffix of "_api", "_data", "_emxutil", "_initialize", "_mex" or "_terminate". This may support extra functionality, whereas our code focus on only the benchmarks and generates stand-alone code.

6.3 Analysis of Results

In this section, we provide some detailed discussion on the experimental results in Table 6.1 and 6.2.

6.3.1 MC2FOR and MATLAB Coder vs. MATLAB

For most benchmarks, there are performance speedups after transforming the code from MATLAB to FORTRAN except for the benchmarks *clos* and *nb3d*.

After profiling the benchmark *clos* with gprof, we discovered that the GFortran intrinsic function for matrix multiplication, MATMUL, is not very efficient. In order to validate that this was the problem, we use the option `-fexternal-blas` of GFortran to replace the the call to MATMUL with a call to the subroutine DGEMM in the default FORTRAN BLAS (Basic Linear Algebra Subprograms) library under Ubuntu.

The program using the subroutine DGEMM runs about 7 times **faster** than the program using the intrinsic function MATMUL, but still 3.5 times **slower** than the *clos* benchmark running under MATLAB. A reasonable explanation is that according to one document on the website of Intel[8], at least since MATLAB R2010a, MATLAB uses the Intel MKL BLAS

[7] MATLAB Coder may not support this feature currently.

[8] http://software.intel.com/en-us/articles/using-intel-mkl-with-matlab

Table 6.2 Physical lines of code comparison

Benchmarks	MATLAB	FORTRAN (checked)		MATLAB Coder (headers + sources)	
	(lines)	(lines)	F / M	(lines)	C / M
adpt	169	327	1.9	-	-
arsm	113	181	1.6	387 + 2228	23
bbai	28	67	2.4	228 + 931	41.4
bubl	23	96	4.2	258 + 686	41
capr	206	479	2.3	339 + 1455	8.7
clos	78	157	2.0	256 + 734	12.7
crte	157	679	4.3	-	-
crni	194	237	1.2	285 + 769	5.4
dich	131	161	1.2	220 + 685	6.9
diff	115	167	1.5	-	-
edit	396	720	1.8	326 + 5011	13.5
fdtd	122	197	1.6	271 + 744	8.3
fft	77	289	3.8	279 + 1004	16.7
fiff	105	139	1.3	213 + 572	7.5
lgdr	118	197	1.7	216 + 640	7.3
mbrt	43	98	2.3	278 + 844	26.1
nb1d	166	327	2.0	394 + 2173	15.5
nb3d	141	215	1.5	386 +3590	28.2
scra	201	533	2.7	468 + 2949	17
spqr	190	497	2.6	345 + 2140	13.1

by default, and Intel MKL BLAS may have a better implementation than the one default for Ubuntu. To further proof that this is the key reason, we use the OpenBLAS instead of default BLAS in Ubuntu. The compiled program runs 1.2 times **faster** than the benchmark under MATLAB.

The benchmark *nb3d* features very flexible array read and write operations on arrays of three dimensions. MATLAB may already have some very optimized compiled subroutines to perform those operations. While for MC2FOR, we currently support those flexible array operations by writing user-defined FORTRAN subprograms and putting them in the library **libmc2for**, and they may not be very optimized. This may also be the same reason why the generated C program by MATLAB Coder for this benchmark also runs slower than

the benchmark runs under MATLAB, because the generated C code by MATLAB Coder is compiled by the GNU C compiler on the local machine, while the optimized compiled subroutines in MATLAB may be compiled with better compilers, for example, the Intel Compilers, and with better optimization.

Since MATLAB is closed source, it's really hard to know what exactly happens inside MATLAB, but in one thread on the MATLAB Answers[9], the MathWorks Support Team gave some feedback about one question of "why is my MATLAB Coder generated MEX file slower than my MATLAB function". Based on the feedback, we know that some of MATLAB functions are compiled already, and sometimes they are even multithreaded, so they are highly optimized for the PC. While, the generated C code on the other hand is more of a readable and portable C code and is not optimized for a particular platform, so the slowdown in this case is expected regardless of MEX or standalone compilation.

