Mc2for: a tool for automatically transforming Matlab to Fortran 95

Sable Technical Report No. sable-2013-04

Xu Li and Laurie Hendren

October 14, 2013

Contents

List of Figures

List of Tables

Abstract

Matlab is a dynamic numerical scripting language widely used by scientists, engineers and students. While Matlab's high-level syntax and dynamic types makes it ideal for prototyping, programmers often prefer using high-performance static programming languages such as Fortran for their final distributable code. Rather than requiring programmers to rewrite their code by hand, our solution is to provide a tool that automatically translates the original Matlab program to produce an equivalent Fortran program. There are several important challenges for automatically translating Matlab to Fortran, such as correctly estimating the static type characteristics of all the variables in a Matlab program, mapping Matlab built-in functions, and effectively mapping Matlab constructs to FORTRAN constructs.

In this paper, we introduce **Mc2for**, a tool which automatically translates Matlab to Fortran. This tool consists of two major parts. The first part is an interprocedural analysis component to estimate the static type characteristics, such as array shape and the range value information, which are used to generate variable declarations in the translated Fortran program. The second part is an extensible Fortran code generation framework to automatically transform Matlab constructs to corresponding Fortran constructs. This work has been implemented within the McLab framework, and we demonstrate the performance of the translated Fortran code for a collection of Matlab benchmark programs.

1 Introduction

Matlab is a well established programming 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 [5]. 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 the programmers by providing **Mc2for**, a tool that automatically converts Matlab programs to Fortran95 programs. This tool enables Matlab users to move their applications from Matlab to Fortran without the effort and knowledge required of manually rewriting their code. To be generally useful our tool 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 matrix 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

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

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 tool must implement sophisticated static analyses, including a mechanism to analyze the many built-in functions.

The main contributions of this paper are as follows:

Identified need/challenges: We have identified the need for a tool to help programmers convert 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 rules.

Range Analysis: We have designed and implemented a custom range analysis that is used to minimize the overhead of array bounds checking and array resizing in the generated Fortran code.

Code Generation Strategies: We have designed and implemented code generation strategies for both the simple control constructs and for 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 benchmarks, showing that we can produce efficient and compact code.

The paper is structured as follows. In Section 2 we give the necessary background and the overall structure of our tool. In Section 3 we provide a detailed explanation of our shape analysis, including our approach for built-in functions. Section 4 describes our approach to range analysis, which is used to minimize array bounds checks and array resizing checks. Section 6 provides our empirical study of using the tool on a collection of Matlab benchmarks, Section 7 discusses related work, and finally we conclude in Section 8.

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 programming language nature, Matlab is especially suitable for solving linear algebra problems. For example, Listing 1 shows a Matlab implementation of a well known linear algebra algorithm, the Babai nearest plane algorithm. This algorithm is an approximation to solve the closest vector problem and has pervasive applications in the field of wireless communication. Sometimes the algorithm has to be implemented on hardware, which means that it requires a lot of efficiency. Imagine that we want to transform this Matlab implementation to Fortran- what potential problems we may encounter?

```
function z_hat = babai(R,y)
%%
%   compute the Babai estimation
%   find a sub-optimal solution for min_z ||R*z-y||_2
%   R - an upper triangular real matrix of n-by-n
%   y - a real vector of n-by-1
```

```
 7 %   z_hat - resulting integer vector
 8 %%
 9   n=length(y);
10   z_hat=zeros(n,1);
11   z_hat(n)=round(y(n)./R(n,n));
12 
13   for k=n-1:-1:1
14      par=R(k,k+1:n)*z_hat(k+1:n);
15      ck=(y(k)-par)./R(k,k);
16      z_hat(k)=round(ck);
17   end
18 end
```

Listing 1: Matlab implementation of Babai algorithm

First of all, how should we declare the Matlab variables in the transformed Fortran program? Matlab is a dynamic programming 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 1, how should we map the `length` function at line 9, the `zeros` function at line 10 and the `round` function at lines 11 and 16. Thirdly, besides these two significant problems, we also need to think about how to map Matlab constructs to the equivalent constructs in Fortran; how should we 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?

In order to solve these problems, we designed and implemented the **Mc2for** tool, as illustrated in Figure 1. First, focus on the input (top of figure) and output (bottom of figure) of **Mc2for**. Note that the user provides the Matlab file which is the entry point of the user's program, as well as any other Matlab files that may be used by the program. 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** tool 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 tool is a collection of Fortran functions, which can be compiled with any Fortran95-compliant compiler. Thus, from the user's point of view, it is very simple to use **Mc2for**.

Now let us concentrate on the actual structural organization of **Mc2for**. The central component driving the compilation process is the Tamer module [4]. 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* is used to scan and parse the file, generating a high-level IR, McAST. The analysis and transformation engine, McSAF [2] is then used to transform to a lower-level AST; and to perform initial analyses such as *kind analysis* [3], which determines

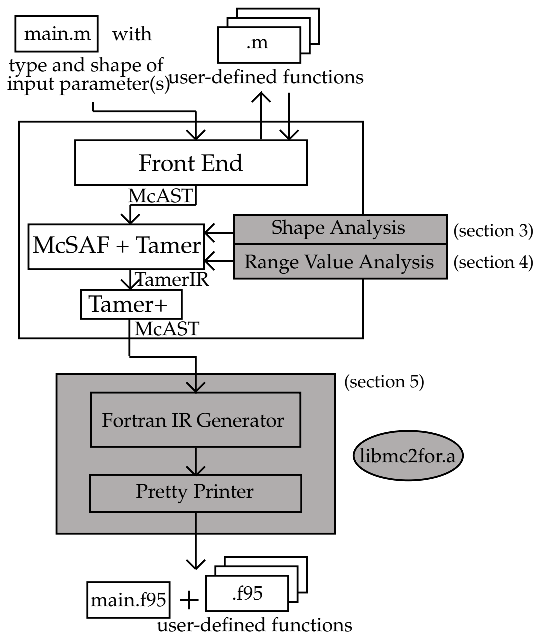

Figure 1: The Overview of **Mc2for**. We highlight the boxes which are the contributions of this paper. Note the size of the boxes does not correspond to either the complexity or the importance of the component.

