Partial Evaluation of MATLAB

Daniel Elphick[1], Michael Leuschel[1] and Simon Cox[2]

[1] Department of Electronics and Computer Science
[2] School of Engineering Sciences
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{dre00r,mal}@ecs.soton.ac.uk, sjc@soton.ac.uk

Abstract. We describe the problems associated with the creation of high performance code for mathematical computations. We discuss the advantages and disadvantages of using a high level language like MATLAB and then propose partial evaluation as a way of lessening the disadvantages at little cost. We then go on to describe the design of a partial evaluator for MATLAB and present results showing what performance increases can be achieved and the circumstances in which partial evaluation can provide these.

1 Introduction

Scientific computing is moving away from hand-coded, hand-optimised codes, written in low to medium level languages like FORTRAN, towards codes written in a general way in high-level programming languages or using problem solving environments, e.g. MATLAB, Maple, MATHEMATICA.

With the current diversity of computer architectures (due to differences in cache configurations, speeds and memory bandwidths as well as processor types), it is becoming increasingly difficult to produce near optimal code without expending excessive time. Even then the code ties the developer to one architecture, which is no longer ideal in an era of heterogeneous systems. One effort to tackle this problem has been the ATLAS project, which produces a BLAS (Basic Linear Algebra Subroutines), automatically tuned to the architecture on which it is being used [21]. MATLAB itself uses ATLAS for its linear algebra operations.

The use of high level languages allows rapid prototyping to ease development. Producing optimal code in the initial stage is not so important as actually producing something which solves the problem. This often leads to general solutions which can be applied to a wide variety of problems albeit slowly due to their generality.

One way to address these issues is to transform general programs written (or produced by other programs) in high level languages such as MATLAB into specialised programs that executes more quickly using partial evaluation and other high level code optimisations. To this end we have developed a partial evaluator called MPE (MATLAB Partial Evaluator).

In Sect. 2, we examine existing work on partial evaluation as well as work on improving MATLAB program performance using parallelisation and compilation. In Sect. 3, we examine the MATLAB programming language, looking at its grammar and performance issues. Sect. 4 presents a formal analysis of the type system used in MPE. The design and implementation details of MPE are given in Sect. 5. In Sect. 6, we demonstrate the effectiveness of automatic partial evaluation by applying our tool to several test programs and comparing timings. Sect. 7 gives our conclusions based on the results and also describes future work that could enable further improvements.

2 Optimising MATLAB

MATLAB is a problem solving environment sold by The Mathworks and currently used by around 500,000 people around the world [20]. It is controlled using a language also known as MATLAB. From now on when we refer to MATLAB, we are referring to the language unless otherwise stated. MATLAB is a dynamically typed imperative language which is normally interpreted. Variables do not need to be declared and can change type. Matrices and arrays are not of fixed size but are reshaped when assignments are made to subscripts outside the current bounds.

The Mathworks provides a compiler called MCC, which translates MATLAB into C. This C code is then compiled by the native compiler to produce an executable. The code produced consists mostly of function calls and very little attempt is made to use native C types, as dynamic typing means that a variable can contain anything from a matrix to a function handle.

Another compiler is FALCON, which produces Fortran 90 code [17]. This uses extensive type inferencing at compile time to produce code which does very little type checking. Initial results showed FALCON outperforming MCC, but no recent comparisons have been reported by its authors.

Following on from FALCON, Almási has developed MaJIC, a MATLAB *just-in-time* compiler as part of "an interactive frontend that looks like MATLAB and compiles/optimises code behind the scenes in real time, employing a combination of just-in-time and speculative ahead-of-time compilation." [1] Because MaJIC compiles code in an interpreted environment, it has information about the parameters used to call functions and attempts to produce more appropriate code. When compiling *just-in-time* it eschews most optimisations in favour of fast compilation times and so cannot easily perform the types of aggressive optimisation seen in offline compilers and partial evaluators. MATLAB 6.5 has also recently introduced *just-in-time* compilation as part of its normal operation, although we have not yet examined its effectiveness.

An example of work on array shape determination is MAGICA, which "takes a MATLAB program as input and statically infers each variable's value range, intrinsic type and array shape." [11] Unlike the research compilers discussed above, MAGICA can handle multi-dimensional arrays, although its approach is different to ours.

Other approaches to speeding up MATLAB execution have involved parallelisation. Mostly this involves adding parallels extensions to the language, like MultiMATLAB [12] or the DP toolbox [15]. However Otter is an attempt to translate MATLAB scripts into C programs targeting parallel computers supporting ScaLAPACK [16]. This approach produces varying results depending on the sizes of matrices and the complexity of the operations performed on them.

Partial Evaluation "Partial evaluation is a technique to partially execute a program, when only some of its input data are available." [13] In other words, to take a program, for which some of the inputs are known prior to full execution, and execute as much of the program as possible. In cases where programs are executed many times with few parameters changing, dramatic savings can be made as many calculations are performed just once during partial evaluation. Partial evaluators normally perform aggressive optimisations like loop unrolling and inlining which, while also possible in traditional compilers, are less easy to control or see the effects of when the transformation is not source to source. A program generated by partial evaluation is called a *residual* program.

Traditionally partial evaluation has been mostly applied to declarative languages, like Scheme [7] or Prolog [10]. But there are also partial evaluators for C [3], Java [19] and Fortran [8].

We believe that partial evaluation has a potential for scientific applications in general and MATLAB code in particular. For example, in some of our MATLAB code to solve partial differential equations, the same mesh structure is used over and over again, and we believe that we might gain speedups by partially evaluating the code for a given mesh structure.

There are two main forms of partial evaluation, online and offline. In *offline* partial evaluation, *binding-time analysis* is performed first which parses the input and determines which parts are static and which parts are dynamic. This data is then embedded in the source file in the form of annotations which are used by the partial evaluator to produce the final result. In *online* partial evaluation, there is no binding-time analysis step but instead decisions about static vs. dynamic expressions are made as late as possible, and it is thus, in principle, more precise. In general, offline partial evaluators can be made more efficient and predictable while online partial evaluators are typically slower but more precise.

Due to the non-static nature of MATLAB (no static types, types of variables can change) and the presence of complicated elementary operations (e.g., multiplication of multi-dimensional matrices), we have chosen to use the online approach.