6.3.2 MC2FOR vs. MATLAB Coder

In our 20 benchmarks, we found that there are two dynamic features of MATLAB can not be supported by current MATLAB Coder. Firstly, the Coder cannot support the benchmarks with dynamically growing arrays, like the benchmarks *adpt* and *diff*; Secondly, the Coder cannot support the benchmarks with unpreallocated arrays, like the benchmarks *crni*, *edit*, *scra* and *spqr*. After adding the statements of array preallocation, these benchmarks then can be transformed by the Coder.

For most of the successfully transformed benchmarks, the running time of the C programs is similar to the running time of the FORTRAN programs transformed by MC2FOR. There are two benchmarks, *dich* and *fiff*, whose MATLAB Coder generated programs run more than two times faster than their MC2FOR generated programs. After reading the generated C code by MATLAB Coder, we found that the Coder does the constant folding optimization and inlines some function files into their calling functions.

In order to validate that this is one possible reason, for the benchmark *dich*, we first profiled the generated FORTRAN program and find there is one expression, at the left hand side of Figure 6.1, whose evaluation result should be constant but seems not be replaced at

[9] http://www.mathworks.com/matlabcentral/answers/102725

runtime and cost a lot of time to evaluate at runtime. Then we manually replaced the expression with its constant value, as in the right of Figure 6.1, and we found that the running time of the generated FORTRAN programs improves from 5.3 seconds to 2.1 seconds and is faster than the C programs generated by the Coder. We did the similar manual optimiza-

```
tol = (10 ** -5);
```

```
tol = 9.9999999999999991E-6;
```

Figure 6.1 Constant value replacement for the power function in FORTRAN

tion for the generated FORTRAN program for the benchmark *fiff* and made its running time improves from 0.92 seconds to 0.52 seconds.

Although now we know that constant folding can help improving the performance of the transformed program, there is a potential limitation for MC2FOR to do this. The MC2FOR is implemented in Java, which means that the constant folding is achieved based on the result computed by Java built-in mathematical methods. One potential concern is that the result may not be the same as it is when the expression is executed in FORTRAN at runtime. While, for MATLAB Coder, since both MATLAB and MATLAB Coder are written in C (or also in C++), the static constant folding result is the same as it is when the expression is executed at runtime. Based on this reason, MC2FOR have to be very careful about the constant folding it performs during the transformation and leave the constant folding work for some complicated expressions to the third-party FORTRAN compilers, like GFortran.

6.3.3 MC2FOR without Checks vs. with Checks

There is a **-nocheck** flag can be passed as an argument to MC2FOR when performing the transformation from MATLAB to FORTRAN. This flag controls whether the compiler will inline the array bounds checking code for the array read operations. In Table 6.3, We list the running time and LOC of the generated FORTRAN code both without and with the array bounds checking.

From the results in Table 6.3, we can see that the extra inlined array bounds checking for array read operations doesn't add significant overhead to the running time of the generated programs. Moreover, for some benchmarks, after adding the checks and compiled

Table 6.3 MC2FOR without and with checks

Benchmarks	Running Time in Second (nocheck \ check)	LOC (nocheck \ check)
adpt	4.9 \ 5	302 \ 327
arsm	0.26 \ 0.32	181 \ 181
bbai	0.55 \ 0.55	67 \ 67
bubl	3.79 \ 1.55	86 \ 96
capr	1.91 \ 1.94	354 \ 479
clos	507 \ 507	157 \ 157
clos+	70 \ 70	157 \ 157
crte	0.29 \ 0.32	484 \ 679
crni	1.9 \ 1.9	237 \ 237
dich	5.3 \ 5.3	161 \ 161
dich+	2.1 \ 2.1	161 \ 161
diff	4.8 \ 4.8	167 \ 167
edit	0.28 \ 0.21	690 \ 720
fdtd	2.3 \ 2.3	197 \ 197
fft	2.6 \ 2.73	229 \ 289
fiff	0.92 \ 0.92	139 \ 139
fiff+	0.52 \ 0.52	139 \ 139
lgdr	0.06 \ 0.06	197 \ 197
mbrt	3.3 \ 3.3	98 \ 98
nb1d	1.06 \ 1.08	302 \ 327
nb3d	39 \ 39	215 \ 215
scra	0.48 \ 0.48	433 \ 533
spqr	0.45 \ 0.46	402 \ 497

with level 3 optimization of GFortran, the performance is even improved more than without adding the checks, for example, the running time of the benchmark *bubl* runs two times faster after adding the checks. In order to find the potential reasons, we compiled the generated FORTRAN both with and without the array bounds checking using **-S** option to get the assembly code. By reading the assembly code, we found that the compiler performs some optimization on the STOP statement in FORTRAN. Since the STOP statement provides a potential exit of the program, the GFortran compiler will not generate assembly code for the statements following the STOP statement in the same path.

