MCVM: AN OPTIMIZING VIRTUAL MACHINE FOR THE MATLAB PROGRAMMING LANGUAGE

by

Maxime Chevalier-Boisvert

School of Computer Science
McGill University, Montréal

August 2009

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

Abstract

In recent years, there has been an increase in the popularity of dynamic languages such as Python, Ruby, PHP, JavaScript and MATLAB. Programmers appreciate the productivity gains and ease of use associated with such languages. However, most of them still run in virtual machines which provide no Just-In-Time (JIT) compilation support, and thus perform relatively poorly when compared to their statically compiled counterparts. While the reference MATLAB implementation does include a built-in compiler, this implementation is not open sourced and little is known abouts its internal workings. The McVM project has focused on the design and implementation of an optimizing virtual machine for a subset of the MATLAB programming language.

Virtual machines and JIT compilers can benefit from advantages that static compilers do not have. It is possible for virtual machines to make use of more dynamic information than static compilers have access to, and thus, to implement optimization strategies that are more adapted to dynamic languages. Through the McVM project, some possible avenues to significantly improve the performance of dynamic languages have been explored. Namely, a just-in-time type-based program specialization scheme has been implemented in order to take advantage of dynamically available type information.

One of the main contributions of this project is to provide an alternative implementation of the MATLAB programming language. There is already an open source MATLAB interpreter (GNU Octave), but our implementation also includes an optimizing JIT compiler and will be open sourced under the BSD license. McVM aims to become a viable implementation for end-users, but could also see use in the compiler research community as a testbed for dynamic language optimizations. In addition to the contribution of the McVM

framework itself, we also contribute the design and implementation of a novel just-in-time type-based program specialization system aimed at dynamic languages.

The novel specialization system implemented in McVM shows much promise in terms of potential speed improvements, yielding performance gains up to 3 orders of magnitude faster than competing implementations such as GNU Octave. It is also easily adaptable to other dynamic programming languages such as Python, Ruby and JavaScript. The investigation of performance issues we make in this thesis also suggests future research directions for the design of dynamic language compilers of the future.

Résumé

Ces dernières années, il y a eu une augmentation de la popularité des langages dynamiques tels que Python, Ruby, PHP, JavaScript et MATLAB. Les programmeurs apprécient les gains de productivité et la facilité d'utilisation associée à ces langues. Cependant, la plupart de ces langages s'exécutent encore dans des machines virtuelles qui ne fournissent aucun support pour la compilation à la volée, et ont donc une performance inférieure si on les compare à leurs homologues compilés statiquement. Bien que l'implémentation de référence de MATLAB comprenne un compilateur intégré, cette application n'est pas open source et son fonctionnement interne demeure un secret industriel. Le projet McVM a mis l'accent sur la conception et l'implémentation d'une machine virtuelle optimisée pour un sous-ensemble du langage de programmation MATLAB.

Les machines virtuelles et les compilateurs à la volée peuvent bénéficier d'avantages que les compilateurs statiques n'ont pas. Il est possible pour les machines virtuelles de faire usage d'informations dynamique à laquelle les compilateurs statiques n'ont pas accès, et donc, de mettre en oeuvre des stratégies d'optimisation qui sont plus adaptées aux langages dynamiques. À travers le projet McVM, plusieurs avenues possibles pour améliorer considérablement la performance des langages dynamiques ont été explorées. Entre autre, un système de spécialisation de programmes à la volée permettant de profiter d'informations sur les types disponible dynamiquement a été implémenté.

L'une des principales contributions de ce projet est de fournir une implémentation alternative du langage de programmation MATLAB. Il existe déjà un interpréteur MATLAB open source (GNU Octave), mais notre application comprend également un compilateur à la volée optimisé et sera distribuée sous la licence open source BSD. McVM vise à devenir

une implémentation viable pour les utilisateurs finaux, mais pourrait aussi être utilisée dans le milieu de la recherche sur les compilateurs comme outil d'expérimentation. En plus de la contribution du logiciel intégré McVM lui-même, nous avons également contribué à la conception et la réalisation d'un système de spécialisation de programme à la volée visant à l'optimisation des langages dynamiques.

Le système de spécialisation mis en oeuvre dans McVM se montre très prometteur en termes de potentiel d'améliorations de la vitesse d'exécution, permettant des gains de performance allant jusqu'à trois ordres de grandeur comparés aux implémentations concurrentes telles que GNU Octave. Il est également facilement adaptable à d'autres langages de programmation dynamique tels que Python, Ruby et JavaScript. L'examen des problèmes de performance que nous faisons dans cette thèse suggère aussi des pistes de recherche pour la conception des compilateurs de langages de programmation dynamiques de l'avenir.

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT).

We acknowledge contributions by Ph.D. student Nurudeen Lameed, implementor of the bounds check elimination analysis and the front-end interface system used in the McVM virtual machine

We acknowledge the support and guidance given by our thesis supervisors: professor Laurie Hendren and professor Clark Verbrugge.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

MATLAB is a well-known, widely-adopted and easy-to-use programming language aimed at the scientific and engineering communities. Unfortunately, the Mathworks reference implementation of MATLAB is neither free nor open sourced, and little about its internal workings is published. There is already an open source MATLAB interpreter (GNU Octave) which offers a fairly complete implementation of the MATLAB language, but this implementation performs poorly, making it unsuitable for many computationally intensive applications.

Dynamic languages such as Python, Ruby, PHP, JavaScript and MATLAB are gaining popularity at an impressive rate[1]. However, most of them perform poorly when compared to their statically compiled counterparts (e.g.: C, C++, Java, etc.). This is largely because most of these languages are purely interpreted. Dynamic languages are built with programmer convenience in mind, but because of their highly dynamic nature, it is difficult to predict their behavior ahead of time.

The McVM virtual machine is a component of a larger effort known as the McLab project[2], which was initiated by Professor Laurie Hendren of McGill university. The overall goal

[1] TIOBE Programming Community Index:
http://www.tiobe.com/index.php/content/paperinfo/tpci/

[2] The McLab Homepage:
http://www.sable.mcgill.ca/mclab

of the project is to find ways to improve the performance, usefulness and accessibility of current scientific programming languages. Several graduate students, members of the Sable Research Group (McGill's compiler research laboratory), participate in this project. The McLab team currently focuses its efforts on the MATLAB programming language.

McVM is McLab's virtual machine, which currently implements a significant subset of the MATLAB language. It is a testing ground for new compiler optimizations aimed at scientific and dynamic languages. It is also an opportunity to test new ideas or language features that could be integrated in scientific programming languages of the future. In the following sections we will explain what makes McVM an interesting and challenging research project, and why we believe it is an important contribution to the compiler research community.

Much of this thesis focuses not specifically on the performance issues faced by scientific programming languages, but rather on the specific problems related to optimizing dynamic languages such as MATLAB. We believe that virtual machines and JIT compilers can benefit from advantages that static compilers do not have. It is possible for virtual machines to make use of more dynamic information than static compilers have access to, and thus, to implement optimization strategies that are more adapted to dynamic languages.

Through the McVM project, we explore some possible avenues to significantly improve the performance of dynamic languages. We have designed and implemented an interpreter and Just-In-Time (JIT) compiler for a non-trivial subset of the MATLAB language. Our JIT compiler integrates analyses and optimizations mechanisms designed specifically to improve the performance of dynamic languages such as MATLAB.

1.1 Contributions

The McVM project makes the following contributions:

- Design and implementation of an extensible interpreter and virtual machine for a non-trivial subset of the MATLAB programming language.