3 Overview of MATLAB

For the purposes of this project we intend only to look at a subset of MATLAB. In this section the structure of MATLAB will be discussed in order to give a better understanding of the later sections.

MATLAB code is always stored in a file with a `.m` extension, called an m-file. An m-file is either a script or a function, depending on whether the file starts with the `function` keyword. Scripts are executed within the current scope whereas functions execute within their own scope.

Functions are declared using the `function` keyword. They can return zero or more values and take zero or more parameters. We have chosen to exclude functions that can take a variable number of parameters. Any values stored in the return variables at the end of the execution of the function are returned. E.g.

```
function [a,b] = f(x,y)
```

MATLAB statements are either separated by new lines or semi-colons, although in matrices, these also delimit rows. A MATLAB statement must be one of the following: an expression, an assignment, an `if` statement, a `for` loop, a `while` loop, a `switch` block, or a `global` variable declaration. Note that we can have a multi-value assignment for use only with functions which return multiple values. To simplify matters, we require that global variable declarations come immediately after the function declaration and cannot appear later.

MATLAB allows the creation of complex expressions in a very intuitive way. The basic construct is the array, of which matrices, vectors and scalars are special cases. The only other data type we consider is the string. Apart from standard matrices which are two dimensional, MATLAB also has n-dimensional arrays. Matrices can be either real, complex or logical. Logical arrays are returned from boolean operators and built-in functions like `isreal` and `isinf`. While generally they will have the values 0 or 1, they can have any real value.

It is possible to index into a matrix using more dimensions than the matrix has, as long as the extra indices are equal to 1. If fewer dimensions are used then the dimensions that are not explicitly specified are flattened, so that the final index can be used to access all of them. For example matrices can be indexed linearly.

MATLAB allows more than just scalars as indices; any appropriately sized matrix can be used. In particular, ranges can be used to extract parts of matrices. Indices start at 1 and `end` can be used to get the last element along a particular dimension. If the index is '`:`', it indicates all elements along a given dimension should be extracted. Finally, if an index is logical it predicates which part of the matrix should be extracted. Below, for example, the first row of `a` is retrieved and stored in `b` and then all elements in `b` not equal to 2 are displayed.

```
>> a = [1 2 3; 4 5 6; 7 8 9];
>> b = a(1,:)
b =      1      2      3
>> b(b ~= 2)
ans =      1      3
```

There are several MATLAB features that we do not yet consider, such as function handles, `try-catch` statements, cell arrays and persistent variables. By limiting the set of MATLAB features that we can handle we limit the number of

MATLAB programs with which we can initially work. However we have to make a pragmatic decision to ignore certain features that are not critical to testing our hypothesis that partial evaluation is a viable technique for the optimisation of MATLAB programs. In the future, it is hoped that these features could also be added to our tool.

4 Abstract Domains

MATLAB has a complicated type system which has evolved over time from just representing two dimensional matrices to N-dimensional arrays, cell arrays, structures, strings and function handles. We have chosen to exclude cell arrays, structures and function handles in the current work and just handle arrays (including matrices). In this section, we formalise the abstract domains our partial evaluator uses to capture information about arrays. The notation and methodology is based on [1] and [4].

4.1 Abstract Type System

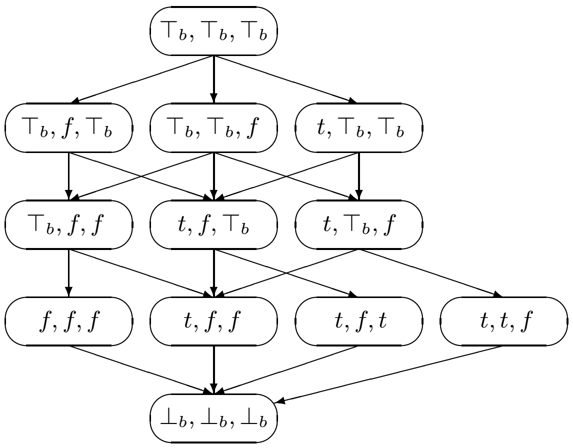
A MATLAB array can have the following types: *real*, *complex*, *logical* or *character*. These types are not exclusive, as *logical* and *character* arrays are also real. There is no single `gettype` function, but there are various boolean functions that determine the properties of an array: `isreal`, `islogical` and `ischar`. An array of type *real* is made up from double precision floats. A *logical* array is identical to a *real* array except that it has a flag indicating that it is logical. *Complex* arrays use twice the memory of real ones, in order to store the real and imaginary components. *Character* arrays are just like arrays except that the elements are two byte characters. Some example types are shown in Fig. 1.

```
>> [isreal([1 pi]), islogical([1 pi]), ischar([1 pi])]
ans =     1     0     0
>> [isreal(3 == 2), islogical(3 == 2), ischar(3 == 2)]
ans =     1     1     0
>> [isreal('abc'), islogical('abc'), ischar('abc')]
ans =     1     0     1
>> [isreal(3 + 2i), islogical(3 + 2i), ischar(3 + 2i)]
ans =     0     0     0
```

Fig. 1. Examining MATLAB types (Non-zero values indicate true)

We cannot always reliably make assumptions about types in MATLAB for several reasons. For one, it is possible to add two complex numbers together and get a real number as the result. A compiler would almost certainly assume the result was complex and allocate enough storage for that. MATLAB allocates the extra space only when it determines that the result is actually complex. It is also possible to force the creation of a complex type with no imaginary component using the `complex` built-in. The `isreal` built-in indicates whether storage has been allocated for an imaginary component, not whether it is really real.

