Mc2FOR: A Tool for Automatically Translating MATLAB to FORTRAN 95

Xu Li and Laurie Hendren
School of Computer Science, McGill University

***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 prototyping, programmers often prefer using high-performance static languages such as FORTRAN for their final distributable code. Rather than rewriting the code by hand, our solution is to provide a tool that automatically translates the original MATLAB program to 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 equivalent 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 the shape of arrays and the range of 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 FORTRAN. This work has been implemented within the McLab framework, and we demonstrate the performance of the translated FORTRAN code on a collection of MATLAB benchmarks.

I. INTRODUCTION

MATLAB 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 [1]. 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 tool that automatically converts MATLAB programs to FORTRAN. 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 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 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 implemented a custom range analysis for MATLAB scalar variables 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

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

tool on a suite of benchmarks, showing that we can produce efficient and compact code.

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

II. 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 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. Imagine that we want to transform this MATLAB implementation to FORTRAN- what potential problems we may encounter?

```
1  function z_hat = babai(R,y)
2  %%
3  %   compute the Babai estimation
4  %   find a sub-optimal solution for min_z ||R*z-y||_2
5  %   R - an upper triangular real matrix of n-by-n
6  %   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 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 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, let's 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 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.

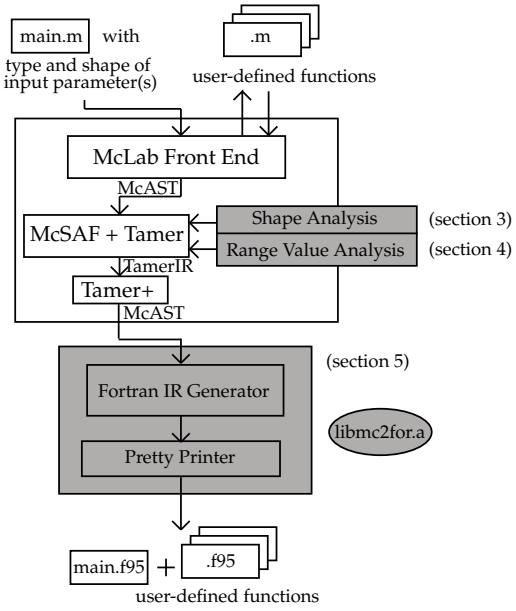

Fig. 1. The Overview of Mc2FOR. We highlight the boxes which are the contributions of this paper.

Now let us concentrate on the actual structural organization of Mc2FOR. The central component driving the compilation process is the Tamer module [2]. 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 intermediate representation (IR), McAST. The analysis and transformation engine, McSAF [3] is then used to transform to a lower-level AST;

and to perform initial analyses such as *kind analysis* [4], which determines 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'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 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 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 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 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 FORTRAN 95-compliant compiler (including the open source GFortran). Further, as we show in Section VI, the resulting FORTRAN code is often significantly more efficient than the original MATLAB code.

III. 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 in the Tamer's extensible interprocedural abstract value analysis framework. The Tamer's 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 MATLAB built-in functions, and (3) a merging operator that merges two abstract shapes.

We abstract shapes by lists of dimensions, where each dimension is either an integer value, a symbolic value represented by a lowercase letter, or the special character `?` which represents an unknown value. For example, if `arr` is a 3-by-5 array, we would abstract its shape as `[3,5]`; the shape `[2,?]` represents a two-dimensional array, where the first dimension is 2, and the second dimension is unknown.

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`? Similar problems occur again at line 10 for the built-in function `zeros` and at lines 11 and 16 for the built-in function `round`.

Our solution to this built-in challenge was to design a concise domain-specific language that is used to describe the shape propagation behaviour of MATLAB built-ins. To design a language covering all the typical behaviours, we studied hundreds of built-ins and categorized them as follows:

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.[3]

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(s). 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.

[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.

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

Other cases: The above three 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.

Based on these behaviours we defined the *Shape Propagation Equation Language* (SPEL), which can be used to write a *shape propagation equation* (SPE) for each MATLAB built-in. We have also designed and implemented 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]`.

We now 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.

a) 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 ORORs (`||`) 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.

b) 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.

```
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.

c) 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 programmer. Mc2FOR will throw a warning to the user.

d) PATTERN EXPRESSION: Pattern expressions can be either primitive pattern expressions or compound pattern expressions. Primitive 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 is 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 output preparation during the shape propagation process.