- Design and implementation of an extensible JIT compiler for the McVM virtual machine. This JIT compiler yields performance numbers up to three orders of magnitude faster than GNU Octave and in some cases it is faster than the reference MATLAB implementation.
- A novel just-in-time type-based program specialization system aimed at dynamic languages. This system, which shows much promise in terms of potential speed improvements, is also easily adaptable to other dynamic programming languages such as Python, Ruby and JavaScript.
- A type inference analysis based on abstract interpretation, designed specifically for dynamic programming languages such as MATLAB.
- Additional type-based optimizations for our JIT compiler. These optimizations make use of information provided by our type inference analysis.
- A detailed analysis of the performance of our JIT compiler, the efficiency of our type inference analysis and gains associated with our JIT compiler optimizations.
- Contribution of the entire McVM source code under the BSD open source license.

We aim to make McVM a viable MATLAB implementation which should become increasingly usable by end-users for real-world scientific applications. However, another potential use for our implementation is as a research framework. This framework will make it possible for other researchers to easily try experimental optimization techniques and test novel language features. It will be possible for developers to add customized features to McVM that are specific to their area of study, something that is hardly possible with the reference MATLAB implementation.

There is already an open source MATLAB interpreter (GNU Octave), but our implementation also includes an optimizing JIT compiler, something that was not previously available. Furthermore, our implementation will be open sourced under the BSD license, which is more liberal than the GNU GPL license, and more likely to encourage reuse of our implementation by both academic and commercial entities.

Finally, based on our experience with McVM, we propose promising future directions for those researching and implementing JIT compilers and virtual machines for dynamic languages. We identify key factors crucial to the implementation of efficient virtual machines for dynamic languages and propose ways to improve upon the performance results we have obtained with the McVM virtual machine.

1.2 Thesis Outline

Our thesis is divided into 8 chapters (including this introduction chapter). Chapter 2 introduces background knowledge and related work helpful in understanding our research, including a brief description of the MATLAB language. Chapter 3 discusses which features of the MATLAB programming language are currently supported in McVM. Chapter 4 examines the McVM virtual machine architecture in detail, including our JIT compilation and just-in-time type-based specialization strategy.

Chapter 5 explains the type inference strategy required by our type-based specialization mechanism. Chapter 6 discusses the performance of our JIT compiler in comparison to MATLAB and GNU Octave, as well as the usefulness of our various optimization strategies and the effectiveness of our type inference system. Chapter 7 discusses issues associated with the MATLAB programming language design that make optimization difficult. Finally, chapter 8 presents our conclusions and outlines some possible future research avenues for optimizing scientific and dynamic languages beyond what we have achieved.

Chapter 2

Background and Related Work

In this chapter, we present background information helpful to the understanding of this thesis as well as research work related to our own. We begin with a brief overview of the MATLAB language. This includes supported features, its type system, the execution model used, programming code examples, typical features of MATLAB programs as well as a discussion of the similarities between MATLAB and FORTRAN.

This is followed by a discussion of virtual machines, dynamic languages and the associated optimization challenges. We then present related work in the areas of compilers for dynamic languages, program specialization, type inference and adaptive optimization. This research work is in direct relation to our own or explores similar areas of compiler research.

2.1 The MATLAB Language

The MATLAB language was originally invented in the late 1970s by Cleve Moler, then a professor of computer science at the University of New Mexico. He designed the language to give his students access to some of the power of FORTRAN, without having to learn the FORTRAN language itself[1]. Aimed towards students, MATLAB was to be easier to learn

[1]The Origins of MATLAB:
http://www.mathworks.com/company/newsletters/news_notes/clevescorner/dec04.html

than other languages commonly used at the time. Since then, MATLAB has gained wide acceptance in both academic, scientific and engineering circles.

The MATLAB language has evolved significantly throughout the years. The language is a procedural dynamic programming language geared towards scientific computation. It is dynamically-typed, weakly-typed, and incorporates many features found in other dynamic languages, such as the run-time creation of closures. It also integrates native support for n-dimensional matrix data types and provides a large library of common matrix operations (e.g.: addition, inversion, multiplication) and algorithms (e.g.: SVD, QR-Factorization).

2.1.1 Supported Features

The Mathworks MATLAB implementation is very feature rich. The short list below enumerates some of its most prevalent features [SD04]:

- Interactive mode with read-eval-print loop
- Code editor and debugging environment
- Effective documentation search system
- Built-in support for complex-numbers
- Uniform treatment of all basic types as matrices
- Range expressions and array slicing/reshaping
- Powerful built-in matrix operations
- Nested function definitions
- Creation of closures from nested functions
- Creation of closures from lambda expressions
- Function handles

- Object-oriented programming support
- Extensive library of numerical algorithms
- Graphical 2D and 3D plotting tools
- C and FORTRAN function wrapping
- Java code integration

The reference MATLAB implementation from Mathworks offers more than a simple MATLAB interpreter. Rather, it offers a fully featured development environment for MATLAB, complete with a code editor, a debugger, a search system to find documentation about library function, and a way to save your "workspace" so you can return to your work at a later time.

2.1.2 MATLAB's Type System

MATLAB is a dynamically-typed and weakly-typed language. It has many basic types, including several integer types of different precision, `single` and `double` real-valued types, implicit support for complex values, `logical` boolean-valued types, and `character` types [SD04]. All of these types are implicitly treated as if they were always matrices. To the programmer, scalar values appear as if they were matrices of size 1x1. Character types are MATLAB's primitive for creating strings, that is, in MATLAB, strings are matrices of characters.

MATLAB also sports a matrix type known as the cell array. This is conceptually a matrix of references to other MATLAB objects. Function calls have call-by-value semantics and symbols refer to objects directly rather than acting as references to them. MATLAB has no pointer type, making it impossible to create truly cyclic data structures without using MATLAB's object-oriented facilities. MATLAB allows the creation of function handles to existing functions as well as the creation of closures on the fly.

In MATLAB, there are no obvious type promotion rules as in C. Indeed, it seems that the language has largely grown organically, and thus, has its own complex typing rules. For

example, operations between unmatched types largely produce erroneous behavior (exceptions being thrown). Operations between integers are only allowed if the integers have the same type. However, MATLAB supports operations between `double` and some of its integer types (but not all), in which case the result is of the integer type. On the other hand, operations between `single` values and integer types are always erroneous.

There is also a basic `struct` aggregate type which allows the creation of objects with named fields. An interesting fact is that the field names are passed as strings when creating a struct object (these objects are created by calling a `struct` function rather than by instantiating them from a previously defined type), and thus can be dynamically generated. Later versions of MATLAB have introduced classes and more advanced object-oriented programming features.

2.1.3 MATLAB's Execution Model

In MATLAB, variables exist within a "workspace" object, which maps variable names to their values. There is a global workspace (the global scope) into which global variables are defined. When performing an assignment in interactive mode (e.g.: by writing `A = 1;` at the prompt), the user creates bindings in the global workspace. MATLAB distinguishes between two types of reusable units of code: functions and scripts. Functions can have multiple input and output parameters. These parameters are bound in the function's own workspace. When a symbol that is unbound in a function is evaluated, however, its value is evaluated in the global workspace. Scripts, on the other hand, take no parameters, and operate directly on the caller's workspace. The caller can be a function, or the global workspace if the script is called at the prompt in interactive mode.

Nested functions are allowed in MATLAB. These functions can only be called from their parent function, or call themselves. When they are called, they get their own workspace where their parameters are bound and which extends the parent function's current workspace. Hence, nested functions can have access (and modify) variables in the parent workspace. Closures can be made from nested functions using the closure operator (i.e.: using the `A = @function_name;` syntax). This will create a function handle to a closure in which

all unbound variables of the nested functions are assigned their current value in the parent workspace. This handle can then be passed to outside functions if desired.