which identifiers refer to arrays, and which refer to functions.[2] The Tamer then processes the IR into an even lower-level 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 framework. The Tamer 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 provides some basic MATLAB type characteristics analyses, like 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 point of the program. The range value analysis can assist the shape analysis in the situation of the array bounds checking.

The transformed IR from Tamer, 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 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 back end. By traversing the McAST, the back end generates an 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

[2] 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.

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. Each of them maps the entry point function file or the user-defined function file(s). The resulting Fortran programs should be easy to redistribute, since they can be compiled with any Fortran95-compliant compiler (including the open source GFortran). Further, as we show in Section 6, the resulting Fortran code is often significantly more efficient than the original Matlab code.

3 Shape Analysis

We use the term *shape* to refer to the number of dimensions and the size of each dimension of a Matlab variable. The shape information of variables in a given Matlab program is essential for transforming Matlab to Fortran. In order to propagate the shape information through an entire given program, we have developed a shape analysis which is implemented using the Tamer's extensible forward interprocedural abstract value analysis framework. The Tamer framework handles propagating the abstract values and computing the fixed points. We only need to provide the following: (1) an implementation of the abstract representation of shapes, (2) a mechanism for processing shapes for built-in Matlab functions, and (3) a merging operator that merges two abstract shapes.

In our shape analysis, we abstract shapes by a list of numbers or lowercase letters[3] enclosed by a pair of square brackets to represent the shape information of a matrix, and each element in the list represents the size of the corresponding dimension of the matrix. For example, if `arr` is a 3-by-5 matrix, we will represent its shape as `[3,5]` in which `3` represents the size of the first dimension of `arr` and `5` represents the size of the second dimension.

3.1 Handling Built-ins

Recall the Babai algorithm implementation in Listing 1. At line 9, if we know the shape of `y` is 15-by-1, what is the shape of variable `n` after evaluating this line, in other words, how does the shape information propagate through the Matlab built-in function `length`? The same problem pops up again at line 10 for the built-in function `zeros` and at lines 11 and 16 for the built-in function `round`.

In this section, we design a concise domain specific language, *shape propagation equation language*, to describe the behavior of how the shape information propagates through Matlab built-in functions. With this concise language, we can propagate the shape information through the built-in function call statements.

3.2 Typical Behaviors of Matlab Built-ins on Shapes

Before formally introducing the shape propagation equation language and how to use this language to describe the behavior of how the shape information propagates through Matlab built-in

[3]In our analysis, the size of a certain dimension can be represented as a symbol.

functions, we need to summarize some most typical behaviors. By studying a a large number of MATLAB built-in functions, we summarized the key possible behaviors into four major categories.

Based on the shape of input argument(s): The most common behaviour is that the shape of the output argument(s) only depends on the shape of the input argument(s). For example, the return shape of some commonly-used arithmetic built-ins, like `+`, `-`, `.*` and `./`, only depends on the shape of the input arguments.[4]

Based on the numeric value of input argument(s): The shape of the output argument(s) of some built-in functions depends on the numeric value of the input argument(s). For example, the return shape of the built-in `zeros` at line 10 in Listing 1 depends on the value of its input argument. In this example, the shape of `z_hat` after evaluating this statement will be `[n,1]`.

Based on optional numbers or strings: Some MATLAB built-ins allow optional numbers or character strings to control the shape of the output argument(s). For example, the return shape of the built-in function `svd`, which is used to compute singular value decomposition of a matrix, depends on an optional input number argument, `0`, and an optional input string argument, `'econ'`. In the case of `[U,S,V] = svd(X)`, assuming the shape of `X` is `[3,2]`, the shape of `U`, `S` and `V` will be `[3,3]`, `[3,2]` and `[2,2]`, respectively; while, in the case of `[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.

Other cases: The three previous categories already cover most behaviours. However, there are still a few special cases in MATLAB. For example, the built-in function `cross`, which computes the cross product of two vectors or matrices. Besides the requirement that both the inputs must have the same shape, it also requires that the vectors must be 3-element vectors or the matrices must have at least one dimension of size 3.

3.3 Shape Analysis through MATLAB Built-ins

In this subsection, we introduce a concise domain-specific language, the *shape propagation equation language* (SPEL). Using this language, we can write a *shape propagation equation* (SPE) for each MATLAB built-in function to describe the behavior of how the shape information propagates through the function. In order to make the language as concise as possible, we focus on supporting features needed to describe all the typical 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 functions work on shapes by writing SPEs in this language.

Besides the language and the SPEs, we also present the *shape matching algorithm*. This algorithm takes as input: (1) the abstract value information of the input arguments to the call of the built-in, and (2) the SPE for the built-in; and produces, as output, the shape information of the output argument(s) of the built-in call. For example, for a built-in function call `a = ones(m,n)`, the shape matching algorithm would take as input the abstract values of `m` and `n` and the SPE rule for `ones`, and would produce an estimate of the shape for `a`. For this case, the algorithm will use the constant value information in the abstract value information of `m` and `n` to return the shape of `[m,n]`.