The difference between the number of LOC of the generated FORTRAN code without and with array bounds checking depends on two factors: (1) how much static information do we know about the shape of the accessed arrays and the constant or range value of the indices; (2) how many array read operations are there in the original input MATLAB code. In other words, even if there are a lot of array read operations, as far as the shape of the accessed arrays and the constant or range value of those indices are exactly known, we can perform the array bounds checking at compile-time and thus don't need to inline any checking code in the generated FORTRAN. For the experimental results in Table 6.2, we can see that when we pass arrays with compile-time-known shapes in the driver files to the benchmarks, there will be no extra inlined array bounds checking code in the generated FORTRAN, while we pass arrays with compile-time-unknown shapes in the driver files to the benchmarks, there will be inlined array bounds checking code in the generated FORTRAN.

6.4 Summary

In summary, the overall running time of the generated FORTRAN code from MC2FOR for these benchmarks is better than the running time of these benchmarks running in MATLAB, and according to the comparison of the LOCs, the size of the generated FORTRAN code is in an acceptable range.

Chapter 7
Related Work

Before MathWorks put a just-in-time (JIT) accelerator under the hood of MATLAB, its inefficient performance had already drawn some attention from researchers and engineers. FALCON [RP99] is a MATLAB to FORTRAN 90 translator with a sophisticated type inference mechanism. Although the FALCON project provided us with a lot of interesting ideas about how to proceed, MC2FOR has quite a few important differences. For example, the inference mechanism in FALCON is based on a forward/backward propagation strategy, while our analysis only involves a forward propagation. FALCON distinguishes scalar, vector and matrix, while we treat all the variables as a matrix. Scalar is a 1-by-1 matrix and vector is a 1-by-n or n-by-1 matrix. FALCON uses static single assignment (SSA) form to make sure all the variables have only one definition, this may simplify the code generation, but may also introduce some extra overhead to the transformed program. Instead, we only split the variables with different types in different webs of definitions and uses. The two projects also have totally different approaches to shape analyses for MATLAB built-in functions: FALCON implements a table for each built-in function to tell how the shape of output depends on the shape of inputs, but this strategy cannot support the case where the shape of output depends on the value of inputs. Further, the type system of MATLAB had been extended since FALCON, and our approach thus handles more MATLAB types. Our system is also available for other researchers.

There are many existing range analyses for different purpose. The one implemented in MC2FOR is specific to address MATLAB and closest to a generalized constant propagation

in C [VCH96] which proposed a similar analysis to estimate the range of a variable may reach at each program point. The range value analysis through MATLAB built-in functions also has its roots in the interval arithmetic.

MC2FOR builds upon previous work in the *Mc*LAB group. In early work, Jun Li developed a prototype which demonstrated the feasibility of translating MATLAB to FORTRAN 95 [Li09]. This early prototype focused on a limited subset of MATLAB and made simplifying assumptions. To provide a more solid analysis basis, the ***Mc*SAF** analysis framework and Tamer's extensible interprocedural abstract value analysis framework were developed. These two frameworks working together form the major transformation and analysis engine in the *Mc*LAB toolkit. Concurrent to our development of MC2FOR, our lab is also working on another project to statically compile MATLAB to **X10** [IBM12, KH13], which also uses the shape analysis in MC2FOR.

MATLAB Coder™ is a commercial translator to generate standalone C and C++ code from MATLAB. MATLAB Coder supports a subset of core MATLAB language features and is a closed source system, with no research papers on its design. In Chapter 6, we already listed the experimental results of our testing benchmarks on MATLAB Coder and compared the performance nd LOC of the generated files of the Coder with our MC2FOR. Part of the objective of our work is to provide an open source framework, which other researchers can easily use. For example, the *Mc*LAB toolkit, plus the shape and range analysis presented in this thesis would be a suitable starting point for developing a C/C++ back end.