As stated before in section 2.1.2, all function calls have call-by-value semantics, and assignments create copies rather than references. This is also true of assignments to cells of cell arrays or to fields of structs. Hence, MATLAB makes it impossible to have cyclic data structures without using classes and object handles. MATLAB allows functions to return multiple values at once. However, it also allows functions returning multiple values to return only some of these values if the caller does not assign all of the return values to variables.

MATLAB does not have a module system in the traditional sense. Rather, it uses a system based on storing units of code (scripts and functions) into M-files, and separating these into directories. MATLAB allows changing the currently accessible functions and scripts when changing the current directory, by issuing a `cd` command. This command takes a string representing the target directory path as input. Note that this string does not need to be a constant value, and could be read from the console, for example. When the `cd` command is issued, symbol lookups in all currently executing functions will be affected.

2.1.4 MATLAB Code Examples

```
function d = editdist(s1, s2)
%-----------------------------------------------------------------------
% This function M-file finds the edit distance between the
% source string s1 and the target string s2.
%
% Source:
%  MATLAB 5 user contributed M-Files at
%  http://www.mathworks.com/support/ftp/.
%  ("Anything Not Otherwise Categorized" category).
%
% Author:
%  Miguel A. Castro (talk2miguel@yahoo.com).
%-----------------------------------------------------------------------
```

```
14
15 DelCost = 1; % Cost of a deletion.
16 InsCost = 1; % Cost of an insertion.
17 ReplCost = 1; % Cost of a replacement.
18
19 n1 = size(s1, 2);
20 n2 = size(s2, 2);
21
22 % Set up and initialize the dynamic programming table.
23 D = zeros(n1+1, n2+1);
24
25 for i1 = 1:n1,
26   D(i1+1, 1) = D(i1, 1)+DelCost;
27 end;
28
29 for j1 = 1:n2,
30   D(1, j1+1) = D(1, j1)+InsCost;
31 end;
32
33 for i1 = 1:n1,
34   for j1 = 1:n2,
35     if s1(i1) == s2(j1)
36       Repl = 0;
37     else
38       Repl = ReplCost;
39     end;
40
41     D(i1+1, j1+1) = min([D(i1, j1)+Repl D(i1+1, j1)+ ...
42     DelCost D(i1, j1+1)+InsCost]);
43   end;
44 end;
45
46 d = D(n1+1, n2+1);
```

Listing 2.1 A MATLAB program example

The example shown in listing 2.1 shows a simple MATLAB function for computing the

edit distance between two strings. This function could be invoked by inputting the `d = editdist('cow', 'meow');` command at the prompt, or it could be called from another function. This example demonstrates array indexing and iterating over ranges of values. Note that MATLAB arrays have indices starting at 1 instead of 0. The `...` operator denotes continuation of a statement to the next line. Also note that the `n1 = size(s1, 2);` statement gets the size of the array `s1` along the second dimension, that is, the length of the string `s1`. The statement `D = zeros(n1+1, n2+1);` creates a 2D matrix initialized with zero values of size (n1+1)x(n2+1).

```
>> % Here a matrix is being assigned to variable "A"
>> A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]

A =

     1  2  3  4
     5  6  7  8
     9 10 11 12
    13 14 15 16

>> % A is accessed using scalar indexing, the last element of the first
   row is read
>> A(1,4)

ans =

     4

>> % An entire row or column of A can be read by specifying a range of
   indices with the colon operator
>> A(:,1)

ans =

     1
     5
     9
```

```
26     13
27
28 % One can also write to a sub-array (or slice) of A using ranges of
       indices
29 >> A(1, 2:3) = [7 7]
30
31 A =
32
33      1 7 7 4
34      5 6 7 8
35      9 10 11 12
36     13 14 15 16
```

Listing 2.2 Sub-arrays indexing (or slicing) in MATLAB

Listing 2.2 shows sample output of MATLAB being run in interactive mode, where commands can be typed at the prompt, and the resulting output is immediately displayed. In this example, we show that the matrix variable `A` can be indexed using scalar values, similar to the way 2D arrays are indexed in languages like Java (except that MATLAB matrices are row-major). However, MATLAB matrices can also be read or written to multiple elements at a time using ranges of indices specified with the colon operator. That is, one can create sub-matrices from existing matrices, or assign to sub-portions of an existing matrix. We refer to this capability as slicing.

2.1.5 Typical Program Features

MATLAB being aimed at scientific computing, the features of the typical MATLAB program are different from those of the typical C, C++ or Java program. In the case of C++ and Java, for example, it is assumed that these programs will contain many short methods that do very little work (e.g.: accessor methods), and that the call graph complexity may be very complex. C, C++ and Java programmers usually aim to divide work into the smallest units possible, so that code will be neatly organized, even if the program grows to very high levels of complexity.

This is usually not the case in MATLAB. Those who use the language seem usually focused on getting work done quickly and on making computations as efficient as possible. Typical MATLAB programs will often have very few functions and a fairly simple call graph. These functions can sometimes grow in length to levels that would be considered undesirable by C, C++ or Java programmers. Use of built-in operators and of matrix operations is strongly encouraged over the use of nested loops whenever possible, sometimes making the code appear rather cryptic.

2.1.6 Similarities to FORTRAN

As previously stated, MATLAB was inspired from FORTRAN, but intended to be easier to learn. As such, it shares several similarities with FORTRAN. Both are procedural and imperative programming languages. Both languages are aimed at scientific computing. Both share powerful matrix manipulation primitives. Both allow slicing of multidimensional arrays at run-time. MATLAB provides a rich array of built-in operators and functions to manipulate matrices, as well common matrix algorithms. Much of these built-in functions are actually interfaces to functions found in the BLAS and LAPACK libraries, typically used with FORTRAN programs. As a result, a technical similarity between FORTRAN and MATLAB is that they both store arrays and matrices in column-major order.

Both languages also share some common types, such as `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL` and `CHARACTER`. FORTRAN allows specifying multiple degrees of precision for integers, similarly to MATLAB, which has multiple integer types. FORTRAN also has two kinds of `REAL` variables, similar to MATLAB's `single` and `double`. One difference is that whereas FORTRAN distinguishes between scalar and array variables, MATLAB implicitly treats all scalar variables as matrices of size 1x1, which can be dynamically extended in size.

We could also say that there are similarities in programming styles used by MATLAB and FORTRAN programmers (possibly because the two crowds intersect to a large extent). It is very common for FORTRAN programs to be written in one or few files containing few procedures, with procedures being potentially quite long. As stated in section 2.1.5,

the same is often true of MATLAB programs. On the stylistic side, it could also be said that both FORTRAN and MATLAB programs use rather nondescriptive variable names, probably because much of the code implemented in MATLAB and FORTRAN is based on widely known algorithms which programmers assume require not much documentation.

In terms of important differences, FORTRAN uses static typing. It has variable declarations with explicit type declarations, whereas MATLAB is dynamically typed and has no explicit variable declarations. FORTRAN programs are also typically statically compiled, whereas MATLAB code typically runs in an interactive environment where users can enter commands into a console, and there is no explicit compilation process. FORTRAN also has a traditional module system, whereas MATLAB uses a system based on file names and directory trees to organize code. Whereas FORTRAN has pointers, MATLAB does not, but has function handles. Finally, FORTRAN programs are expected to make parallelism explicit, but MATLAB programs do not, and instead rely on optimized parallel libraries.

2.2 Virtual Machines and Dynamic Languages

Dynamic languages are high-level programming languages that execute, at run-time, behaviors that would otherwise be statically compiled in other languages. Common dynamic languages include Python, Perl, PHP, Ruby, Scheme, Smalltalk and of course, the MATLAB language. These languages are often garbage-collected, dynamically-typed and weakly-typed (variable types are determined at run-time). They also tend to have superior reflective capabilities compared to static languages.