To handle these types, an inclusive type system is required, which indicates what possible types an array might have. This would have three trinary flags, which indicate *true*, *false* or *unknown*. The *unknown* value is actually equivalent to $\top_b$ in a lattice and so we shall write that from now on. These values all form the set $\mathbf{B}$, given in Definition 1. There is also nominally another value, $\bot_b$, which denotes an *invalid* type. The three flags are *real*, *logical* and *character*. While in theory this would allow 3^3 possible types, in practice the value of one flag can dictate the value of another flag, as for instance logical arrays cannot be complex. There are in fact only 12 valid combinations including the *invalid* type, which are shown in Fig. 2.



$$\begin{tikzcd}
 & \top_b, \top_b, \top_b \arrow[dl] \arrow[d] \arrow[dr] & & \\
\top_b, f, \top_b \arrow[d] \arrow[dr] & \top_b, \top_b, f \arrow[dl] \arrow[d] \arrow[dr] & t, \top_b, \top_b \arrow[dl] \arrow[d] & \\
\top_b, f, f \arrow[d] \arrow[dr] & t, f, \top_b \arrow[d] \arrow[dr] & t, \top_b, f \arrow[dl] \arrow[dr] & \\
f, f, f \arrow[dr] & t, f, f \arrow[d] & t, f, t \arrow[dl] & t, t, f \arrow[dll] \\
 & \bot_b, \bot_b, \bot_b & &
\end{tikzcd}$$

Fig. 2. Visualisation of the type lattice $\mathfrak{L}_t$, of valid triples $(\mathit{real}, \mathit{logical}, \mathit{complex}) \in \mathbf{B} \times \mathbf{B} \times \mathbf{B}$, where the arrows indicate the partial ordering

Definition 1. *$\mathbf{B} = \{\top_b, \mathit{true}, \mathit{false}, \bot_b\}$ is the extended boolean type, with an associated partial order $\sqsubseteq_b$ defined by $\bot_b \sqsubseteq_b \mathit{false} \sqsubseteq_b \top_b$ and $\bot_b \sqsubseteq_b \mathit{true} \sqsubseteq_b \top_b$. $\mathbf{T} \subset \mathbf{B} \times \mathbf{B} \times \mathbf{B}$ is the following set of all valid triples of $\mathbf{B}$:*

$$\begin{aligned}
\mathbf{T} = \big\{ &\langle \top_b, \top_b, \top_b \rangle, \langle \top_b, \mathit{false}, \top_b \rangle, \langle \top_b, \top_b, \mathit{false} \rangle, \\
&\langle \top_b, \mathit{false}, \mathit{false} \rangle, \langle \mathit{false}, \mathit{false}, \mathit{false} \rangle, \langle \mathit{true}, \top_b, \top_b \rangle, \\
&\langle \mathit{true}, \top_b, \mathit{false} \rangle, \langle \mathit{true}, \mathit{false}, \top_b \rangle, \langle \mathit{true}, \mathit{false}, \mathit{false} \rangle, \\
&\langle \mathit{true}, \mathit{false}, \mathit{true} \rangle, \langle \mathit{true}, \mathit{true}, \mathit{false} \rangle, \langle \bot_b, \bot_b, \bot_b \rangle \big\}
\end{aligned}$$

The partial order $\sqsupseteq_t$ associated with $\mathbf{T}$ is depicted in Fig. 2. Finally we define the lattice $\mathfrak{L}_t = (\mathbf{T}, \bot_t, \top_t, \sqsubseteq_t, \sqcup_t)$ where $\top_t = \langle \top_b, \top_b, \top_b \rangle$, $\bot_t = \langle \bot_b, \bot_b, \bot_b \rangle$, and where $\sqcup_t$ and $\sqcap_t$ are defined in the usual way (cf., Fig. 2).

4.2 Dimension Information

The above abstract domain captures the types of arrays. As matrix manipulations are the backbone of MATLAB, to enable advanced optimisations, we need to capture abstract information about the shape of matrices. As loops are frequently controlled by the size of a matrix dimension, knowing the matrix shape can allow loop unrolling.

Arrays have a number of dimensions, which is always greater than or equal to 2 and is returned by the `ndims` function. Each dimension then has a value greater than or equal to 0. The `size` function us used to get the dimension sizes. Requesting a dimension beyond the number of dimensions always returns 1. The shape of an array is defined initially in Definition 4 and Definition 5.

Definition 2. *We define the extended set of non-negative integers $\mathbb{Z}^\omega = \mathbb{Z}^* \cup \{\omega\}$, We extend the ordering $<$ on $\mathbb{Z}^\omega$ by stating $\forall n \in \mathbb{Z}^\omega, n \leq \omega$ and also $\omega \leq n \Rightarrow n = \omega$.*

Definition 3. *A* range *is a tuple $\langle l, u \rangle$, where $l, u \in \mathbb{Z}^\omega$ and $l \leq u$. l is called the lower bound and u the upper bound of the range. We define the set $\mathbf{R}$ to contain all possible ranges with the addition of the $\bot_r$ element to indicate an invalid range ($\mathbf{R} \subset \mathbb{Z}^\omega \times \mathbb{Z}^\omega \cup \{\bot_r\}$). The top element is the least constrained range possible, i.e. $\top_r = \langle 0, \omega \rangle$. We define two functions: low$(\langle l, u \rangle) = l$ and up$(\langle l, u \rangle) = u$. We also define* join *($\sqcup$) and* meet *($\sqcap$) operations on this set.*

$$\begin{array}{rcl} \langle l_1, u_1 \rangle \sqsubseteq_r \langle l_2, u_2 \rangle & \iff & l_1 \geq l_2 \text{ and } u_1 \leq u_2 \\ \langle l_1, u_1 \rangle \sqcup_r \langle l_2, u_2 \rangle & = & \big\langle \min(l_1, l_2), \max(u_1, u_2) \big\rangle \\ \langle l_1, u_1 \rangle \sqcap_r \langle l_2, u_2 \rangle & = & \begin{cases} \big\langle \max(l_1, l_2), \min(u_1, u_2) \big\rangle & \text{if } l_1 \leq u_2 \wedge l_2 \leq u_1 \\ \bot_r & \text{otherwise} \end{cases} \end{array}$$

We use ranges to represent the possible values a shape characteristic might have. Note that ω is used for ranges without an upper bound.