Besides MATLAB, programmers or researchers also have interests in other similar array languages, for example, Python and R. Shed Skin[1] is an experimental restricted Python to C++ compiler, that can translate pure, but implicitly statically typed Python (2.4-2.6) programs into optimized C++ programs. The Cython language[2] is a superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes. This allows the compiler to generate very efficient C code from Cython code. Rcpp[3] provides a powerful API on top of R, permitting direct interchange of rich R objects (including S3, S4 or Reference Class objects) between R and C++.

[1] http://code.google.com/p/shedskin/
[2] http://www.cython.org/
[3] http://www.rcpp.org/

Chapter 8
Conclusions and Future Work

In this thesis, we have presented a source-to-source compiler which transforms MATLAB programs to equivalent FORTRAN programs. Since MATLAB is a dynamic and weakly-typed language, while FORTRAN is a static and strongly-typed language, there are quite a number of challenges.

In our MC2FOR, we introduced a shape analysis in Chapter 3, which is used to estimate the number and extent of dimensions of all the variables in a given MATLAB program. In the shape analysis, we also proposed a domain-specific language, the shape propagation equation language, to write equations used for propagating shape information through MATLAB built-in functions. In order to remove unnecessary run-time array bounds checking code in the transformed FORTRAN program, we implemented a range value analysis specific for MATLAB in Chapter 4, which is an extension of constant analysis in our framework, to estimate the possible range of value a scalar variable may reach at each program point. Both the shape and range value analysis are implemented in the Tamer's framework. In the code generation of MC2FOR in Chapter 5, we started with our approach to assigning declared types and introducing explicit type conversions, then we introduced the built-in mapping framework used to map numerous MATLAB built-in functions to FORTRAN, and we also presented the way how MC2FOR handles the various linear indexing transformation from MATLAB to FORTRAN.

Finally, in Chapter 6, we evaluated our MC2FOR on a collection of 20 MATLAB benchmarks. First we compare the running time of the original benchmarks under MATLAB

2013a, which is the standard software and environment to run the MATLAB code, and the GFortran-compiled FORTRAN programs generated by MC2FOR for the benchmarks. We also examine the running time of the benchmarks under Octave, which is an open source software and environment for numerical computations and mostly compatible with MATLAB, and the C programs generated by MATLAB Coder for reference. We use a scale number for each benchmark to make sure that the time of the benchmarks running under MATLAB, Octave, FORTRAN and MATLAB Coder is for solving the same size of problem. Besides the running time, another factor to evaluate the performance of our compiler is the size of the code generated by MC2FOR for the input MATLAB benchmarks. So we list the physical lines of code (LOC) of the original MATLAB benchmarks and the generated FORTRAN code by MC2FOR. We also list the LOC of the generated C code by MATLAB Coder for reference. From the results, we show that the code generated by MC2FOR is usually more efficient than the original MATLAB code running under MATLAB, at the cost of only a modest increase in code size

In order to improve the performance of MC2FOR, we plan to make the range value analysis support symbolic values. In this way, we may remove more run-time array bounds checking code in the transformed program. Moreover, we may also want to translate MATLAB code into parallel FORTRAN code, in order to achieve this, we need a valid dependency analysis to determine which MATLAB code block is free from dependency and safe to be transformed to parallel code. We also hope that others will build upon our tool, which has been implemented in an extensible manner, and is freely available at `www.sable.mcgill.ca/mc2for.html`.

Appendix A
Shape Propagation Equation Language

A.1 Tokens

```
LineTerminator = \r|\n|\r\n
WhiteSpace = {LineTerminator} | [ \t\f]
Number = -? [:digit:] [:digit:]*
Identifier = [:jletter:] [:jletterdigit:]+
Uppercase = [A-Z]
Lowercase = [a-z]

{Number} -> Terminals.NUMBER
{Identifier} -> Terminals.ID
{Uppercase} -> Terminals.UPPERCASE
{Lowercase} -> Terminals.LOWERCASE
"$" -> Terminals.SCALAR
"#" -> Terminals.ANY=
"||" -> Terminals.OROR
"->" -> Terminals.ARROW
"," -> Terminals.COMMA
"(" -> Terminals.LRPAREN
```

```
")" -> Terminals.RRPAREN
"?" -> Terminals.QUESTION
"*" -> Terminals.MULT
"+" -> Terminals.PLUS
"=" -> Terminals.EQUAL
"[" -> Terminals.LSPAREN
"]" -> Terminals.RSPAREN
"|" -> Terminals.OR
"'" -> Terminals.SQUOTATION
```