In the remainder of this subsection, we introduce the general structures and the semantics of constructs in SPEL and at the same time we explain how the shape matching algorithm infers the output shape, starting with the top-level constructs of the SPEL.

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

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

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

The separated cases are evaluated from left to right by the shape matching algorithm. If any of them are matched successfully with the shape of input argument(s), the matching process will terminate and return 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 on the left-hand side of the ARROW symbol, and all the shape output list expressions will be on the right-hand side.

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

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

PATTERN LIST SIDE The pattern list side is composed of a list of pattern expressions which are separated by COMMA (,) symbols.

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

The pattern expressions are evaluated from left to right. 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. **Mc2for** will throw a warning to the user.

PATTERN EXPRESSION Pattern expressions can be categorized into three different groups: shape matching expressions, helper function calls, and assignment expressions. Among these, only the shape matching expressions are used to match the shape of the input argument(s) and if the matching is successful, the current 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, helper function calls and assignment expressions, are used for special checks and storing information during the shape propagation process.

Shape matching expression (SME) : There are four kinds of symbols which are used to represent shape matching expressions: the DOLLAR (`$`) symbol, upper-case letters, dimension expressions and the ANY (`#`) symbol.

- The `$` symbol is used to match scalars, which in MATLAB are stored at 1-by-1 arrays;
- Upper-case letters match input arguments of matrices which are not in the shape of 1-by-1. 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 represent a matrix shape, not combination of letters; and
- Dimension 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]`. Dimension expressions are also used to match input arguments of matrices, while it may impose more restrictions on the number of dimensions or/and the size of certain dimension;

- The `#` symbol: In some cases, we may not care about the shape of current input argument. We use this symbol as a wildcard to consume the current input argument, no matter what the shape it has.

Helper function calls: There are a set of pre-defined functions which provide some extra computation to assist the shape propagation process. For instance, the helper function `previousScalar` retrieves the value of previous matched scalar input argument. Some of the helper functions are also used as assert expressions, which have the functionality to control whether the matching process should continue on based on certain conditions. For example, the assert expression `atLeastOneDimEqls(arg)` checks whether there is at least one dimension's size of matched matrix equals `arg`, if not, the current matching process will terminate and start over from next case again if there is any case left.
Since MathWorks may introduce new MATLAB built-in functions with some new restrictions in the future, we made this language extensible by allowing users to add new pre-defined functions into the language, which could then be supported in matching algorithm.

Assignment expression: `lvalue = rvalue`, where `lvalue` can be lower-case letters, upper-case letters, `#` symbol, and indexed upper-case letters. The `rvalue` can be numbers, lower-case letters, other shape matching expressions and helper function calls. Assignment expressions are used during the matching process to store extra needed information to assist the shape analysis. The assignment expression `n=previousScalar()` will be explained in a SPE for the built-in function `zeros` after a few lines.

SHAPE OUTPUT LIST SIDE The shape output list side contains a list of only shape expressions, specifying the shapes of the output parameters.

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

Note that the matching of the input arguments and binding of values are done by the pattern list side, and the building and returning of the the shape of the output is done by the shape output list side.

OPERATORS We have also defined a set of operators.

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 work together with other operators to achieve advanced matching logic;

The ? operator: Putting a question mark operator after a shape matching expression or a 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 a 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 depends on the number of input argument(s);

The * operator: Putting a star operator after shape matching expression or a compound expression in the pattern list side means that during the matching process, the preceding expression may be evaluated one or more times 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 after the operator;

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

Now let us consider some SPEs for the built-in functions for our example Babai algorithm in Listing 1: `length`, `zeros` and `round`. For `length`, it doesn't care about the shape of the input argument, no matter whether the input is a scalar (`$`) or a matrix (`M`), `length` will always return a scalar as a result, which means that the return shape is `$`.

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

For the built-in `zeros`, if the input argument list is empty (`[ ]`), the built-in `zeros` will return a scalar `0` (`$`); if not, each element in the list represents the size of corresponding dimension of the returned shape.

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

The second line of this equation is interpreted as: repeat matching process with the pattern expressions in the parentheses before `+` until there is no input argument to match. The expression inside the parentheses specifies that using `$` to match an input scalar argument, consume this input and associate the value of this scalar with `$`, the expression `n=previousScalar()` will try to fetch the value of previous matched scalar and store the value into `n`, the expression `add(n)` will add the value of `n` into a default vector preparing for final result emission, when there is no input arguments to match, go to the shape output list side. On the output side, `M` is used to represent the default vector if it's not used in pattern list side, the values in the default vector will be the returned shape information.

The `round` function returns the same shape as the shape of its input, so the SPE for `round` is:

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

3.4 Merging Shapes for Flow Control Statements

After solving the problem of estimating the shape information through MATLAB built-in functions, we may have the shape information of all the variables at each point of the program if the program only has sequential statements. What if at the program point of the end of flow control statements, like if-else, for loop and while loop statements, the shape information for the same variable from different branches or different iterations is different and needs to be merged?

The overview merging strategy of the shape is given in Table I.

Table I: 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

There are three categories of abstract shapes. The `ordinary` shape is the shape with a dimension list where some dimensions in the list may be unknown, but at least the number of dimensions is known. The strategy for merging two ordinary shapes is straightforward: if the length of the dimension lists of two shapes are not equal, add 1(s) to the end of the shorter one to make them have the same length. Now, given two dimension lists of the same length, for each dimension: (1)

if the values are equal, keep it as the value for the corresponding dimension in the merged shape; or (2) if the values are not equal, mark the value of that dimension as unknown.