Definition 4. *The number of dimensions is a range (Definition 3), except that the minimum value is 2, as an array always has at least 2 dimensions. This is given by the set, $\mathbf{N} = \mathbf{R} - \big\{ \langle i, j \rangle \mid i \in \{0, 1\}, j \in \mathbb{Z}^\omega \big\}$. The top element in $\mathbf{N}$ is $\top_n = \langle 2, \omega \rangle$. The partial ordering $\sqsubseteq_n$ and operations $\sqcap_n$ and $\sqcap_r$ are equivalent to $\sqsubseteq_r$, $\sqcap_r$ and $\sqcup_r$ respectively.*

Definition 5. *The list of dimensions is a sequence of ranges, $\langle r_1, r_2, \ldots, r_n \rangle \in \mathbf{D}$, where $\mathbf{D} = \mathbf{R}^*$. We also define two functions low$(i, d) =$ low(r_i) and up$(i, d) =$ up(r_i). The length of a list, d, is given by $|d|$. The join and meet operations ($d \sqcup_d d' = \langle r_1 \sqcup_r r'_1, \ldots, r_n \sqcup_r r'_n \rangle$ and $d \sqcap_d d' = \langle r_1 \sqcap_r r'_1, \ldots, r_n \sqcap_r r'_n \rangle$) and the partial ordering, $d \sqsubseteq_d d' \iff \forall i \in \mathbb{Z}^+, i \leq |d|,\ r_i \sqsubseteq_r r'_i P(\sqsubseteq_d)$, are defined for dimension lists of the same size, where $d = \langle r_1, r_2, \ldots r_n \rangle$ and $d' = \langle r'_1, r'_2, \ldots r'_n \rangle$.*

$$
dim(i,n,d) = \begin{cases} \bot_r & \text{if } n = \bot_n \vee d = \bot_d \\ \langle low(i,d), up(i,d) \rangle & \text{if } i \leq |d| \\ \langle 0, \omega \rangle & \text{if } |d| < i \leq up(n) \\ \langle 1, 1 \rangle & \text{otherwise} \end{cases} \tag{1}
$$

The function in (1), given $i \in \mathbb{Z}^+$, $n \in \mathbf{N}$ and $d \in \mathbf{D}$, gives the range representing the ith dimension. If the range is in the dimension list, this is returned; else if the dimension number is less than the number of dimensions, the dimension size is unknown so $\top_r$ is returned; otherwise the dimension is beyond the number of dimensions and its size is 1. Not all values of n and d give meaningful values for (1) and we will describe the constraints in Sect. 4.4.

4.3 Definedness

We also need to consider that a variable may not be defined. When a function is called, it does not need to be passed as many parameters as there are in the function signature. As a result some of the parameters may be undefined. Variables can also be undefined if they are only set on one branch of a conditional statement. In this case, it would be unknown whether the variable was defined. The *defined* flag is $\delta \in \mathbf{B}$. It is an error to use an undefined variable, but the built-in function `exist` takes a variable name and returns whether the variable exists.

4.4 Putting it all together

The full type of an array could be described by $\mathbf{T} \times \mathbf{N} \times \mathbf{D} \times \mathbf{B}$, but components of the type are not entirely independent. It is possible to produce many $n \in \mathbf{N}$, $d \in \mathbf{D}$ which give the same values of $dim(i,n,d)$ for all $i \in \mathbb{Z}^+$, for instance:

$$
\begin{gathered}
n_1 = \langle 2,2 \rangle, d_1 = \big\langle \langle 1,1 \rangle, \langle 1,1 \rangle \big\rangle, \quad n_2 = \langle 3,3 \rangle, d_2 = \big\langle \langle 1,1 \rangle, \langle 1,1 \rangle, \langle 1,1 \rangle \big\rangle \\
n_3 = \langle 2,3 \rangle, d_3 = \big\langle \langle 5,10 \rangle, \langle 2,2 \rangle \big\rangle, n_4 = \langle 2,3 \rangle, d_4 = \big\langle \langle 5,10 \rangle, \langle 2,2 \rangle, \langle 0,\omega \rangle \big\rangle \\
n_5 = \langle 2,3 \rangle, d_5 = \big\langle \langle 5,10 \rangle, \langle 2,2 \rangle, \langle 0,\omega \rangle, \langle 1,1 \rangle, \langle 1,1 \rangle \big\rangle
\end{gathered}
$$

In the above examples, $\langle n_1, d_1 \rangle$ and $\langle n_2, d_2 \rangle$ represent the same shape. However MATLAB would return 2 as the value of `ndims` and so n_2 is wrong. The same values of *dim* would also be given for $\langle n_3, d_3 \rangle$, $\langle n_4, d_4 \rangle$ and $\langle n_4, d_4 \rangle$ so clearly redundant information is present in d_4 and d_5. Finally if $n = \langle 2,2 \rangle$ and $d = \big\langle \langle 1,1 \rangle, \langle 1,1 \rangle, \langle 2,2 \rangle \big\rangle$, then clearly n and d contradict, as there are 3 dimensions. In order to compare types, we need a concrete description of each type with no ambiguity. In addition if a variable is undefined, it is meaningless for it to have shape or type. If the definedness is unknown, it can still have shape and type as might be the case when a variable is defined in only one branch of a conditional statement. The constraints on the type are given below:

1. The shape, $s \in \mathbf{S}$ ($\mathbf{S} \subset \mathbf{N} \times \mathbf{D}$), must be in the canonical form described in [5].

2. If an array, with full type $\langle t, s, \delta \rangle$, is undefined, e.g. $\delta = \mathit{false}$, then the values of t, n and d can only be $\perp_t$, $\perp_n$ and $\perp_d$ respectively.

We have thus defined a canonical form for our type system (along with the abstract interpretation concretisation functions), which allows us to compare two shapes and compute their least upper bound. Full details are given in [5]. All of this is then used by our data flow analysis and our partial evaluator proper, as described later in the paper.

5 Design of a Partial Evaluator

We wish to perform optimisations based on the characteristics of arrays such as shape and type. For example, while the exact value of a matrix may be unknown, the dimensions and whether it is real or complex could well be known. In this case built-in functions which try to determine these properties can be replaced by the actual values, which might lead to speed-ups due to loop unrolling and the removal of conditionals.