A.2 Grammar

```
caselist
    = case.c
    | case.c OROR caselist.l
    ;

case
    = patternlist.p ARROW outputlist.o
    ;

outputlist
    = vectorExpr.v
    | vectorExpr.v COMMA outputlist.o
    ;

patternlist
    = pattern.e
    | pattern.e COMMA patternlist.p
    ;
```

```
pattern
    = matchExpr.m
    | assignExpr.a
    | assertExpr.a
    ;

matchExpr
    = basicMatchExpr.m OR basicMatchExpr.n
    | basicMatchExpr.m QUESTION
    | basicMatchExpr.m MULT
    | basicMatchExpr.m PLUS
    | basicMatchExpr.m
    ;

basicMatchExpr
    = LRPAREN patternlist.p RRPAREN
    | SQUOTATION ID.i SQUOTATION
    | SQUOTATION LOWERCASE.i SQUOTATION
    | vectorExpr.v
    ;

vectorExpr
    = SCALAR.d
    /* used to match a scalar input argument,
    whose shape is [1,1]. */
    | UPPERCASE.u
    | vertcatExpr.v
    /* used to match a vertcat vector,
    something like [m,n], [1,k] or even []. */
    | ANY.a
    ;
```

```
vertcatExpr
    = LSPAREN RSPAREN
    | LSPAREN arglist.al RSPAREN
    ;

arglist
    = arg.a
    | arg.a COMMA arglist.al
    ;

arg
    = scalarExpr.s
    | vectorExpr.v
    ;

scalarExpr
    = NUMBER.n
    | LOWERCASE.l
    ;

assignExpr
    = assignmentLHS.l EQUAL assignmentRHS.r
    ;

assignmentLHS
    = LOWERCASE.l
    | UPPERCASE.u
    | UPPERCASE.u LRPAREN scalarExpr.s RRPAREN
    | ANY.a LRPAREN scalarExpr.s RRPAREN
    ;
```

```
assignmentRHS
    = scalarExpr.s
    | vectorExpr.v
    | fnCall.f
    ;

fnCall
    = ID.i LRPAREN RRPAREN
    | ID.i LRPAREN arglist.al RRPAREN
    ;

assertExpr
    = fnCall.f
    ;
```

A.3 Some Shape Equation Examples

The equation for the built-in functions `pi`, `i` and `j`:

```
[] -> $
```

The equation for the built-in functions `tril` and `triu`:

```
M,$? -> M
```

The equation for the built-in function `diag`:

```
[m,n],k=minimum(m,n) -> [k,1]
```

The equation for the built-in functions `plus`, `minus` and `times`:

```
$|M,$|M -> M
```

The equation for the built-in function `mtimes`:

```
$|M,$|M -> M || [m,n],[n,k] -> [m,k]
```

The equation for the built-in function `mpower`:

```
$,$|M -> M || $|M,$ -> M
```

The equation for the built-in function `mldivide`:

```
$,M->M||[m,k],[m,n]->[k,n]
```

The equation for the built-in function `mrdivide`:

```
M,$->M||[m,k],[n,k]->[m,n]
```

The equation for the built-in functions `min` and `max`:

```
[1,n]|[n,1] -> $
|| M,M(1)=1 -> M
|| M,M -> M
|| M,[],$,n=previousScalar(),M(n)=1 -> M
```

The equation for the built-in function `median`:

```
[1,n]|[n,1] -> $
|| M,M(1)=1 -> M
|| M,M -> M
|| M,$,n=previousScalar(),M(n)=1 -> M
```

The equation for the built-in functions `sin`, `cos`, `tan`, `cot` and so on:

```
$|M -> M
```

Note that the equation for above functions can also be written as:

```
$ -> $ || M -> M
```

The equation for the built-in functions `eq`, `ne`, `lt`, `gt`, `le` and `qe`:

```
$|M,$|M -> M
```