The `not_matched` shape arises when the built-in matcher cannot find a shape match, which corresponds to cases where a programmer misuses a built-in function. Merging `not_matched` with any shape produces `not_matched`.

The `unmergeable` shape arises from our treatment of the fixed-point for `for` and `while` statements. If the shapes from different iterations do not reach a fixed-point after 5[5] iterations, we push the shape to `unmergeable`.

3.5 Summary

In summary, in order to estimate the shape information of all the variables in a given MATLAB program, we implemented a forward interprocedural flow analysis which uses the Tamer's extensible interprocedural abstract value analysis framework. To propagate shape information through the numerous MATLAB built-in functions, we designed a concise domain-specific language, the shape propagation equation language. By writing shape propagation equations in this language and applying our shape matching algorithm, we provided a concise and extensible method to propagate shape information through built-in functions. To handle flow control constructs, we provided a merging strategy which handles ordinary shapes, as well handling the `not_matched` shape used for cases when a built-in is used incorrectly and the `unmergeable` shape used to ensure termination of the fixed-point computation.

4 Range Value Analysis

In translating to FORTRAN, we must ensure that we retain MATLAB's semantics for reading and writing elements of an array. For reading from an array (i.e. an expression of the form `lhs = a(i)`), we must ensure that `i` is within the array bounds, and raise an exception otherwise. For writing to an array (i.e. an expression of the form `a(i) = rhs`), the MATLAB semantics are somewhat unusual. In this case, if `i` is not in bounds, the array should be automatically enlarged so that `i` is in bounds, and the extra columns/rows added should be initialized to 0.

In both the read and write case we need to estimate the value of the index value `i` using range analysis, so as to avoid generating unnecessary dynamic bounds checks in the generated FORTRAN code. In the write case, we also need the range information to eliminate unnecessary checks and reallocation statements, for the case when an array could grow. Furthermore, the range analysis is needed to perform more precise shape analysis, since writing to the array could change its shape.

In order to get a better static array bounds check result, we extended Tamer's constant value analysis to a *range value analysis*, which statically estimates the minimum and maximum values each scalar variable [6] could take at each point of a given program. Similar to shape analysis, the range value analysis uses Tamer's extensible interprocedural abstract value analysis framework. The range value of a variable is a pair of values in the *domain of the range values*: the first element represents the minimum possible value, which we call the lower bound; and the second represents

[5]Any k will work with our approach, empirically we found 5 to be a good setting.

[6]We also support range values for some vector variables, which mostly come from the range expressions in for loops or the array constructions by using colon built-in function.

the maximum possible value, which we call the 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 relational MATLAB built-ins, we add two special superscript symbols, `+` and `-`, for instance, 5^+ and 5^-. You can interpret these two superscripted real numbers as $5+\epsilon$ and $5-\epsilon$, where ϵ is positive and close to 0. For example, < 10,`+inf`> means that variable can be any value greater than or equal to `10` to `+inf`, and <10^+,`+inf`> means that the variable can 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 `+`; and the `upper bound` in a range value can only be one of `+inf`, any real number and any real number with `-`.

4.1 Range Value Analysis through MATLAB Built-ins

We implemented our range value analysis to support the most important MATLAB built-in functions, listed in Table II.

Table II: Range Value Analysis Supported Operators

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

Since the domain of range values involves both symbolic and real number values, the challenge here is how to infer the range value result from computing the symbolic and real numbers together. In this paper, we propose the *range value propagation functions*, which can infer the range value result for the above built-ins based on the range values of their input arguments. The range value propagation functions are based on the order of and some operations defined for the values in the range value domain. The order of the values in the domain is defined when we introduce the domain. To support the range value propagation functions, we have defined a set of arithmetic functions that operate on range values including: `min`, `max`, `==`, `unary +`, `unary -`, `binary +`, `binary -`, $\times$, $\div$, `log` and `exp`. As an example, consider the `binary +` operation on the values in the domain and the range value propagation function for the MATLAB built-in `binary plus`.

`binary +`: 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:[7]
$x^- + y^-$, $x^- + y$ 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.

```
function range_value_binary_plus(op_a, op_b)
  if both op_a and op_b have known range values
    <a,b> = get range value pair from op_a
```

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

```
    <c,d> = get range value pair from op_b
    return <a+c,b+d>
  else
    return unknown
  end if
end function
```

Listing 2: binary plus operator (+)

Besides the twelve built-in operators in Table II, the range value analysis also supports five MATLAB built-in relational operators: less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=) and equal to (==). For example, if the conditional expression in an if clause is `var < 5`, we can infer that in the if branch, the upper bound of the variable `var` is smaller than `5`, which can be represented as <`something, 5`$^-$>. 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.

4.2 Merging Range Values for Flow Control Statements

The merging result of two range values <`a,b`> and <`c,d`> is then the range which covers them both: <`min(a,c),max(b,d)`>, where the `min` and `max` are defined operations on the values in the range value domain. Similar to the shape analysis, if range values from different iterations of loop statements cannot be merged to a fixed point for 5 times, the range value analysis will push the corresponding bounds of the range value to `-inf` or `+inf` respectively.

4.3 Shape Analysis for Matrix Indexing Gets and Sets

The static array bounds check can succeed only when the lower and upper bounds of the range values of all the indices are in real numbers and the shape of accessed matrix is constant at the same time. Otherwise, we have to leave the array bounds check to the runtime by the inlined checking code. The only difference of the static array bounds check and shape analysis of two statements is that the matrix can only be grew by an out-of-bound index in the set statements, not get statements. For the set statements, the MATLAB will always first try to grow the matrix based on the out-of-bound index (or indices) and current shape of the matrix before throwing an exception.[8]

4.3.1 for Get Statements

The static array bounds check in the shape analysis will first extract the range values of each index, then compare the lower bound and upper bound of that index respectively with the lower bound, which is always 1 in MATLAB, and upper bound of the corresponding dimension of the accessed matrix. If an out-of-bound matrix indexing occurs, the shape analysis will inform the abstract value analysis framework to mark the current flow set as `nonviable`[9].