Because of their dynamic components, such languages typically run in execution environments called Virtual Machines (VMs). A VM is essentially a program (written in any language) that executes programs written in the dynamic language [AAF+05]. This execution can be achieved through the use of an interpreter (as with Python), or by compiling programs at run-time using a Just-In-Time (JIT) compiler (as with Java). Virtual machines have the fundamental role of implementing the dynamic components of dynamic languages.

In addition to providing an execution environment for a dynamic language, virtual ma-

chines can also incorporate analyses and transformations to optimize programs at run-time. This can be advantageous, because more information about the programs to be optimized is available at run-time than would be available statically. A program can also be dynamically optimized in various stages, such that the VM will adjust its optimization strategy based on profiling data gathered while program is running.

Dynamic languages were once somewhat marginal, but they have recently started becoming more widely used. Programmers tend to like languages like Python, for example, because they claim that those languages put less constraints on the programmer. It is often said that dynamic languages achieve more "work" per line of code than static languages, and thus, allow programmers to be more productive. This makes some sense from a technical standpoint, because virtual machines perform much background work at run-time when executing dynamic languages, in a manner that is transparent to the programmer.

2.2.1 Optimization Challenges

Dynamic languages present some optimization challenges of their own. They are typically harder to optimize than statically compiled languages because they are, so to speak, more dynamic. That is, the semantics of dynamic languages make it less obvious what the exact behavior of the program will be at run-time. A static compiler has less information to work with when compiling a language that is dynamically typed, versus a statically typed language, for example.

When a language is dynamically typed, the type of a given variable can change at different program points, and in some cases, be impossible to determine statically, because it could depend on the program's input. This is problematic for an optimizing compiler, because if the type of a variable cannot be determined, the variable cannot always be efficiently stored, and operations performed on that variable may need to check what type the operation will be performed on.

However, dynamic typing is not the only challenge dynamic languages present. These languages often offer many other dynamic features, such as the ability to add code to a program, or even to modify existing code (e.g.: adding fields to a class, as in python)

at run-time. One notable feature found in many dynamic languages is the `eval` construct. This construct allows new code to be entered as input to the program, often directly through a command-line interface.

As recognized by the designer of the `phc` PHP compiler [BdVG09], the `eval` construct presents a major barrier to any optimization attempt, as it breaks almost all assumptions one typically makes in static intraprocedural analyses. Once once encounters a program statement containing the `eval` construct, it is essentially encountering a statement that could do almost anything to the current state of the program. Potentially, this statement could change the type of variables in the current scope, call an undetermined function in the program, or even load a new module, but none of these actions can be predicted ahead of time.

The MATLAB programming language has its share of dynamic features that pose optimization problems. For one, it does not use a classical module system, but rather one that is based on the directory tree. At any program point, a `cd` command can be issued. When this happens, all function lookups may change in every function of the program. However, this construct can take its target directory string from any source, and thus, the target directory (and hence, which bindings will be changed) cannot always be statically determined.

The MATLAB language also has a rather unusual type system which allows on-the-fly type conversions. For example, if one has a matrix containing only double values, but sets one element of this matrix to a complex value, then the whole matrix must now be able to store complex values. MATLAB makes this appear seamless to the programmer (all double matrices can store complex values), but for performance reasons, it is best to make the distinction between matrices that can and cannot contain complex values when compiling or interpreting the language.

To efficiently execute dynamic languages, virtual machines must find ways to deal with all the dynamic features these languages allow. This often entails performing dynamic analyses on programs to attempt to determine what actually happens at run- time, often through profiling. It also often means that these virtual machines must be able to adapt to situations in which certain assumptions made about the running program are found to be

invalid.

2.3 Dynamic Language Compilers

The McVM virtual machine makes an attempt at optimizing a subset of the MATLAB programming language through the use of a Just-In-Time (JIT) compiler. The idea that dynamic languages could be compiled instead of simply interpreted, either ahead of time or just-in-time is not new, as many of the older dynamic languages such as LISP have now had static compilers for years. However, much of the modern popular dynamic languages, such as PHP, Python and Ruby, have reference implementations that are still interpreted.

This is probably due to many of the optimization challenges outlined in section 2.2.1. These languages are often designed to make the programmer's task easier, making the language as dynamic as possible, with little regard to which language features are easy to optimize. Designing a compiler for such languages is difficult. If the programmer can add fields to classes, change the type of any variable, and even insert new code at any point in time, it becomes very challenging to map the programs to machine code in any kind of efficient manner. This is very different from languages like C, which make it almost trivial to map every construct to machine code. With dynamic languages, much fewer assumptions can safely be made about the semantics of the program.

Despite these challenges, in recent years, there have been many independent efforts to implement ahead of time and just-in-time compilers for languages such as PHP, Python and Ruby. The `phc` compiler [BdVG09] is a static compiler being designed for PHP. It has shown mean performance gains of 53% as compared to the reference php implementation. The PyPy project [RP06] aims to create a virtual machine for Python written in Python itself which can generate optimized code for arbitrary target languages. Psyco [Rig04] is a virtual machine which makes use of Just-In-Time compilation and Just-In-Time specialization to improve over the performance of the reference Python implementation. They show impressive speedups of up to 109 times in very specific scenarios.

There have also been successful efforts to compile MATLAB programs, both statically and

Just-In-Time. The FALCON system [DRG+95, RP96] uses type, shape and rank analysis to efficiently translate MATLAB programs into FORTRAN 90 with parallel directives. The resulting code obtained was shown to be often as fast as hand-written FORTRAN programs. The MaJIC system [AP02] combines JIT-compilation with an offline code cache maintained through speculative compilation of MATLAB code into C/FORTRAN. The Match VM project [HNK+00], which largely focuses on adaptively parallelizing MATLAB programs, also integrates a compiler which translates MATLAB source code to a lower-level intermediate form.

MaJIC is similar to our own system: a part of their approach is to generate code on-the-fly as late as possible, in order to specialize the code generation for performance. An important difference with our system is that we use a single unified JIT-compiler back-end and do not rely on intermediate languages like C or FORTRAN. We believe this allows our approach to be faster and more flexible, as there is only one execution context. In MaJIC, the JIT compiler is presented as a fallback mechanism, whereas our system is designed with the goal of utilizing the optimization opportunities made available by a JIT compiler whenever possible.

The Match VM project is similar to our own in that it is a virtual machine which includes a MATLAB compiler and performs type and shape analysis. However, the main focus of their project is not so much the optimization of dynamic code at run-time (as with ours), but the execution of MATLAB code on multiprocessor architectures using automated parallelization. The focus of their work is on developing adequate data dependency detection, resource allocation as well as job scheduling techniques.

2.4 Program Specialization

Procedure cloning is a technique by which a compiler can create specialized copies of function bodies, dividing incoming calls between the original and cloned procedures [CHK92]. This cloning technique is motivated by the fact that different call sites will make use of the same procedure to operate on different input data. Hence, it can be advantageous to

produce specialized versions of a procedure which are optimized to deal with input data possessing different characteristics. The technique essentially allows an optimizing compiler to make more assumptions about the specialized procedure bodies than it could make about the original procedure.

Such specialization techniques have been applied to languages such as SELF [CU89], Java [SC03] and C++ [PC95], most notably to reduce the overhead of virtual method calls by statically establishing which derived method will be called in a given context. In practice, this can yield very significant performance gains. Schultz and Consel report speedups of up to 300% for their specializing Java compiler [SC03]. However, it is trivial to see that program specialization does not always come for free. Just as with method inlining, it comes at the cost of code size expansion.