5.1 Overview

Below we give an overview of the stages of our partial evaluator:

1. Parse main source file to be partially evaluated (Sect. 5.2).
2. Insert the static values (given at the command line) by creating assignments within the parse tree of the main source file (Sect. 5.3).
3. Convert placeholders in the parse tree of the first function from the main source file.
   (a) Whenever a function is encountered, load and parse the function, while adding it to the list of functions called by the current function.
   (b) Repeat step 3 for all the functions in the list created in step 3a.
4. Partially evaluate the parse tree of the main function obtaining a new parse tree (Sect. 5.4).
5. Post-process the new tree, removing dead code.
6. Write out the final tree as MATLAB source code.

Each of these stages will be described in more detail in the following sections.

Our MATLAB partial evaluator (MPE) was written for GNU/Linux systems in C++, but only uses the MATLAB runtime libraries aside from the core libraries and so should be easy to port to other platforms on which MATLAB is supported. It is invoked from the command line and in its basic mode of operation takes one file and partially evaluates the functions found within it. It does not currently partially evaluate the whole system and currently does no polyvariant specialisation. Its effectiveness is therefore currently limited to functions that do not call other functions with dynamic parameters. Function calls with fully static parameters can be evaluated and so the results can be embedded directly into the partially evaluated function. Whether this is always desirable will be discussed later.

5.2 Lexical Analysis and Parsing

MATLAB was designed more to allow mathematicians to read it than for simple parsing. This leads to ambiguous constructs that are fairly simple for a human to understand as they can more easily make contextual judgements, but a lot harder for a lexical analyser. Problems occur because in matrices, spaces can be column delimiters or white space. Outside matrices new lines are treated as an end of command indicator, but inside they are treated as row separators. Fortunately `flex` can be made stateful thus avoiding the need for a hand-written lexical analyser.

Due to difficulties in disambiguating variables and function calls, identifiers that could represent either are stored initially as placeholders. In the following stage, these will be replaced with either variables or function calls.

5.3 Converting Placeholders

This stage is necessary because the distinction between variables and functions is not immediately determinable unlike in C (assuming macros are not used). Variables do not have to be declared but are created as required by assignments. Variables can also shadow the names of both built-in and ordinary functions. This means that an identifier could be used to indicate a function call at one point in a function and then later be used to access a variable if there is assignment to the variable in between. This also needs to be done by all MATLAB compilers and is discussed in [18] and [2].

The end result of this stage is one parse tree where all placeholders have been replaced with either variable, subscript or function call identifiers, for each function in all m-files in the system.

5.4 Partial Evaluation

In this pass, we try to evaluate as much of the function as possible. This stage takes a list of MATLAB statements and a table mapping variables to values and returns a new list of statements. This can be applied recursively to lists of commands within control flow statements like `for` loops.

For every expression, we have a structure which stores what information we have for it. In some cases the value will be known, in which case full substitutions can be performed. In the case where the actual value of an expression is not known, we store all relevant information that can be inferred about it. This corresponds to the full type described in Sect. 4.4.

There are two types of partial evaluation that are carried out by our tool. In one, a function is evaluated as much as is possible in conjunction with a list of variables and information about their contents, in the end producing a final table containing the values of variables at the end of a function. The other mode of operation also produces a new parse tree for the residual code as it goes along. Evaluation without producing code is required for dealing with function calls as well as for iterating over loops, which will be described later in Sect. 5.4.

For each kind of statement discussed in Sect. 3, we will now give a description of how they are partially evaluated.

Expressions. Expressions are stored in a binary tree structure. Our implementation performs a depth first traversal of the tree evaluating wherever possible.

Most binary operations in MATLAB are called *element-wise* binary operations. These either operate on two arrays with equal dimensions, one non-scalar and a scalar or two scalars. They always result in an array of the same shape as the non-scalar operand or a scalar in the case of two scalars. These include addition, subtraction, array multiplication (as opposed to matrix multiplication), left and right array division, array power, relational operations and logical operations. The operations that have non-array alternatives (multiplication, division and power) are prefixed with a full stop to differentiate them (e.g. `.*`, `./`, `.\` and `.^`). While the values of these of operands will frequently not be known it is often possible to make inferences about the dimensions of the operands and thus infer the dimensions of the result of an operation. We use the following scheme for $c = a \oplus b$, (a, b and c have shapes $s_a, s_b, s_c \in \mathbf{S}$. $s_s \in \mathbf{S}$ is the shape of a scalar and $\oplus$ is an element-wise binary operator):

- If a is a scalar ($s_a = s_s$), c will have the same shape as b ($s_c = s_b$). If b is a scalar ($s_b = s_s$), c will have the same shape as a ($s_c = s_a$). If both are scalars ($s_a = s_b$), c will also be a scalar ($s_c = s_s$).
- If there is complete information about the shape of a and b and they have identical shapes ($s_a = s_b$), c will also have the same shape ($s_c = s_a = s_b$).
- If incomplete information is known about the shapes of a and b, but they are definitely not scalars ($s_a \sqcap s_s = \bot_s \wedge s_b \sqcap s_s = \bot_s$) and their shapes do not conflict ($s_a \sqcap s_b \neq \bot_s$), then the shape of c will be the meet of the two shapes ($s_c = s_a \sqcap s_b$). E.g. a has 5 rows and is added to b which has 3 columns; the result will therefore have 5 rows and 3 columns.
- If a can be a scalar but b cannot ($s_a \sqcap s_s = s_s \wedge s_b \sqcap s_s = \bot_s$), use the shape of b ($s_c = s_b$), or if b can be a scalar but a cannot ($s_a \sqcap s_s = s_s \wedge s_b \sqcap s_s = \bot_s$), use the shape of a ($s_c = s_a$). E.g. a has 2 rows and 2 columns is added to b which has 1 column but an unknown number of rows. The only way the operation can be valid is if b has 1 row and is thus a scalar, in which case the result would have 2 rows and 2 columns.
- If both a and b can be scalars ($s_a \sqcap s_s = s_s \wedge s_b \sqcap s_s = s_s$), use the join of the two shapes ($s_c = s_a \sqcup s_b$). E.g. If a has between 1 and 3 rows, b has between 0 and 2 rows and both have one have column, then the result will have between 0 and 3 rows as either a or b could be a scalar.
- If both are definitely not scalars and there are conflicting dimensions ($s_a \sqcap s_s = \bot_s \wedge s_b \sqcap s_s = \bot_s \wedge s_a \sqcap s_b = \bot_s$), give an error.