[8]Because of the limitation of the space, the rule of whether MATLAB can succeed in growing the matrix is given on the **Mc2for** web page.

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

4.3.2 for Set Statements

Besides the similar work as for the get statements, when an out-of-bound matrix indexing occurs, the shape analysis will first try to enlarge the accessed matrix based on the rule in the Matlab. If the endeavour succeeds, the shape information of the accessed matrix will be updated to the new shape, if not, the shape analysis should also inform the abstract value analysis framework to mark the current flow set as `nonviable`.

5 Transformation from Matlab to Fortran

After obtaining the shape and range information from analyzing the input Matlab program, we finally get to the extensible Fortran code generation framework of our **Mc2for**. The framework consists of two components: the Fortran IR generator and the IR pretty printer. By traversing the input IR of McAST, the framework transforms the Matlab constructs to equivalent Fortran constructs. During the transformation, the framework builds up the IR of the generated Fortran program. Finally, the IR pretty printer will print out the IR of Fortran into corresponding Fortran files.

First, let's examine the generated code for our example Babai program, given in Listing 3 (automatically produced by **Mc2for** from the Matlab code given in Listing 1). Note that for this example we use the the *no-check mode* mode of **Mc2for**, which tells the tool not to inline any run-time array bounds checking code. This mode is useful when the user has verified (by hand or using some checking aspects) that there are no out-of-bounds problems.

```
MODULE mod_babai
CONTAINS
SUBROUTINE babai(R,y,z_hat)
USE mod_zeros
IMPLICIT NONE
DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE:: z_hat
DOUBLE PRECISION, DIMENSION(:,:) :: R,y
DOUBLE PRECISION :: par, n, ck
INTEGER(KIND=4) :: k
! compute the Babai estimation
! find a sub-optimal solution for min_z ||R*z-y||_2
! R - an upper triangular real matrix of n-by-n
! y - a real vector of n-by-1
! z_hat - resulting integer vector
n = SIZE(y);
! inline runtime allocate
IF (ALLOCATED(z_hat)) THEN
   DEALLOCATE(z_hat)
END IF
ALLOCATE(z_hat(INT(n),1))
!
z_hat = zeros(n, 1.0d+0);
z_hat(INT(n), 1) = NINT((y(INT(n), 1)
                         / R(INT(n), INT(n))));
DO k = INT((n - 1)), 1, -(1)
    par = DOT_PRODUCT(R(k, (k + 1):INT(n)),
                          z_hat((k + 1):INT(n), 1));
    ck = ((y(k, 1) - par) / R(k, k));
    z_hat(k, 1) = NINT(ck);
ENDDO
END SUBROUTINE
END MODULE
```

Listing 3: Fortran generated for the Babai example with no-check-mode

Overall, we believe that the generated code is quite readable, and it works for input arrays of any size. Note we retain the original comments from the MATLAB program, as well as introducing new comments to explain some of the generated code. All variables have been given types according to MATLAB semantics, so some of the types may look surprising. For example, the type of `z_hat` is a `DOUBLE` array, even though the original MATLAB comments said it was an integer vector. The generated code is correct, because the MATLAB `round` function does indeed return type double.

If we run **Mc2for** on this example without the `no-check` flag, then the generated code will include dynamic checks for the array reads and writes in the body of the for loop. For example, the following lines would be inserted at the beginning of the for loop body.

```
! inline runtime ABC and error handle
IF (k < 1 .OR. k > SIZE(R, 1) .OR.
          (k + 1) < 1 .OR. INT(n) > SIZE(R, 2)) THEN
    STOP "INDEX OUT OF BOUND";
END IF
IF ((k + 1) < 1 .OR. INT(n) > SIZE(z_hat, 1)) THEN
    STOP "INDEX OUT OF BOUND";
END IF
```

Note that it is difficult to remove these array bounds checks without more powerful range analyses, and some further information from the user about the symbolic sizes of the input parameters.

However, even tighter code can be generated if the **Mc2for** user is willing to specialize the generated code to specific sized input parameters. For example, if the user was using `Babai` to solve a problem in telecommunications, the shape of `R` and `y` is double the number of antennas in a multiple-in multiple-out system. Thus, the user may wish to generate code for a specific sized problem, and then run the algorithm with different values for that size. If we specify that `R` is a 10-by-10 array and `y` is a 10-by-1 vector, then **Mc2for** generates the code found in Listing 4. Note that in this case the range analysis can precisely estimate all array indices and thus can safely eliminate all dynamic checks from the generated code. Furthermore, the generated FORTRAN can include more specific type declarations, which includes the sizes of the dimensions.

```
MODULE mod_babai
CONTAINS
SUBROUTINE babai(R,y,z_hat)
USE mod_zeros
IMPLICIT NONE
DOUBLE PRECISION, DIMENSION(10,1) :: z_hat
DOUBLE PRECISION , DIMENSION(10,10) :: R
DOUBLE PRECISION , DIMENSION(10,1) :: y
DOUBLE PRECISION :: par, n, ck
INTEGER(KIND=4) :: k
!    compute the Babai estimation
!    find a sub-optimal solution for min_z ||R*z-y||_2
!    R - an upper triangular real matrix of n-by-n
!    y - a real vector of n-by-1
!    z_hat - resulting integer vector
n = SIZE(y);
z_hat = zeros(n, 1.0d+0);
z_hat(INT(n), 1) = NINT((y(INT(n), 1)
                         / R(INT(n), INT(n))));
DO k = INT((n - 1)), 1, -(1)
    par = DOT_PRODUCT(R(k, (k + 1):INT(n)),
                      z_hat((k + 1):INT(n), 1));
    ck = ((y(k, 1) - par) / R(k, k));
    z_hat(k, 1) = NINT(ck);
ENDDO
END SUBROUTINE
END MODULE
```