Specialization of cloned procedures is typically achieved through a technique known as partial evaluation. Partial evaluation is a form of program transformation in which a program or procedure is partially evaluated at compilation-time based on statically-known facts. As a part of this process, optimizations such as constant folding can be used to optimize the program. Partial evaluation can be thought of as statically "fixing" some of the inputs of a program or procedure and optimizing code by eliminating unnecessary computations which could then be performed at compile-time [JGS93]. This technique has been used by Elphick et al. [ELC03] in MPE, an online system to partially evaluate MATLAB source functions into more efficient MATLAB code.

Program specialization, as relevant as it may be to static object-oriented language, offers even more of an optimization potential for dynamic languages. This is because in dynamic languages, the source code of the original program specifies even less information about the precise semantics of the program. Furthermore, dynamic languages tend to encourage the reuse of procedures to deal with wide arrays of different types (i.e.: in MATLAB, it is conceivable that the same method could operate both on strings as well as on numeric matrices). In dynamic languages, there is more room for information about the program to be inferred, and more interpretive overhead to be potentially eliminated.

2.5 Type Inference

Type inference is the process of reconstructing type information that is missing from the lexical definition of a program based the usage of variables in the said program as well as on semantic rules inherent to the language this program was written in [DB96]. It can be thought of as a way of recapturing type information that is missing from a program's lexical specification through inference based on the type information that is provided, both implicitly through the semantics of the language, and explicitly when variables are assigned values.

Statically typed languages such as C and C++ make the types of variables almost entirely explicit, since each declaration specifies a variable type, but even in those languages, polymorphism and pointers can make the concrete type of a variable uncertain. In these languages, simple type inference schemes can help to resolve virtual function calls [BS96], thereby greatly reducing overhead. Other languages like ML and Haskell make variable types implicit, while still being statically typed. In such languages, type inference is considered essential in order to statically compile programs, and to warn programmers about potential errors ahead of time.

In dynamically typed languages, such as Python, Ruby and MATLAB, it is common for variables to have no explicit declaration point. Types are associated with values rather than variables, and thus, the type of a given variables is not fixed. In such languages, type inference can help to drastically reduce interpretive overhead. Without it, the types of all variables must be assumed to be unknown, in which case any operation performed on a variable must first check what type the variable's value currently has, and dispatch code to perform the operation as appropriate. Furthermore, if the type of a variable is unknown, it is sometimes difficult to efficiently store this variable in memory, as we do not know how much space its value will occupy.

Generally speaking, the result of a type inference analysis is a mapping of variables to the set of possible types these variables can occupy at each program point. Type inference analysis can be achieved through many different techniques. One classical way to view the

problem is as a bidirectional dataflow analysis [Sin04]. Such a dataflow analysis propagates information through the control flow graph of functions both along and against the direction of the control flow. Each program expression defines associated type propagation rules, and a fixed point computation is performed to determine the final type mapping. In such analyses, types are usually considered static. Dynamic types are typically reduced to static types through the use of a Static Single Assignment (SSA) form.

```
1  function c = foo(a, b)
2   % Perform horizontal concatenation of two matrices
3   c = [a b];
4  end
5
6  function bar(x, y)
7   % Call foo with arguments x,y
8   d = foo(x, y);
9
10  % Call foo with x,y and assert that the output is a string
11  e = foo(x, y);
12  assert(ischar(e));
13 end
```

Listing 2.3 Types and matrix concatenation in MATLAB

Listing 2.3 shows an example of a program where type inference using bidirectional flow as well as the MATLAB language rules can help us determine the types of variables. In this example, we see that the function `bar` calls `foo` twice with arguments `x`, `y`. The function `foo` simply performs horizontal matrix concatenation on its arguments and returns the result. If we look at line 12, we can see that an assertion is in place to ensure that `e` is always a string variable. Following the control flow backwards from this point, we can conclude that, provided the program is correct, `e` must be a string. Thus, from line 11, we can conclude that `x`, `y` are strings, because otherwise, the concatenation operation would not work. If we then follow the flow of the function `foo` forwards, we can conclude that `d` is also a string, since the result of the concatenation of two strings is also a string.

There are additional difficulties when performing type inference on dynamic languages. An

important problem is that due to constructs like `eval`, MATLAB's `cd`, as well as dynamic loading and reflection features, it may be impossible to know the entire call graph of a program ahead of time. Despite this, there have been efforts to statically perform type inference on dynamic languages such as MATLAB [JB01] and Ruby [FAFH09]. Although both of these approaches seem to ignore the aforementioned problem, they have shown potential in detecting type errors ahead of time.

An approach which relates more closely to the McVM project is the use of abstract interpretation in the context of type inference. Abstract interpretation is a static analysis technique which simulates the execution of a program on abstract objects rather than on real values in order to gain additional knowledge about the said program [CC77]. In the context of type inference, this means simulating the computations the program performs using abstract objects representing possible types the program values can take. Such an approach to type inference has been successfully applied to the ML programming language [GL02].

2.6 Run-time and Adaptive Optimizations

For a long time, statically compiled languages have been widely perceived as better performing than languages executing under virtual machines. However, this perception is changing. Programming languages like Sun Microsystem's Java have shown that they can in some cases perform better than statically compiled languages[2]. The reason for this is that conceptually, VMs and JIT compilers have a big advantage over static compilers: whereas static compilers can make educated guesses about a program's future state, virtual machines can actually observe this state as the program executes.

This advantage of VMs over static compilers can be technically difficult to materialize. The more dynamic the implemented language is, the less information is known statically about the semantics of programs, the more work the virtual machine needs to do in order to efficiently execute program code. This translates into increased complexity of the VM's implementation. Despite this, virtual machines are increasingly seen as a way to make

[2] Performance of Java versus C++:
http://www.idiom.com/ zilla/Computer/javaCbenchmark.html

software more portable, and to overcome the limits of static compilers.

The main limitation that static compilers have to face is that no matter how many analyses they can perform on a program, these analyses must be conservative in order to maintain correctness. Static compilers cannot know what they cannot conservatively infer, which, in practice, can be very limiting, especially where dynamic languages are concerned. Virtual machines, on the other hand, can actively gather information about a program's state through instrumentation and profiling.

Several virtual machines, such as Jikes RVM, have implemented systems where methods of a program can be compiled at several different optimization levels, and the virtual machine will choose which level to compile a given method at based on profiling information [AAF+05, SYK+05, GV08]. Jikes RVM uses call-stack sampling and a cost-benefit prediction model, and can recompile methods if its estimates change. The general idea is that compiling (or recompiling) methods at higher optimization levels is costly, and that the speedup obtained by compiling a method at a higher optimization level should justify this cost.

Recompiling methods at run-time as is done in some Java virtual machines may seem trivial to perform, but it is sometimes challenging, because a method that we wish to recompile may be currently executing. This has been made feasible through On-Stack Replacement (OSR), a technique that allows the call frame of an executing function to be replaced so that the new state is as expected by the recompiled method [FQ03]. This technique has been implemented into Jikes RVM and is necessary to make run-time recompilation possible.

Going beyond the idea of multiple optimization levels, Lee & Leone [LL96] have devised an ML compiler that defers compilation of certain portions of code. That is, most of the program is natively compiled, but some portions of it will be compiled at run-time. They have shown that despite the cost of run-time code generation, this can yield significant performance benefits, because the generated code can benefit from the knowledge of invariants that were not statically known. This work has later been extended by others to specialize Java bytecode at run-time, inside of a virtual machine [MY99].