Shape matching expression (SME) : There are four kinds:

- `$` matches scalars (1-by-1 arrays);
- upper-case letters match matrices which are not scalars;
- *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]`, and which impose more restrictions on the number of dimensions or/and the size of certain dimension;
- # is the wildcard symbol to match any shape.

Helper function: 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.

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 functions. 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.

e) SHAPE OUTPUT LIST SIDE: The shape output list side contains a list of only shape matching expressions, specifying the shape information of the output.

```
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.

f) OPERATOR: SPEL also supports several standard notations, with their usual meanings. Compound pattern expressions are grouped together using `()`. The `?`, `*`, and `+` operators can be used after a SME or a compound pattern expression, and have the usual regular expression meanings.

The | operator can be used to denote choice between two possible shape matching expressions.

Now let's 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
```

Recall that the Tamer's framework takes care of propagating the abstract values, applying the rules for the built-ins, and propagating values through assignment statements. It also handles the control flow for conditional constructs such as `if-else` and loop constructs such as `for` and `while`. For these control constructs, we need to provide a merging operator, which defines how to merge abstract shapes coming from two different control flow paths.

The high-level merging strategy for shapes is given in Table I. There are three different 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: 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 shape propagation through built-ins fails, which corresponds to cases where programmers misuse 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 iterations,[4] we push the shape to `unmergeable`. To generate FORTRAN code, at least the number of dimensions of each variable must be known, thus our compiler cannot generate code for the input programs in which the variables have `not_matched` or `unmergeable` shapes.

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

IV. 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 added columns/rows 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 array bounds checking in the generated FORTRAN. 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 also 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 checking, we extended Tamer's constant value analysis to a *range value analysis*, which statically estimates the minimum and maximum values each scalar variable[5] could take at each program point. Similar to the shape analysis, the range value analysis was also implemented in the Tamer's interprocedural 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 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 MATLAB `if-else` constructs, we add two special superscript

[4]Other iteration numbers will also work with our approach, empirically we found 5 to be a good setting.

[5]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.

symbols, + and -, for instance, 5^+ and 5^-. You can interpret these two superscripted real numbers as 5+ϵ and 5-ϵ, 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`, a real number, or a real number with +; and the upper bound in a range value can only be one of `+inf`, a real number, or a real number with -.

Some most commonly-used MATLAB built-in scalar operators supported by our range value analysis is 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 symbolics 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. To support the range value propagation functions, we have defined a set of arithmetic operators 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:[6]

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

Listing 2. Range value propagation function for the binary plus operator (+)

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 the range values from different iterations of loop statements cannot be merged to a fixed point after 5 iterations, the range values of the bounds will be pushed to `-inf` or `+inf` respectively.

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

V. 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 the equivalent FORTRAN. 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 algorithm, 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 *nocheck* 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.

```
1  MODULE mod_babai
2  CONTAINS
3  SUBROUTINE babai(R,y,z_hat)
4  USE mod_zeros
5  IMPLICIT NONE
6  DOUBLE PRECISION , DIMENSION(:,:) , ALLOCATABLE
7                                        :: z_hat, R, y
8  DOUBLE PRECISION :: par, n, ck
9  INTEGER(KIND=4) :: k
10 ! compute the Babai estimation
11 ! find a sub-optimal solution for min_z ||R*z-y||_2
12 ! R - an upper triangular real matrix of n-by-n
13 ! y - a real vector of n-by-1
14 ! z_hat - resulting integer vector
15 n = SIZE(y);
16 z_hat = zeros(n, 1);
17 z_hat(INT(n), 1) = NINT((y(INT(n), 1)
18                          / R(INT(n), INT(n))));
19 DO k = INT((n - 1)), 1, -1
20    par = DOT_PRODUCT(R(k, INT((k + 1)):INT(n)),
21                      z_hat(INT((k + 1)):INT(n), 1));
22    ck = ((y(k, 1) - par) / R(k, k));
23    z_hat(k, 1) = NINT(ck);
24 ENDDO
25 END SUBROUTINE
26 END MODULE
```

Listing 3. Generated FORTRAN for the Babai example with *nocheck* 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 in MATLAB.