Listing 4: Fortran generated for the Babai example with specific dimensions given for input parametersr

As we have seen by the generated code examples, the translations for program constructs like for loops are quite standard, so in the remainder of this section we concentrate discussing the most interesting and challenging issues for mapping Matlab to Fortran.

5.1 Mapping Types

In general, the mappings of types from Matlab to Fortran is listed in Table III. Besides these primitive data types, **Mc2for** also supports *cell arrays* in Matlab. We use *derived data types*[10] in Fortran to map Matlab *cell arrays*.

5.1.1 Variables with more than one dynamic type

Due to its dynamic nature, a variable in Matlab may hold different types at different program points (for example, variable x may be an integer at one place in a function, and a double at another place). Whereas in static languages, like Fortran, a variable must contain only the declared type. In **Mc2for**, we have a two-phase strategy to solve this problem. The first phase is the variable renaming phase achieved by analyzing the webs of definitions and uses of a variable, which is provided by the restructuring component Tamer+. If different webs for the same variable hold different types, then **Mc2for** creates renamed copies of the variable, one copy for each different type. The second phase is for the situation where a variable still may hold different types in the

[10]Also unknown as union types and similar to the structs in C/C++.

Table III: 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

same web of definitions and uses. In this case, **Mc2for** transforms this variable to a derived data type variable in FORTRAN. In the transformed derived data type, each field represents a different type of this variable in the original MATLAB program.

5.1.2 Implicit type conversion in MATLAB

Due to its weakly-typed language nature, the type of a variable can be implicitly converted in MATLAB. For example, although MATLAB requires that subscript indices must either be real positive integers or logicals, programmers are allowed to use indices in the type of double. While in FORTRAN, which is a strongly-typed language, the type of a variable cannot be automatically converted. To map this difference, **Mc2for** uses some FORTRAN intrinsic functions to force the type conversion, like `INT` and `DBLE`, in the situations where the type of the variable may be implicitly converted in MATLAB. For example, **Mc2for** will add `INT` around the variables used as matrix indices and variables or values used as start, end or increment in a range expression of a for loop statement.

5.2 Built-in Mapping Framework

To map MATLAB built-in functions to FORTRAN, we implemented a built-in mapping framework. In the framework, built-ins in MATLAB are mapped to FORTRAN in three different ways.

5.2.1 Directly-mapped

There are some MATLAB built-ins which can be mapped directly to equivalent intrinsic functions in FORTRAN, or we can also say, replaced directly by the equivalent intrinsic functions. The complete lists of these built-ins is given on the **Mc2for** web page.

5.2.2 Transform-then-inlined

For some built-ins, although they cannot be directly replaced with certain FORTRAN intrinsic functions, they can be transformed easily and inlined in the generated code, i.e., the colon and left division functions.

5.2.3 Not-directly-mapped

For most MATLAB built-ins, they cannot be directly mapped or easily transformed to equivalent intrinsic functions in FORTRAN. In order not to update **Mc2for** every time when there is a new MATLAB built-in introduced, we decided to use the following strategy to handle this mapping problem. The strategy is that we keep the function signature[11] in the generated FORTRAN code the same as in the original MATLAB program, then write an equivalent-functionality user-defined function in FORTRAN and add it into the standalone `libmc2for` library. [12] In other words, **Mc2for** leaves a "hole" for that not-directly-mapped MATLAB built-in inside the generated FORTRAN program and requires that there is a corresponding FORTRAN function in `libmc2for` to fill up the "hole" during compilation.

Almost all the MATLAB built-in functions are overloaded. In FORTRAN, intrinsic functions are also overloaded. Recall that the built-in mapping framework leaves the "hole" for those built-ins without directly-mappings in the transformed program and requires that there are FORTRAN functions with the same function signatures in `libmc2for` to fill up the "hole", so besides providing those FORTRAN functions, we should also make sure those functions are overloaded. Fortunately, we can overload user-defined functions with an interface since FORTRAN 90. Recall the built-in function `zeros` at line 10 in Listing 1. This function is always overloaded in MATLAB program as a storage preallocation for variables. A code snippet of the equivalent FORTRAN function in `libmc2for` for `zeros` is given in Listing 5. With this interface, the MATLAB built-in `zeros` can not only be supported, but also be supported with overloading features in the generated FORTRAN program.

```
MODULE mod_zeros

INTERFACE zeros
MODULE PROCEDURE zeros_1 , zeros_2 ! may be more
END INTERFACE zeros

CONTAINS

FUNCTION zeros_1(x)
IMPLICIT NONE
DOUBLE PRECISION, INTENT(IN) :: x
DOUBLE PRECISION, DIMENSION(INT(x),INT(x)) :: zeros_1
! default is 0, no need assignment.
END FUNCTION zeros_1

FUNCTION zeros_2(x,y)
IMPLICIT NONE
DOUBLE PRECISION, INTENT(IN) :: x, y
DOUBLE PRECISION, DIMENSION(INT(x),INT(y)) :: zeros_2
```

[11]The function name and the names of the input arguments.