The scheme above assumes that user code has no errors as it could hide errors by inferring valid shapes for operands when complete information is not available. This is discussed further in Sect. 7. Currently information is never passed backwards meaning that information, about the operands themselves, is never updated, but this change is planned for the future.

There are several non-array binary operations, matrix multiplication, division and power. These operations only work on two dimensional matrices and each has different dimension requirements.

– Matrix multiply : `a * b`
  The number of columns in a must match the number of rows in b. The result of the operation will have the same number of columns as b and the same number of rows as a. The exception here is that if either a or b are scalars then the result will have the same dimensions as the other.
– Left matrix division : `a \ b`
  a must have the same number of rows as b, in which case the result will have as many rows as a has columns and as many columns as b. The exception is if a is scalar, in which case it is equivalent to array division of b by a.
– Right matrix division : `a / b`
  a must have the same number of columns as b, in which case the result will have as many columns as b has rows and as many rows as a. The exception is if a is scalar, in which case it is equivalent to array division of a by b.
– Matrix power : `a ^ b`
  Either a or b must be a scalar and the other must be a square matrix. The result will have the same dimensions as the matrix.

There are two types of function calls that we have to deal with in expressions: built-in functions and m-files.

Built-in functions that have static parameters can usually be executed directly via the MATLAB runtime libraries. There are however some built-in functions that cannot be executed directly as they require context. Examples include `exist`, which can be used to determine the existence of variables as well as files and functions, I/O functions, graphing functions and timing functions.

Some built-in functions like `exist` can be evaluated but indirectly by examining our symbol table. It makes little sense to evaluate timing functions while partially evaluating as the intention would usually be to time the final program. I/O functions are discussed in Sect. 7.

Built-in functions that cannot be evaluated directly are handled by the partial evaluator internally. For each built-in function we extract as much information as possible about the return values based on the input passed to it. In the case of functions like `size` and `ndims`, we can fully evaluate them if we have sufficient shape information. If insufficient information is available, an entry, describing what we can determine of the shape and type of the returned value, is returned.

Functions stored in m-files are evaluated using the partial evaluation process without code generation. Later we plan to implement polyvariant specialisation where new specialised functions are created when some of the parameters are fixed. Currently functions are evaluated as completely as possible and if the final result is static then the function call can be removed and a constant substituted for it. If this is not possible, then any shape or type information garnered by evaluating the function is retained.

Assignments. Simple assignments discard the old value of the target variable and are relatively easy to handle. If the new value is known, it is stored in the symbol table, otherwise we store inferred information such as type, shape and rank. With assignments to subscripts, the target variable is changed but not replaced. If the indices are outside the bounds of the matrix, then the matrix is resized to allow the assignment.

Functions with multiple outputs are more complicated. If the function can be fully evaluated, it is removed and replaced with several assignments. E.g.

```
[a,b] = size(c);
```

If the shape of `c` is known then two assignments to `a` and `b` are substituted, but if only one dimension is known, we would have to use a function call. E.g.

```
a = size(c,1);
b = 1;
```

This is more expensive than the original as there are two assignments and a function call, but post-processing could eliminate the assignment to `b`. It is easy to transform the `size` function into multiple assignments, but most function calls are not so simple and so we do not perform this kind of transformation.

`for` loops. There are two cases to consider with `for` loops: the loop bounds are either static or dynamic. In the static case, the number of iterations is immediately determinable without data flow analysis as we do not consider `break` or `return` statements. Unrolling is achieved by partially evaluating the body of the loop for each iteration, setting the value of the loop variable as appropriate. Indexed assignments, to the loop variable in the loop body, require an explicit assignment to be written out. At the end of the loop, an assignment may be required to ensure the final value of the loop variable is available to the rest of the program. If the loop bounds indicate that the loop body is never executed, the entire loop is deleted.

If the loop bounds are dynamic, the loop is retained in the residual program. To find the least upper bound of the loop state, we iterate over the loop body evaluating its statements, comparing the state of the symbol table after each iteration. Note the loop variable is reset to dynamic before each iteration, although its shape and type is inferred if possible. A separate final state table is maintained with which the results are merged each time. If, as a result of a merge, the final state table is left unchanged, iteration ceases. This is almost certainly not an optimal approach and could, in the case of nested loops, lead to excessive computation but this will be refined later. This approach detects variables left unchanged after every iteration and preserves their values.

If the loop might not execute even once (due to its bounds), the final state table of the loop is merged with the symbol table that would be obtained if the loop did not execute, possibly leading to information being lost; therefore it is important to examine the loop bounds first in case this merge can be skipped.

`while` **loops.** No attempt is made to unroll these loops apart from removing `while` loops that are never executed. A fix-point iteration is performed for shape and type analysis as described above for `for` loops.

`if` **statements.** If the condition expression is static, the conditional statement is removed and replaced with an appropriate set of commands depending on whether or not the condition expression evaluated to zero. Otherwise, both sets of commands are partially evaluated with the same initial conditions and the resulting symbol tables merged, meaning that if both branches set a variable to the same value, it will have that value after the loop. If the value is different, but the type or size of the matrix are the same, then this information is retained instead as the least upper bound for the shape and type is calculated.

`switch` **statements.** As with `if` statements, if the condition is static the entire statement is removed and replaced with the appropriate set of commands. Otherwise cases that cannot match are removed, thus reducing the number of comparisons required. If the exact control flow cannot be determined, each branch of the `switch` statement needs to be partially evaluated in parallel and the results then merged, skipping branches that can never be reached.

Annotations. In addition to standard MATLAB language constructs, our tool recognises annotations which guide the partial evaluation. These always begin with `%#` and are ignored by MATLAB as comments. There are two types of annotations: variable annotations and function annotations. Variable annotations specify the type, shape and definedness of a variable.

```
%# x size [1 1]
%# x complex
%# y undefined
```

Function annotations describe how the function has been called. They specify the values returned by built-in functions like `nargin` and `nargout`, which return the number of parameters and the number of return values respectively.

```
%# nargin 2
%# nargout 1
```

5.5 Post-processing

In this phase dead code is removed. A statement is considered dead if it has no side-effects and does not affect the final result. We will use a simple approach, which is to work backwards through a function marking variables which are used so that the last assignment before it can be marked as live. As noted by Knoop [9], removing dead assignments or expressions can change the program semantics as the dead code could generate a run-time error. Multiplying two matrices with incompatible dimensions or raising a matrix to the power of another matrix is illegal and will halt execution. Problems like this are discussed further in Sect. 7.