If we run Mc2FOR on this example without the `nocheck` 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 for the statement at line 20.

```
! 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 checking 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 algorithm to solve a problem in wireless communications, the shape of `R` and `y` is double the number of antennas in a multiple-input and multiple-output 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, y
DOUBLE PRECISION :: par, n, ck
DOUBLE PRECISION , DIMENSION(10,10) :: R
INTEGER(KIND=4) :: k
!   ... the comment is the same as previous example
n = SIZE(y);
z_hat = zeros(n, 1);
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, INT((k + 1)):INT(n)),
                     z_hat(INT((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. Generated FORTRAN for the Babai example with specific dimensions given for input parameters

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 on discussing the most interesting and challenging issues for mapping MATLAB to FORTRAN.

A. 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*[7] in FORTRAN to map MATLAB *cell arrays*.

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

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, or sometimes after merging the data flow from different branches, a variable may also hold different types at the same program point. 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 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.

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`, for the situation 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.

B. 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.

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

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

equivalent intrinsic functions or operators. The complete lists of these built-ins is given on the Mc2FOR web page.

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 `mean` functions.

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[8] 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. [9] 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.

```
1  MODULE mod_zeros
2  INTERFACE zeros
3  MODULE PROCEDURE zeros_1, zeros_2 ! may be more
4  END INTERFACE zeros
5  CONTAINS
6  FUNCTION zeros_1(x)
7  IMPLICIT NONE
8  DOUBLE PRECISION, INTENT(IN) :: x
9  DOUBLE PRECISION, DIMENSION(INT(x),INT(x)) :: zeros_1
10 ! default is 0, no need assignment.
11 END FUNCTION zeros_1
12 FUNCTION zeros_2(x,y)
13 IMPLICIT NONE
14 DOUBLE PRECISION, INTENT(IN) :: x, y
15 DOUBLE PRECISION, DIMENSION(INT(x),INT(y)) :: zeros_2
16 ! default is 0, no need assignment.
17 END FUNCTION zeros_2
18 END MODULE mod_zeros
```

Listing 5. User-defined function in `libmc2for` to map MATLAB `zeros`

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

[9]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`.

C. 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, if this prerequisite cannot be satisfied, Mc2FOR will call certain functions in `libmc2for` to transform the indexing.[10]

VI. Experiments and Result Analysis

At last, we demonstrate some experiments to evaluate both the correctness and performance of Mc2FOR. The set of the benchmarks for the experiments was acquired from a variety of sources, most of them come from related projects, like FALCON [5] and OTTER projects [6], Chalmers University of Technology[11] and "The MathWorks' Central File Exchange"[12]. In general, the subset of MATLAB features supported by Mc2FOR includes commonly used numericial computations; standard constructs such as `if-else`, `for` loop and `while` loop statements; run-time array bounds checking; run-time array growth by out-of-bound array indexing; variable re-definition with different type values; and built-in function overloading. All the features in the benchmarks are covered by this subset. 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.
- *bbai* uses Babai algorithm to compute on fixed-sized arrays.
- *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 loop-based and involves 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.

[10]The naming convention of these functions and some function examples are given on Mc2FOR web page.

[11]http://www.elmagn.chalmers.se/courses/CEM/

[12]http://www.mathworks.com/matlabcentral/fileexchange

- *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 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 computes 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.

Before comparing the performance of the benchmarks and the generated FORTRAN, we first make sure that the generated FORTRAN programs have the same functionality and produce the same result as its input benchmark. Then, in order to get a measurable execution time, we used a scale number[13] 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. We use the same scale numbers for the generated FORTRAN programs to ensure they run on the same problem size as the MATLAB benchmarks. In Table IV, we list the execution time of different benchmarks under MATLAB and FORTRAN.