[12]In the library shipped with **Mc2for**, we have already implemented some user-defined functions in FORTRAN to map some commonly-used but not-directly-mapped MATLAB built-ins, like `ones` and `zeros`.

```
! default is 0, no need assignment.
END FUNCTION zeros_2

END MODULE mod_zeros
```

Listing 5: `libmc2for` library function to map MATLAB `zeros`

5.3 Linear Indexing Transformation

In MATLAB, the programmer may leave out some of the trailing indices when accessing an element of a matrix. In this case, 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, which we call *rigorous array indexing*. **Mc2for** has a built-in component to transform linear indexing in MATLAB to rigorous array indexing in FORTRAN. For example, assuming that `arr` is a 3-by-3 matrix in MATLAB, matrix indexing in MATLAB `arr(5)` will be transformed to `arr(2,2)` in the generated FORTRAN program. Note that the prerequisite of this direct transformation is that both the indices and the shape of the accessed matrix are constants when **Mc2for** performs this transformation, if this prerequisite cannot be satisfied, **Mc2for** will call certain functions in `libmc2for` to map the indexing.[13]

6 Experiments and Result Analysis

In this section, we demonstrate some experiments to evaluate the performance of **Mc2for**. The set of benchmarks for the experiments was acquired from a variety of sources, most of them come from related projects, like FALCON [10] and OTTER projects [9], Chalmers University of Technology[14] and "The MathWorks' Central File Exchange"[15]. A brief description of the benchmarks is given here.

- *adpt* finds the adaptive quadrature using Simpson's rule. This benchmark features an array whose size cannot be predicted before compilation.
- *bubl* is the standard bubble sort algorithm. This benchmark contains nested loops and consists of many array read and write operations.
- *capr* computes 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.
- *clos* calculates the transitive closure of a directed graph. It contains matrix multiplication operations between two 450-by-450 arrays.
- *crni* computes the Crank-Nicholson solution to the heat equation. This benchmark involves some elementary scalar operations on a 2300-by-2300 array.
- *dich* computes the Dirichlet solution to Laplace's Equation. It's also a loop-based program which involves basic scalar operation on a small-sized array.
- *diff* calculates the diffraction pattern of monochromatic light through a transmission grating for two slits. This benchmark also features an array whose size is increased dynamically like the

[13]The naming convention of these functions and some function examples are given on **Mc2for** web page.
[14]http://www.elmagn.chalmers.se/courses/CEM/
[15]http://www.mathworks.com/matlabcentral/fileexchange

benchmark *adpt*.
- *fiff* computes the finite-difference solution to the wave equation. It's a loop-based program which involves basic scalar operation on a 2-dimensional array.
- *mbrt* computes a mandelbrot set with specified number elements and number of iterations. This benchmark contains elementary scalar operations on complex type data.
- *nb1d* simulates the gravitational movement of a set of objects. It involves computations on vectors inside nested loops.

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 and the generated FORTRAN code is compiled with the GFortran compiler of GCC version 4.6.3 using optimization level -O3.

In order to get a measurable execution time, we used a scale number[16] for each benchmark to adjust the problem size, including the number of iterations and the size of arrays, to make the program to run approximately 20 seconds under MATLAB. In Table IV, we list the execution time of different benchmarks under MATLAB and FORTRAN and we illustrate the speedup of the generated FORTRAN code over MATLAB for all the benchmarks in Figure 2.

The speedup of FORTRAN over MATLAB ranges from 3 to 26 times, except for the benchmark *clos*. The generated FORTRAN code for *clos* ran almost 28 times **slower** than the *clos* benchmark running in MATLAB. After examining the generated code, 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 replaced the call to MATMUL with a call to the DGEMM from the FORTRAN BLAS (Basic Linear Algebra Subprograms) library to make a new benchmark named *clos2*. The result is impressive, the compiled[17] program runs over 7 times faster than the *clos* benchmark running in MATLAB.

Table IV: Benchmarks' Execution Times in Seconds

Benchmarks	MATLAB	**Fortran**
adpt	20.08	3.4
bubl	20.48	1.3
capr	20.20	1.7
clos	20.62	569.7
clos2	20.62	2.9
crni	20.56	2.0
dich	20.10	6.8
diff	20.61	4.7
fiff	20.76	1.1
mbrt	20.85	3.3
nb1d	20.60	0.8

In Table V, we list the physical lines[18] of code (LOC) of both MATLAB benchmarks and the generated FORTRAN code from **Mc2for**. At the fourth column in the table, which is named

[16]The scale number for each benchmark is listed on **Mc2for** web page.

[17]In order to get the best performance, we statically link the subroutine from BLAS library when we compile the transformed program.

[18]Including whitespace and comment lines.

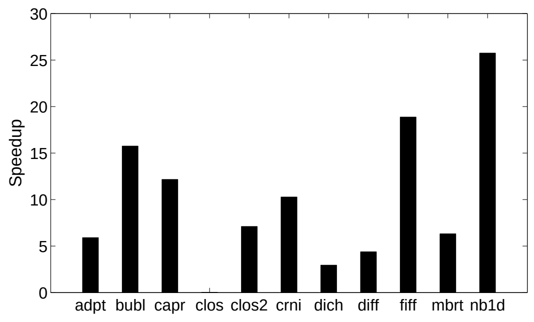

Figure 2: Speedup of Generated FORTRAN over MATLAB

"F/M", we also list the ratio of the LOC of FORTRAN to MATLAB.