Expressions. If an expression is on a line on its own and it is terminated with a semi-colon then it has no side-effects and can be safely removed. If there is no semi-colon, then the result of evaluating the expression will be printed to the screen. Expressions, containing function calls that have side effects, cannot be removed.

Assignments. If the assignment is to a variable whose value may be required for a later statement, it is considered live. If nothing depends on the variable being assigned then it is dead and can be removed in the same way as expressions above depending on the presence of semi-colons.

If the assignment is to multiple variables, but only some of those variables are live then ideally we would transform it into an assignment which only assigned to the relevant variables, but as with the partial evaluation of assignments, the transformation is tricky and so we do not attempt it. As with stand-alone expressions, we must be wary of removing assignments involving expressions containing function calls with side effects.

Loops. Our simple approach of moving backwards through the code will not work for loops. E.g.

```
function y = f(x)
a = 1;
y = 1;
for n = 1:x
  y = n * a;
  a = a + n;
end
```

Moving backwards with `y` as the only live variable would result in the second assignment to `a` being removed, even though the previous line is dependent on it. To avoid this, the loop dependants must first be found and treated as live variables before working backwards. In this case `n` and `a` would be marked live and the assignment to `a` would be left intact. Since removing a statement can change the loop dependants, the two steps need to be iterated until no changes are made.

In the previous example, `x` could be less than 1, meaning that the loop body would never be executed. If the loop body is never executed then the value of `y` would remain 1, but moving backwards naively assuming the body of the loop is executed would lead to the removal of the initial assignment leaving `y` undefined. Unless we can guarantee that $\texttt{x} \geq 1$, we must assume that the loop might not be executed, in which case the live variable information from the loop must be merged with the live variable information from after the loop to ensure that correct code is generated.

This is not a perfect algorithm, but it is quite simple to implement. For instance an assignment which only creates new values for dead variables will not be removed. Muchnick [14] describes an algorithm that deals with even these assignments using UD and DU chains. Knoop [9] describes partial dead code elimination which allows assignments to be moved to the code blocks in which the assigned variables are live, speeding up the execution for alternate code blocks.

Conditional Statements. A simple conservative approach to dealing with conditional statements is to post-process all code blocks in parallel and then merge the lists of live variables. Later code motion may be used to perform partial dead code elimination [9].

Annotations. These are just removed from the final output.

6 Results

In this chapter we will evaluate the effectiveness of our partial evaluator on several source programs. The code for these tests is a mixture of code developed inside the Computational Engineering and Design research group at the University of Southampton, code come from partners from other universities and code found in code repositories on the internet. All of the timings were taken using the MATLAB 6.1 interpreter.

Experiment 1. The first code tested was a function for the generation of Chebyshev polynomials, which, like power series, are used to approximate functions by summing terms. As with power series, using more terms leads to better approximations. This function has two parameters, a m-by-n matrix, c of coefficients for calculating m functions with n terms and a vector, x, as input to the functions. Table 1 shows the relative timings for the chebyshev function (iterated 5000 times to get measurable results). Timings are shown where the function has been partially evaluated where just n is fixed, c is fixed and lastly where c is fixed along with the size of x. The timings are further subdivided according to whether post-processing was used. The results show a steady increase in performance as more information is fixed, with the final function running in half the time of the original. Partial evaluation has been previously been successfully applied to Chebyshev approximation previously [6].

Experiment 2. The second test function, when given a set of points from a function, computes the Lagrange interpolating polynomial that passes through them and returns a set of points on the curve. The MATLAB code is comprised of two nested loops both dependent on the number of points to interpolate followed by a third loop also dependent on the number of points. We specialised this function by first fixing the number of points, n, (and thus the number of x and y coordinates) resulting in all of the loops being unrolled and then further specialised it by fixing the x coordinates. Again, timings were taken with and without post-processing. Table 2 shows the improvements we achieved including at least 50% speed increases when the x coordinates are completely fixed.

	Original	n fixed		c fixed		c and size of x fixed	
`size(c,2)`		No p.p.	With p.p.	No p.p.	With p.p.	No p.p.	With p.p.
2	1.00	0.90	0.89	0.81	0.77	0.60	0.48
4	1.00	0.94	0.92	0.84	0.81	0.59	0.50
6	1.00	0.94	0.93	0.85	0.82	0.58	0.51
8	1.00	0.94	0.94	0.85	0.83	0.57	0.52
10	1.00	0.95	0.95	0.86	0.84	0.57	0.52

Table 1. Relative timings for the Chebyshev functions with $m = 3$ and $p = 3$, relative to original function (p.p. is post-processing).

		n fixed		x-coordinates fixed	
n	Original	No postproc	Postproc	No postproc	Postproc
2	1.00	0.73	0.67	0.67	0.58
4	1.00	0.82	0.79	0.68	0.64
6	1.00	0.85	0.83	0.67	0.65
8	1.00	0.86	0.84	0.68	0.66
10	1.00	0.87	0.86	0.68	0.66

Table 2. Relative timings for the Lagrange functions with values of n

Experiment 3. The next example function solves the Gaussian Hypergeometric differential equation, $x(1-x)\frac{\mathrm{d}^2y}{\mathrm{d}x^2} + c - (a+b+1)x\frac{\mathrm{d}y}{\mathrm{d}x} - aby = 0$, using a series expansion. The main work is done by a single `for` loop which calculates the series terms. To get more accurate results, higher order series terms are required and thus more iterations. The number of terms is the parameter that we have chosen to specialise. As can be seen from Table 3, partial evaluation with post-processing is very effective at speeding up the function, showing a 62% performance increase over the original function.

n	Original	Partially Evaluated	Post-processed
4	1.17	0.85 (0.73)	0.72 (0.62)
6	1.47	1.07 (0.73)	0.94 (0.64)
8	1.77	1.30 (0.73)	1.17 (0.66)
10	2.08	1.54 (0.74)	1.40 (0.67)

Table 3. Timings in seconds for the Gaussian Hypergeometric differential equation solver (iterated 8000 times). (Relative times are given in brackets).