The equation for the built-in function `colon`:

```
$,n=previousScalar(),$,m=previousScalar(),k=minus(m,n) -> [1,k]
|| $,n=previousScalar(),$,i=previousScalar(),
$,m=previousScalar(),k=minus(m,n),d=div(k,i) -> [1,d]
```

The equation for the built-in functions `ones`, `zeros`, `magic` and `eye`:

```
[] -> $
|| ($,n=previousScalar(),add())+ -> M
```

A.4 Implementation Details

Based on the grammar of the shape equation language, every correct shape equation can be parsed into a parsing tree. Matching process is achieved by implementing a traversal object to walk through the parsing tree applying the matching algorithm which is introduced in Subsection 3.1.4. Following are some major information or data members a traversal object must equip in order to accomplish this task.

- The input arguments' string names of current encountered built-in function;
- All the value information achieved before this function. If the input arguments are some variables, assuming the program is correct, those variables should be defined before this function call, and their value information should be already stored. If the input arguments are integer literals, according to the tamer simplified transformation, all those integer literals will be replaced by assigning the literals to some temporary variables and then using the temporaries as the input arguments. In this case, the temporary assignments will be analyzed prior to the built-ins, and when the propagator goes to the built-ins, the value information of the input temporary arguments will be already known;
- A pointer[1] which is used to index the current matching input argument. While the matching for the current input argument is successful, the pointer should point to the

[1] Which is `MatchingPosition` in our pseudocode for the matching algorithm.

next input argument if there is any left. Each time we start matching for a new case, the pointer will reset to point to the first input argument;

- A boolean value. Since there can be shape matching expressions on both pattern list and shape output list side, there should be a boolean value which is used as a flag to help the traversal object to choose different operations when it encounters shape matching expression nodes on different sides.

According to the shape matching algorithm, when a traversal object traverses the parsing tree, in the pattern list side, a shape matching expression node will try to match the shape of current pointed input argument, an assignment expression node will try to get extra information more than shape of previous matched argument or do some preparation for the final shape result emission, and an assert expression node will try to determine whether the matching progress should carry on based on some status examination. After the traversal object succeeds walking through the pattern list side, the shape output list side will produce the shape information of the output argument(s) using all the information collected during the traversal object traversing the tree.

Here is an example in Figure A.1 to illustrate how a traversal object produces shape information of the output argument(s) by traversing the shape equation parsing tree. In a given program, a shape propagator is walking through the program to estimate shape information of all the variables. When the propagator encounters a built-in function call, in this example, `true(3,3)`[2], first, it will look up this built-in function's shape propagation equation in a sort of built-in function dictionary where we put all the shape and other value propagation equations for MATLAB built-ins. In the dictionary, the propagator get the shape propagation equation for this function as:

```
($,m=previousScalar(),add(m))* -> M
```

This equation will be parsed into a corresponding parsing tree, as it is in Figure A.1. Then the shape propagator will generate a traversal object to traverse the tree with the input argument list, `(3,3)`. The traversal object will go through the whole tree to complete the matching process. In Figure A.1, we can see that the first input argument, constant

[2]Built-in function `true` will return a matrix of logical 1s.

Figure A.1 Example to illustrate shape matching process

scalar `3` will be matched by the `$` symbol, and then its value will be stored into the lower-case `m` by an assignment expression `m=previousScalar()`. The function call expression `add(m)` will add `m` into the default return shape array. The `()*` compound operator will test whether there are still remaining input arguments, if there is, repeat the matching process again with the pattern expressions inside `()` operator. In this example, the enclosed pattern expressions will repeat the matching process again on the second input argument, `3`. After the pattern list side matches the input argument list successfully, the traversal object goes to the shape output list side. In the parsing tree, the shape output list side is the right hand side subtree of the node `->`. The traversal object will produce shape propagation result by looking up the shape information stored in `M`, and we know that since `M` doesn't appear in the pattern expression matching side, this `M` represents the default return shape array, which is `[m,m]`. And by look up the associated value of `m`, the final shape propagation result will be `[3,3]`. After the traversal object return the shape propagation result, `[3,3]`, to the shape propagator, the shape propagator will associate the shape information with the corresponding output argument(s), and then continue walking through the rest of the given program.