In order to give a more intuitive feeling, in Figure 3, we put the benchmarks in the increasing order of their LOC along the x axis, and link their ratio of the LOC of FORTRAN to MATLAB. From this figure, we feel that with the size of the code growing bigger, the ratio of the LOC of FORTRAN to MATLAB goes to smaller and close to 1. This is reasonable, since our LOC also include the lines of variable declarations in the generated FORTRAN code, when the size of the MATLAB code is small, the lines of variable declarations take a lot of space in the generated FORTRAN code, while the program grows bigger and bigger, the number of variables keeps in a relatively stable number, so the lines of the statements will take the major numbers of the LOC, and the ratio of the LOC of generated FORTRAN to MATLAB goes down and gets closer and closer to 1. However, there are two exceptions, *diff* and *adpt*, in Figure 3. According to the description at the beginning of this section, we know that there are arrays whose sizes are increased or grew dynamically during the execution in those two benchmarks, which means, the generated FORTRAN code is inlined with a lot of run-time array bounds check and reallocation code.

Table V: LOC of Benchmarks

Benchmarks	MATLAB	**Fortran**	**F / M**
adpt	182	344	1.9
bubl	23	62	2.7
capr	217	360	1.7
clos	90	117	1.3
crni	198	281	1.4
dich	143	213	1.5
diff	127	204	1.6
fiff	116	158	1.4
mbrt	60	105	1.8
nb1d	179	311	1.7

In summary, the overall performance of the generated FORTRAN code from **Mc2for** for these benchmarks is better than the performance 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.

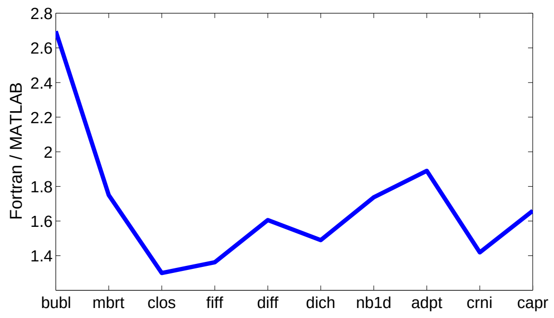

Figure 3: Code Size of Generated FORTRAN over MATLAB

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 [10] 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, with type inference in MATLAB and how to translate MATLAB to FORTRAN, **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 distinguish 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, with our approach being flexible and extensible, and capable of handling many features of modern MATLAB. 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 approaches to range analysis. The **Mc2for** approach is closest to a generalized constant propagation in C [11] 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 has its roots in the interval arithmetic.

Mc2for builds upon previous work in the McLab group. In early work, Jun Li developed a prototype which demonstrated the feasibility of translating MATLAB to FORTRAN 95 [7]. This early prototype focused on a limited subset of MATLAB and made simplifying assumptions. To provide a more solid analysis basis, the McSAF analysis framework [1, 2] and Tamer's extensible interprocedural abstract value analysis framework [4] were developed. Both of them have a different intermediate representations which are suitable for their own specific transformation and analysis. These two frameworks working together form the major transformation and analysis engine in the McLAB toolkit. Concurrent to our development of **Mc2for**, our lab is also working on another project to statically compile MATLAB to X10 [6], which also uses the shape analysis in **Mc2for**.

MATLAB Coder^{TM} [8] is a commercial translator to generate standalone C and C++ code from

MATLAB. MATLAB Coder supports a subset of core MATLAB language features, including program control constructs, functions, and matrix operations. This is a closed source system, with no research papers on its design. Part of the objective of our work is to provide an open source framework, which other researchers can easily use. For example, the McLab toolkit, plus the shape and range analysis presented in this paper would be a suitable starting point for developing a C/C++ back end.

8 Conclusion and Future Work

In this paper, we have presented a tool which can automatically transform MATLAB programs to equivalent FORTRAN programs. Since MATLAB is a dynamic and weakly-typed programming language, while FORTRAN is a static and strongly-typed language, there are quite a number of challenges, such as how to collect type information for variable declaration, how to map the numerous MATLAB built-in functions to FORTRAN, how to make FORTRAN support the features of the matrix growth and matrix linear indexing in MATLAB, and how to eliminate unnecessary run-time array bounds checking.

In our **Mc2for**, we introduced a shape analysis, 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 designed and implemented a range value analysis, 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 using the Tamer's extensible interprocedural abstract value analysis framework. In the code generation framework of **Mc2for**, 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 linear indexing transformation from MATLAB to FORTRAN.

Finally, in Section 6, we evaluated **Mc2for** on a collection of MATLAB benchmarks, examining both the speedup and physical lines of code of the transformed code. From the results, we show that the code generated by **Mc2for** is usually more efficient than the original MATLAB code and the code size is quite acceptable.

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, adding a MATLAB storage analysis, which can determine when the default double type can be safely stored in integers, may further improve the code readability and save quite a lot storage. In the future, 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`.

Acknowledgments

We would like to thank all the members of the McLab team, in particular Jun Li who developed the first prototype, Anton Dubrau who developed Tamer and helped start this project, and Amine Sahibi who developed Tamer+. This work is supported, in part, by NSERC.

References

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

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

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

[4] Anton Dubrau and Laurie Hendren. Taming MATLAB. In *Proceedings of OOPSLA 2012*, pages 503–522, 2012.

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

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

[7] Jun Li. McFor: A MATLAB to FORTRAN 95 Compiler. Master's thesis, McGill University, August 2009.

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

[9] Michael J. Quinn, Alexey Malishevsky, Nagajagadeswar Seelam, and Yan Zhao. Preliminary Results from a Parallel MATLAB Compiler. In *Proceedings of Int. Parallel Processing Symp.*, IPPS, pages 81–87, 1998.

[10] 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.

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