A more recent effort applies run-time code generation to a dynamic language. The Psyco

python virtual machine implements specialization by need. This new specialization technique involves interleaving program specialization and execution [Rig04]. Their specializer can inquire about facts such as the type of variables while a procedure is executing, and depending on the result, potentially modify the compiled code of the said procedure to be more efficient. One of their main goals was to eliminate much interpretive overhead through the use of JIT compilation without sacrificing dynamic features of the language.

Similarly to the Psyco effort, those behind the TraceMonkey VM for the JavaScript language have focused their efforts on just-in-time specialization based on type information in order to increase performance. Their system is based on a bytecode interpreter that can identify frequently executed bytecode sequences (traces) going through loops and compile them to efficient native code based on collected type information [GES+09]. A crucial assumption of their system is that programs will spend most of their time in loops, and that the types of variables will remain mostly stable through the execution of loops. They have achieved speedups of up to 25 times on some benchmarks. However, their current VM does poorly on benchmarks making extensive use of recursion.

It is technically possible to go very far with the use of adaptive optimizations. One possible approach to optimization, as opposed to the use of fixed heuristics, is to dynamically explore the space of all optimizations that can be applied to a given program or method. Such techniques, often classified as forms of iterative optimization, can iteratively generate multiple versions of a given code segment and evaluate the performance of each in order to find the best performing one [VE01]. More recently, it has been shown that using machine learning techniques to focus iterative optimization can drastically reduce the number of versions needing to be evaluated [ABC+06].

Chapter 3

Supported Language Features

The MATLAB programming language is highly elaborate. It integrates specialized matrix computation functionality, multiple ways to achieve object oriented programming, but also many more features, such as function handles, closures, as well as 2D and 3D visualization. Beyond this, it also provides hundreds of library functions as well as a way to incorporate Java code into MATLAB programs.

Due to the highly complex nature of this language, and the limitations in the time frame allotted to the completion of a Master's thesis, the scope of the language McVM supports has been restricted to a subset of the actual MATLAB language. Nevertheless, this subset is significant enough to be useful for real scientific computations, as demonstrated by the fairly extensive set of benchmarks we support.

This chapter outlines the main differences between our implementation and that of Mathworks. It provides important details about the features of MATLAB we support and those we do not. We discuss supported data types, provided library functions as well as unsupported language features. We also outlines the differences in semantics between our implementation and the original MATLAB language. We explain how and why our execution model and our implementation of specific MATLAB constructs differs from that of Mathworks.

3.1 Supported Types

MATLAB uses double precision matrices by default; if the programmer wants any other kind of numerical matrix (excluding complex matrices), he has to explicitly request it. Implementing new matrix types means writing programming code to handle each type of matrix appropriately for each kind of matrix operation the language offers. Hence, we have chosen to support only the core matrix types in order to reduce the required programming effort and simplify our implementation.

Support is not provided for integer matrices, or floating-point matrices with single precision. There are also no object-oriented features at this point, and thus no structs or classes. The matrix types that are supported are logical arrays, character arrays, double precision floating-point matrices (64-bit per element) and double precision complex number matrices (128-bit per element). McVM also integrates cell array (matrices of pointers) and function handle types.

3.2 Supported Features

Our implementation supports most of the non object-oriented features of the MATLAB language. The traditional `if`, `for`, `while` and `switch` control statements are all supported. The use of range expressions in array indexing to create sub-arrays is supported. Extensive support is provided for cell arrays as well as for matrix concatenation operations. Basic support for string handling and file I/O is also provided.

We also provide support for calling functions with a variable number of arguments and returning a variable number of arguments from functions. We allow the use nested function as well as the use of multiple functions per source M-file. It is possible to create function handles to any function, including library functions. Closures can also be created by creating a handle to a nested function or by using lambda expressions.

3.3 McVM's Execution Model

In MATLAB, scripts operate on the caller's environment and nested functions can modify variables in the calling function (see section 2.1.3). Some library functions such as `eval` can also modify the caller's environment. For simplicity, and for philosophical reasons, we have decided to go with a slightly different execution model. We find that allowing callees to modify a caller function's environment violates the encapsulation principle. It allows unintuitive behaviors where callees can have unforeseen side effects, and is also very unfriendly to optimizing compilers.

In McVM, no callee may modify variables in the caller's scope or environment. Scripts operate in the global environment, and can only touch global variables. The same goes for library functions. As for nested functions, they are able to read variables from the caller's environment, but assigning them a value will create a binding in the local environment instead of modifying the value of the caller's variables. This preserves a sense of separation between caller and callee.

We have found that these changes are not an issue in practice. None of our benchmark programs make use of these kinds of side effects. Programmers rarely seem to intend for a callee to modify variables in the caller. In fact, in MATLAB, it is an issue they have to be careful about (to avoid unpredictable errors). The limitations we have placed on library functions also make them more compatible with our optimization scheme: they cannot unpredictably assign any value to any local variable.

3.4 Library Functions Provided

MATLAB provides a wide library of basic functions for I/O and common operations. Table 3.1 shows a list of the MATLAB library functions we support in McVM. These are re-implementations of the same functions provided as part of the Mathworks MATLAB environment. At this point, not all of these functions provide all the functionality of their Mathworks equivalent, but they support the most common use cases. Fairly extensive doc-

Table 3.1 List of supported library functions

abs	eval	fprintf	min	sin
any	eps	i	mod	size
bitwsand	exist	iscell	not	sort
blkdiag	exp	isempty	num2str	sprintf
cd	eye	isequal	numel	squeeze
ceil	false	isnumeric	ones	sqrt
cell	fclose	length	pi	strcat
clock	feval	load	pwd	strcmp
cos	find	log2	rand	sum
diag	fix	ls	reshape	system
disp	floor	max	round	tic
dot	fopen	mean	sign	toc

umentation about the behavior of these library functions is provided on the Mathworks website, as part of the online MATLAB documentation.

3.5 Unsupported Features

Although McVM is capable of running non-trivial benchmarks, from a language standpoint we have only implemented a small fraction of the hundreds of MATLAB library functions. McVM does not yet support object oriented programming and does not handle integration with languages other than C and C++. Those are some of the most obvious limitations of our implementation. Our system also does not provide a development environment as rich as that of MATLAB: we do not supply a code editor, a debugger or 2D/3D plotting features.

Some of these limitations will likely be overcome in the near future. Implementing new library functions in C++ is currently very simple. It would be relatively easy to add object oriented support to McVM. Basic support for MATLAB structs (without JIT compiler optimizations) could likely be implemented in less than 1000 lines of code. Supports for classes could be added relatively easily as well; it could likely be be implemented in less than 2500 lines of code. Other features, such as 2D plotting, are being implemented by other members of the McLab team as this thesis is being written.

3.6 Dynamic Features

MATLAB contains a number of dynamic features we have limited in order to ensure good performance. We have briefly mentioned in section 3.3 that we have chosen to restrict the `eval` language construct so that it can only affect global variables. From an implementation perspective, this was done because it fits better with our optimization model. The `eval` construct, in its unrestricted form, can read or write to any local variable based on input, in ways that often cannot be predicted when a function is compiled. This destroys much of the information an optimizing compiler could rely on (type information, reaching definitions, live variables, etc.). Thus, we have chosen a compromise which makes optimization easier and still allows `eval` to be useful.

Another such dynamic feature found in the MATLAB language is the `cd` function. This function changes the current directory based on input, and thus, changes the currently visible functions. This function is also unfriendly for an optimizer, because it means that unless we can be certain `cd` will never be called while a function runs, we cannot know at compile time that a given symbol will always refer to the same function. This is particularly problematic because we may want to know ahead of time what types a given callee function takes as input and returns as output, for example.