The code implementation of the shape propagation through MATLAB built-in functions is written in Java and other two compiler-compilers[3], JFlex and Beaver.

- The scanner file is generated by JFlex and the parser file is generated by Beaver.
- We wrote all the CST nodes files by ourselves in which we implement the shape matching algorithm in the corresponding CST nodes.
- There is another class, named **ShapePropMatch**, which is regarded as the traversal object. It contains all the data members and operations which are essential to accomplishing the shape matching process.
- There is also a class, named **ShapePropTool**, which is regarded as the interface connecting the shape analysis in programs and shape propagation through MATLAB built-ins. The scenario is simple, when the shape analysis encounters a built-in function during its analysis in a given MATLAB program, it will invoke the shape propagation for built-ins component through **ShapePropTool** class, and when the shape propagation for the built-in is done, the result will be returned to the shape analysis through this class again.

[3] A compiler-compiler or compiler generator is a tool that creates a parser, interpreter, or compiler from some form of formal description of a language and machine.

A.5 The Aspect File to Detect Array Growth

```
aspect grow

properties
% this aspect catches every set and records data that should be useful
    in
% determining which operations increase or decrease the array size.
% to that effect the size of every variable during the run of the
% program is checked. In the end, the line number of the operation at
% which the size of each array was maximum, is printed out along with
    the size.


variables = struct(); % creates the mapping 'variable' -> index

changeShape = {}; % how often the dimensions of the array changed (has
    to exist previously)
decreaseSize = {}; % how often the size decreased (i.e. a previously
    nonzero element was set)
increaseSize = {}; % how often the size increased

arraySize = {}; % size of the array
maxSize = {}; % maximum size of the array
lineNum = {}; % at line number
arraySet = {}; % the number of 'set' operations
arrayShapePrevious = {}; % the string representation of array size,
    like [2,2]
arrayShapeCurrent = {}; % the same as above

nextId = 1; % next available index
end


methods
function b = sameShape(this,a,b)
```

```
30 % returns true if a and b have the same shape
31 if (ndims(a) ~= ndims(b))
32   b = false;
33 elseif (size(a) == size(b))
34   b = true;
35 else
36   b = false;
37 end
38 end
39
40 function id = getVarId(this,var,line)
41 % get id of variable by string-name, update 'variables' if necessary
42
43   % find id of variable and put it in the variables structure if not
       present
44   if (~isfield(this.variables,var))
45     this.variables = setfield(this.variables,var,this.nextId);
46     id = this.nextId;
47     this.nextId = this.nextId+1;
48     % initialze entry <id> for all the cell arays
49     this.arraySet {id} = 0; % the number of 'set' operations
50     this.changeShape{id} = 0; % how often the dimensions of the array
         changed (has to exist previously)
51     this.decreaseSize{id} = 0; % how often the size decreased (i.e. a
         previously nonzero element was set)
52     this.increaseSize{id} = 0; % how often the size increased
53     this.arraySize{id} = 0;
54     this.maxSize{id} = 0;
55     this.lineNum{id} = line;
56     this.arrayShapePrevious{id} = '';
57     this.arrayShapeCurrent{id} = '';
58   else
59     id = getfield(this.variables,var);
60   end
61 end
62
63 end
```

```
64
65 patterns
66 arraySet : set(*);
67 execMain : mainexecution();
68 end
69
70 actions
71 message : before execMain
72  disp('tracking the operations that grow arrays in the following
       program...');
73 end
74
75
76 displayResults : after execMain
77 % will display the results
78   vars = fieldnames(this.variables);
79   result = {'var', 'arraySet', 'shape changes', 'decrease', 'increase',
         'max size', 'previous shape', 'current shape'};
80   pm = [' ', char(0177)];
81   for i=1:length(vars) %iterate over variables
82     result{i+1,1} = vars{i};
83     result{i+1,2} = this.arraySet{i};
84     result{i+1,3} = this.changeShape{i};
85     result{i+1,4} = this.decreaseSize{i};
86     result{i+1,5} = this.increaseSize{i};
87     result{i+1,6} = this.maxSize{i};
88     result{i+1,7} = this.arrayShapePrevious{i};
89     result{i+1,8} = this.arrayShapeCurrent{i};
90   end
91   disp(result);
92 end
93
94
95 bset : before arraySet : (newVal,obj,name,line,args)
96    t = obj;
97    t(args{1:numel(args)}) = newVal;
98    newVal = t;
```

```
99
100    % we will exit if the newval is not a matrix
101    if (~isnumeric(newVal))
102      return;
103    end;
104
105    % get id of variable by string-name, update 'variables' if necessary
106    id = this.getVarId(name,line);
107
108    % get var infor
109    newSize = numel(newVal);
110    oldSize = this.arraySize{id};
111
112    this.arraySize{id} = newSize;
113
114    % update the number of 'set' operations
115    this.arraySet{id} = this.arraySet{id}+1;
116
117    % set shape/sparsity changes
118    if (~this.sameShape(newVal,obj))
119      this.arrayShapePrevious{id} = num2str(size(obj));
120      this.changeShape{id} = this.changeShape{id}+1; % how often the
           dimensions of the array changed (has to exist previously)
121    end
122    if (newSize < oldSize)
123      this.decreaseSize{id} = this.decreaseSize{id}+1; % how often the
           size decreased
124    end;
125    if (newSize > oldSize)
126      this.increaseSize{id} = this.increaseSize{id}+1; % how often the
           size increased
127      this.lineNum{id} = line;
128      this.maxSize{id} = newSize;
129    end
130  end
131
132  aset : after arraySet : (newVal,obj,name,line)
```

```
133     id = this.getVarId(name,line);
134     this.arrayShapeCurrent{id} = num2str(size(obj));
135 end
136 
137 end
138 end
```