The speedup of FORTRAN over MATLAB ranges from 3 to 34 times, except for the benchmark *clos*. The generated FORTRAN code for *clos* ran about 24 times **slower** than the *clos* benchmark running in MATLAB. After performing more experiments, 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 compiled program runs about 3.5 times **slower** than the *clos* benchmark running in MATLAB. A reasonable explanation is that according to one document on the website of Intel[14], at least since MATLAB R2010a, MATLAB uses the Intel MKL BLAS by default, while the BLAS library we use in the experiments is the default BLAS for Ubuntu, and Intel MKL BLAS may have a better implementation than the one default for Ubuntu.

In Table V, we list the physical lines[15] of code (LOC) of both MATLAB benchmarks and the generated FORTRAN code with nochecks from Mc2FOR. At the fourth column in the table, which is named "F / M", we also list the ratio of the LOC of FORTRAN to MATLAB. The ratio ranges from 1.2 to 2.6, which seems quite reasonable since the generated FORTRAN code requires lines for the variable declarations and run-time array reallocation. Very short benchmarks like *bubl* have a larger ratio because the number of lines of declarations is large as compared to the small number of statements.

TABLE IV
PERFORMANCE COMPARISON

Benchmarks	MATLAB (s)	**FORTRAN** (s)	**Speedup**
adpt	20.08	3.4	5.9
bbai	20.58	0.6	34.3
bubl	20.48	1.3	15.7
capr	20.20	1.7	12.1
clos	20.62	490.5	0.04
clos2	20.62	68.7	0.3
crni	20.56	2.0	10.2
dich	20.10	6.8	2.9
diff	20.61	4.7	4.3
fiff	20.76	1.1	18.8
mbrt	20.85	3.3	6.3
nb1d	20.60	0.8	25.7

TABLE V
LOC COMPARISON

Benchmarks	MATLAB	**FORTRAN**	**F / M**
adpt	169	301	1.7
bbai	26	59	2.3
bubl	23	60	2.6
capr	206	342	1.7
clos	78	157	2.0
crni	192	234	1.2
dich	131	156	1.2
diff	115	148	1.3
fiff	104	135	1.3
mbrt	43	97	2.3
nb1d	167	261	1.6

In summary, the overall performance of the generated FORTRAN by Mc2FOR is better than the performance of the input benchmarks running in MATLAB, and according to the comparison of the LOCs, the size of the generated FORTRAN code is in an acceptable range.

VII. 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 [5] 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

[13]The scale number for each benchmark is listed on Mc2FOR web page.
[14]http://software.intel.com/en-us/articles/using-intel-mkl-with-matlab
[15]Including whitespace and comment lines.

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 [7] 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 McLab group. In early work, Jun Li developed a prototype which demonstrated the feasibility of translating MATLAB to FORTRAN 95 [8]. 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 [3], [9] and Tamer's extensible interprocedural abstract value analysis framework [2] were developed. 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 [10], which also uses the shape analysis in Mc2FOR.

MATLAB Coder™ [11] 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. 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.

VIII. 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 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, 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, 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, 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 VI, we evaluated our 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 starting this project, and Amine Sahibi who developed Tamer+. This work is supported, in part, by NSERC.

References

[1] GNU, "GNU Fortran Home Page," 2013, http://gcc.gnu.org/fortran/.
[2] A. Dubrau and L. Hendren, "Taming MATLAB," in *Proceedings of OOPSLA 2012*, 2012, pp. 503–522.
[3] J. Doherty and L. Hendren, "McSAF: A Static Analysis Framework for MATLAB," in *Proceedings of ECOOP 2012*, 2012, pp. 132–155.
[4] J. Doherty, L. Hendren, and S. Radpour, "Kind Analysis for MATLAB," in *In Proceedings of OOPSLA 2011*, 2011, pp. 99–118.
[5] L. D. Rose and D. Padua, "Techniques for the Translation of MATLAB Programs into Fortran 90," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 2, pp. 286–323, 1999.
[6] M. J. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao, "Preliminary Results from a Parallel MATLAB Compiler," in *Proceedings of Int. Parallel Processing Symp.*, ser. IPPS, 1998, pp. 81–87.
[7] C. Verbrugge, P. Co, and L. Hendren, "Generalized Constant Propagation A Study in C," in *Proceedings of the 1996 International Conference on Compiler Construction*, ser. CC '96, Linkoping, Sweden, 1996.
[8] J. Li, "McFor: A MATLAB to FORTRAN 95 Compiler," Master's thesis, McGill University, August 2009. [Online]. Available: http://www.sable.mcgill.ca/mclab/matlab_fortran.html
[9] J. Doherty, "McSAF: An Extensible Static Analysis Framework for the MATLAB Language," Master's thesis, McGill University, December 2011.
[10] V. Kumar and L. Hendren, "First Steps to Compile MATLAB to X10," in *Proceedings of the third ACM SIGPLAN X10 Workshop*, Seattle, USA, 2013, pp. 2–11.
[11] MathWorks, " MATLAB Coder," http://www.mathworks.com/products/matlab-coder/.