At this time, we have chosen to circumvent this issue by making it so that symbols are looked up and bound to specific callee functions at compilation time. Thus, if the `cd` command is issued, it cannot undo function call bindings that have already been made. It can, however, allow access to functions that were not defined at compilation time. This behavior may seem restrictive, but we have found that it works well in practice, and is perhaps more intuitive than MATLAB's default behavior.

The compromises we have made correspond to real issues when it comes to compiling dynamic languages. It may be possible to avoid making them in the future by resorting to schemes where the JIT compiler will "fall back" to interpretation when commands like `cd` and `eval` are issued. Alternatively, techniques like on-stack replacement may provide a way to circumvent the limitations of our compilation model (see section 8.2.5). However,

at this time, we consider them reasonable as most of the limitations they impose can be easily circumvented and rarely create compatibility issues in practice.

Chapter 4

Virtual Machine Architecture

This chapter examines the design of our virtual machine and JIT compiler in detail. We begin with a discussion of the design goals on which our design decisions were based. This is followed by an overview of the overall architecture of our virtual machine and its division into sub-components. We then explain the interface between our VM and the front-end of our compiler, the intermediate representation used by our system and the design of our interpreter and run-time environment, as well as our interpretation strategy. Finally, we conclude with a detailed discussion of the design of our JIT compiler and its compilation strategy.

4.1 Design Goals

One of the most important goals behind our virtual machine design was to aim for a simple and easily extensible design. Our architecture reflects this goal: the JIT compiler was built as an extension of the interpreter, somewhat similar to the way the `phc` compiler was built [BdVG09]. This makes it possible to add new data types or statements to the supported language by modifying only the interpreter, and not the JIT compiler. The JIT compiler can later be modified, if necessary, to gain performance benefits from additional optimization opportunities.

Language design largely focuses on performance. We have taken inspiration from the C++ programming language, which was designed with the clearly stated objective that the programmer should not pay a performance cost for features he or she does not use. In McVM, we have striven to minimize performance costs incurred by MATLAB's more complex and dynamic language features. This is achieved partly through the use of our type-driven just-in-time specialization scheme (see section 4.7.3), and partly through our interpreter fallback mechanism (see section 4.7.1). Some costly dynamic features are still interpreted, but the JIT compiler has been designed to optimize the clearest and most common performance bottlenecks as well as possible.

We have also striven to keep our system interactive. Our virtual machine provides an interactive environment where commands can be typed at the prompt, and new code can be introduced on the fly, similarly to Mathworks MATLAB implementation. This is unlike some previous attempts at compiling MATLAB into a target language like FORTRAN [DRG+95, RP96], for example. These approaches can yield great performance benefits, but they require whole-program static analyses which are incompatible with an interactive environment.

Some technical choices were made to balance compatibility with the MATLAB language with design simplicity and "cleanliness". As such, we do not implement a "pure" subset of MATLAB (see chapter 3). Rather, we implement a close variant which we consider to be more intuitive for programmers, less error-prone and more practical from an implementation perspective. Our language variant matches the MATLAB semantics in all MATLAB benchmarks programs with which we have had to deal, and most likely, in the vast majority of real-world programs.

We note that reducing compilation times was not a high priority when designing our VM. MATLAB being a language aimed at the scientific community, we have judged objective performance in terms of total running time to be more important than real-time interactivity. Scientific programs can potentially run over very large data sets, sometimes for hours. Thus, we have deemed it acceptable to use up to several seconds of compilation time for a given program, since it could, in the end, save up to several hours of total execution time.

4.2 Architecture Overview

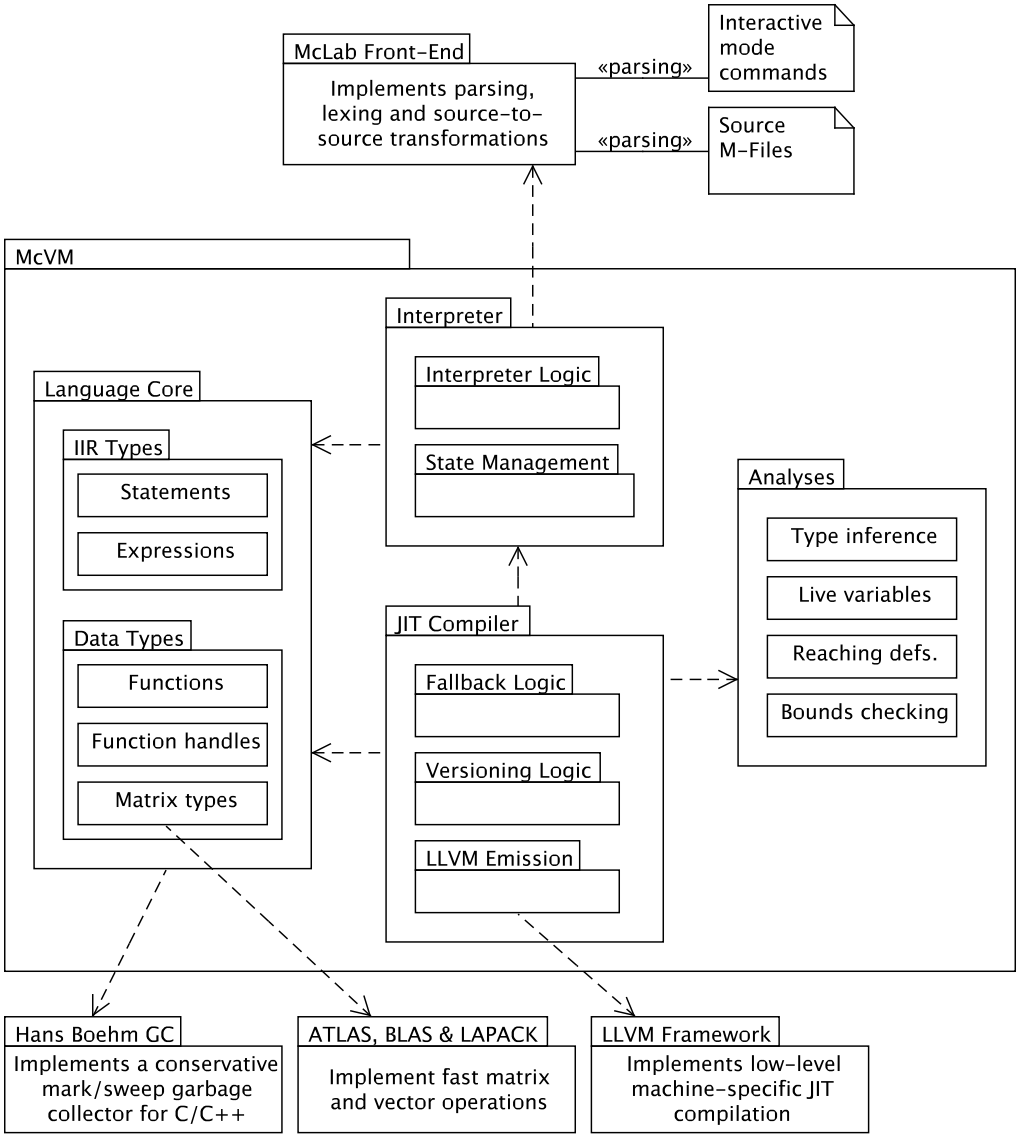

Figure 4.1 Structure of the McVM Virtual Machine

The illustration in figure 4.1 shows the overall structure of the McVM virtual machine implementation. At the core, McVM's implementation of matrix types relies directly on a set of mathematical libraries (ATLAS, BLAS and LAPACK) to implement fast matrix

and vector operations (matrix multiplication, scalar multiplication, etc.). All language data types and Internal Intermediate Representation (IIR) types use the Boehm garbage collector library for garbage collection [BS07]. Our JIT compiler also relies on the LLVM framework to implement low-level JIT compilation (emission of machine-specific code) [Lat02]. McVM also depends on the McLab front-end, because the interpreter uses it to parse interactive-mode commands as well as source code in the form of M-files.