Listing A.1 The aspect file to detect array growth

Bibliography

[DH12a] Jesse Doherty and Laurie Hendren. McSAF: A Static Analysis Framework for MATLAB. In *Proceedings of ECOOP 2012*, 2012, pages 132–155.

[DH12b] Anton Dubrau and Laurie Hendren. Taming MATLAB. In *Proceedings of OOP-SLA 2012*, 2012, pages 503–522.

[DHR11] Jesse Doherty, Laurie Hendren, and Soroush Radpour. Kind Analysis for MAT-LAB. In *In Proceedings of OOPSLA 2011*, 2011, pages 99–118.

[Doh11] Jesse Doherty. McSAF: An Extensible Static Analysis Framework for the MAT-LAB Language. Master's thesis, McGill University, December 2011.

[Dub12] Anton Dubrau. Taming matlab. Technical report, School of Computer Science, McGill University, April 2012.

[GNU13] GNU. GNU Fortran Home Page, 2013. http://gcc.gnu.org/fortran/.

[IBM12] IBM. X10 programming language. http://x10-lang.org, February 2012.

[KH13] Vineet Kumar and Laurie Hendren. First Steps to Compile MATLAB to X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, 2013, pages 2–11.

[Li09] Jun Li. McFOR: A MATLAB to FORTRAN 95 compiler. Master's thesis, August 2009.

[Mata] MathWorks. MATLAB Coder. http://www.mathworks.com/products/matlab-coder/.

[Matb] Matlab. The Language Of Technical Computing. Home page http://www.mathworks.com/products/matlab/.

[McL] McLab. Mclab home page. Home page http://www.sable.mcgill.ca/mclab/.

[Mola] Cleve Moler. MATLAB Incorporates LAPACK. http://www.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html.

[Molb] Cleve Moler. The Origins of MATLAB. http://www.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html.

[oct] GNU Octave. http://www.gnu.org/software/octave/index.html.

[RP99] Luiz De Rose and David Padua. Techniques for the Translation of MATLAB Programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.

[TAH10] Anton Dubrau Toheed Aslam, Jesse Doherty and Laurie Hendren. Aspectmatlab: An aspect-oriented scientific programming language. In *AOSD '10: Proceedings of the 9th international conference on Aspect-oriented software development*, Rennes and St. Malo, France, 2010, pages 181–192. ACM, New York, NY, USA.

[VCH96] Clark Verbrugge, Phong Co, and Laurie Hendren. Generalized Constant Propagation A Study in C. In *Proceedings of the 1996 International Conference on Compiler Construction*, 1996, CC '96.