Experiment 4. Our final set of results in Table 4 are for a computational fluid dynamic solver in our research group. The main function is very general allowing users to choose what parameters they want to optimise. After some initial set

Original	After `mpe`	After `mpe` with postproc
67.91	54.64	52.77
67.88	54.54	52.71
67.97	54.55	52.74
67.92	54.58 (0.80)	52.74 (0.78)

Table 4. Timings for the computational fluid dynamic solver in seconds. (Relative times to the original are shown in brackets).

up code which can be fully evaluated when specialised, it consists of a main loop that cannot be unrolled and an inner loop which can be, leading to the removal of many conditional statements and allowing the full evaluation of several function calls. This function would benefit from many traditional optimisation techniques but since MATLAB does not perform these and our partial evaluator does, we can see some improvements.

The code we have examined in this section was chosen because it was in the subset of MATLAB that we can handle and because it had control flow structures that would benefit from unrolling. We have not examined highly vectorised codes with few control flow structures as we conjecture that partial evaluation can do little without making code non-vectorised, which would hurt performance. The code we have examined has shown promising results. Inferring shape characteristics has allowed loop unrolling in many cases, but the large performance increases in experiment 1 and experiment 2 came from knowing the actual values stored in these matrices.

7 Future Work and Conclusion

The results show that partial evaluation can give large performance increases for relatively simple functions like those in experiments 1 to 3, but also for larger programs as in experiment 4. The results we have seen are encouraging and show that partial evaluation can achieve performance increases for MATLAB code. Unfortunately the number of functions that we could assess was limited by the subset of MATLAB that we currently handle. The omission of support for functions taking variable numbers of parameters, cell arrays and control flow change keywords like `return`, `break` and `continue` means that many functions, that could have benefited from partial evaluation, had to be discarded. Below are other areas where we plan to expand the capabilities of our tool.

While Loops. Currently while loops are not unrolled at all except for in the trivial case where the loop condition can never be met and so it can be deleted. Data flow analysis will be required to determine whether the loop condition will be static throughout the loop.

Assignment Amalgamation. As a result of unrolling the partial evaluator generates large series of assignments, either overwriting the same variable again and again while using its value as an operand or by writing to different subscripts of a variable. Many of these cases can be combined into single assignments in the preprocessing stage giving performance increases. Compilers for many languages optimise these away and so partial evaluators for these languages do not need to perform this optimisation themselves. Unforunately MATLAB does not do this (even when using the compiler).

Asserting Assumptions. When confronted with operations on operands for which only incomplete shape information is known, our implementation assumes that the program is bug-free and still attempts to infer shape and type information. Function calls using only this information could be fully evaluated and the original operation could then be removed, changing the semantics of the program in the case of programmer error. We plan to offer options to automaticall insert into the residual code assertions to check for these cases.

Widening. Currently MPE can loop infinitely on input containing loops which steadily increase shape values. Checks needs to be carried out to find these cases and widen the shape value to prevent further iteration.

I/O operations. As mentioned in Sect. 5.4, we currently do not support loading in data with I/O commands like `fopen`, `fscanf` and `fclose`. To be effective with mathematical codes, we must handle data sets that are loaded from files.

Polyvariant specialisation. This is very important for getting speed increases from large programs rather than the fairly simple functions examined in this work. For instance, most MATLAB functions check to see that they have been passed enough parameters and that their dimensions are valid. A significant advantage of polyvariant specialisation should be the prevalidation of parameters thus simplifying many functions.

We have presented the first partial evaluation system for MATLAB. We have shown how to deal with MATLAB data structures, notably how to store shape information about partially specified matrices. This is of utmost importance for scientific MATLAB code, as knowing the shape of a matrix often enables one to perform loop unrolling (and it is less common to statically know the full values of an entire matrix). We have presented our implementation and have shown on several non-trivial, practical examples that our system has achieved a significant speed increase.

References

1. G. Almási. *MaJIC: A MATLAB Just-In-Time Compiler.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.
2. G. Almási and D. Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 294–303. ACM Press, 2002.
3. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.
4. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, 1992.
5. D. Elphick. Implementation of a MATLAB partial evaluator. Technical Report DSSE-TR-2003-4, University of Southampton, 2003.

6. R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.
7. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
8. P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. In G. Snelting and U. Meyer, editors, *Semantikgestützte Analyse, Entwicklung und Generierung von Programmen. GI Workshop*, pages 45–54, Schloss Rauischholzhausen, Germany, 1994. Justus-Liebig-Universität Giessen.
9. J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 147–158. ACM Press, 1994.
10. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
11. MAGICA website. `http://www.ece.northwestern.edu/cpdc/pjoisha/MAGICA/`.
12. V. Menon and A. E. Trefethen. MultiMATLAB: Integrating Matlab with high performance parallel computing. In *Supercomputing '97 ACM SIGARCH and IEEE Computer Society*, pages 1–18, 1997.
13. T. Æ. Mogensen and P. Sestoft. Partial evaluation. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.
14. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.
15. S. Pawletta, T. Pawletta, W. Drewelow, P. Duenow, and M. Suesse. A MATLAB toolbox for distributed and parallel processing. In Moler C. and S. Little, editors, Proc. of the Matlab Conference 95, Cambridge, MA. MathWorks Inc., October 1995.
16. M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary results from a parallel MATLAB compiler. In *International Parallel Processing Symposium*, pages 81–87. IEEE CS Press, 1998.
17. L. D. Rose. Compiler techniques for MATLAB programs. Technical Report UIUCDCS-R-96-1956, University of Illinois at Urbana-Champaign, 1996.
18. L. D. Rose, K. Gallivan, E. Gallopoulos, B. A. Marsolf, and D. A. Padua. FALCON: A MATLAB interactive restructuring compiler. In *Languages and Compilers for Parallel Computing*, pages 269–288. Springer, 1995.
19. U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *European Conference on Object-oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390, 1999.
20. The MathWorks, Inc. - About Us.
`http://www.mathworks.com/company/aboutus.shtml`.
21. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, Jan. 2001.