Internally, both the interpreter and the JIT compiler rely on the language core to define the basic primitives on which they operate. This is the Internal Intermediate Representation (IIR) tree, which defines the forms valid programs can take, and the primitive data types the language supports. The JIT compiler itself depends on the interpreter because it does not emit compiled code for all operations, it sometimes uses interpreter fallback to evaluate code for which there is not yet compiler support.

The functionality of the interpreter is divided into interpretation logic and state management (house keeping). The JIT compiler manages the function versioning system, emits LLVM code for the statements it can compile, and performs interpreter fallback for those it cannot. The JIT compiler also largely relies on a set of analyses to gain additional information about source programs being compiled. These analyses (live variables, reaching definitions, bounds check alimination and type inference) are crucial to generate high-performance code.

We have chosen to implement McVM entirely in C++. This choice was made in part because C++ allows low-level access to the way its data types are stored in memory and offers a relatively high level of performance, both very useful characteristics for the implementation of a virtual machine. However, another important motivating factor was the availability of the LLVM framework itself. This powerful compiler building framework was crucial to the timely implementation of our JIT compiler.

4.3 Front-End Interface

The McVM virtual machine only implements the back-end part of our MATLAB compiler. The actual lexing and parsing into an Abstract Syntax Tree (AST) is done in a front-end program, implemented in the Java programming language. This front-end program also incorporates several analyses meant to perform high-level transformations and optimizations on MATLAB source code.

When the virtual machine is started, it launches a front-end program instance and connects to it through a TCP socket. This interface was made socket-based for portability. When source files or commands need to be parsed, a request is sent to the front-end, which then returns the source code in a pre-parsed AST high-level Intermediate Representation (IR) encoded in XML format.

4.4 Intermediate Representation

The internal intermediate representation (IIR) used by McVM (by the interpreter, the JIT compiler and the analyses), is a simplified and transformed version of the original source code's Abstract Syntax Tree (AST). Due to its close resemblance to the original source, it is machine-agnostic in nature and can easily be printed in human-readable form.

In terms of implementation, all IIR nodes inherit from a common superclass IIRNode. The IIR is defined by the IIRNode class and all its subclasses. These are organized in a hierarchical order to define functions, statement sequences, statements (including control statements) and expressions. The IIRNode class and all its subclasses are garbage-collected, making it easier to write IIR transformation passes, as it eliminates the need to keep track of which nodes should be deleted upon replacement.

In order to keep the virtual machine design simple, the input AST (produced by parsing the source), is simplified in a few ways. Among these transformations, `if-elseif-else` chains are transformed into multiple nested `if-else` statements with only two possible branches each, `switch` statements are transformed into equivalent language constructs and

both `for` and `while` loops are transformed into a single loop form that is structurally closer to the `while` construct.

Some parts of the IIR tree are also annotated to make subsequent execution or compilation easier. For example, all instances of the `end` keyword, used to mark the end of a numeric range (effectively an expression), are annotated such that all the possible symbols it can refer to are all known at execution time. This is useful because the possible bindings of the keyword changes based on its position in the source code. These bindings would be difficult to keep track of in our interpreter, because the interpreter only looks at small IIR subtrees during execution, but is fairly easy to pre-compute ahead of time.

The interpreter component of McVM uses the IIR tree as-is, after the previously described simplification and annotation passes have been performed. However, the JIT compiler requires further transformations. Namely, prior to JIT-compilation, the IIR tree is transformed so that all expressions are in 3-address form. This is a simplified form where expressions cannot contain sub-expressions other than symbols, numerical ranges and constants [ASU86]. Note that while 3-address form expressions are usually limited to a maximum of 3 operands, some expressions, such as function calls, are allowed an potentially unlimited number of operands.

4.5 Interpreter Design

The interpreter component of McVM is implemented in a very simple and straightforward way. It performs naive interpretation directly on the IIR tree. This interpretation is performed in a recursive manner, that is, evaluating a statement results in a pre-order traversal of the corresponding IIR subtree of expressions. When a function is executed, all of its statements are recursively interpreted in sequence.

Trees of expressions are used to represent compound expressions. Leaf nodes in such a tree can represent constant values or variables. Variables are bound to data objects which store their type and value, and any leaf node operating on one or more variables generates a new such object to store the result of the operation. Intermediate nodes represent function

calls or arithmetic and logical operators that operate on the results of the evaluation of their subtrees, again producing data objects to store the result of the operation. All data objects are garbage-collected.

The interpreter manages program variables, both local and global, through "environment" objects. These objects are essentially hash-maps that map symbol identifiers to pointers to objects. They are a straightforward way of binding and looking up symbols at run-time. They offer the advantage of a fairly quick O(1) lookup time, and can be grown at run-time without a need to pre-allocate or pre-calculate where bindings are to be stored.

Environments are also extensible, meaning an environment object can have a parent environment. That is, environments are a simple mechanism to implement recursive scoping: when looking up a symbol in a given environment object, if it is not found, the symbol will then be recursively looked up in its parent. Thus, in the interpreter, stack frames for functions are typically implemented by creating an environment object that extends another global environment object (containing global variable bindings) when a function is called.

This mechanism has the advantage that it is highly flexible. For example, it makes it trivial to implement an "eval" command that can create unanticipated bindings in a function's call environment at run-time. It also serves as a useful "black-box" implementation of symbol bindings: functions with unbound symbols can be executed without regard for where and when those symbols will be bound.

In addition to its role as a basis or reference implementation for the implementation of our JIT compiler, the interpreter also serves housekeeping roles. It takes care of loading MATLAB files on-demand, executing interactive-mode commands, hosting library function bindings, maintaining bindings to global variables, etc. It also has the role of filling in and interpreting parts of the language that the JIT compiler cannot fully compile.

Library functions in McVM are currently all implemented as native C++ functions, so as to maximize performance. These functions do not operate on environment objects like program functions, but rather take in a dynamically allocated array of object pointers as input, and also return such an array as output. As such, they have support for multiple input and output values, as well as variable arity.

4.6 Program Analysis Framework

The JIT compiler component of McVM requires multiple analyses to be run on the IIR tree. These analyses, which include live variable analysis, reaching definitions analysis, type inference and bounds check elimination serve to gather additional information about the input source which can then be used to allow specific optimizations. All of these analyses were designed as forward or backward flow analyses, which are typically performed over a transformed Intermediate Representation (IR) format called a Control Flow Graph (CFG).

Instead of transforming our IIR tree into a CFG, we have chosen to perform structured flow analyses directly on the IIR. This turns out to be surprisingly easy in practice. The only constructs requiring a fixed point computation are loops. The information flow is otherwise linear from one statement to the next. Special care must be taken to gather data flow sets at return, break and continue points, because these statements disrupt the normal control flow.

While analyzing one specific function, some analyses (such as type inference) may require callees to be analyzed. This has the potential of creating infinite loops, or to waste computational time analyzing specific functions over and over (if a function is called by many others). To get around this issue, we have implemented a singleton class (the analysis manager) to cache analysis results. Whenever a function is analyzed, the analysis results are cached. Different results may be cached for each function version (see section 4.7.3).

The analysis manager also keeps track of which functions are currently being analyzed. This way, if analysis information is requested about a function which is already being analyzed by the same analysis, the analysis manager will return a “bottom” value to avoid infinite loop scenarios. The “bottom” value is defined on a per-analysis basis. It can either mean no information, or whatever the analysis can assert about a function without requiring recursive calls.

4.7 JIT Compiler Design

This section discusses the design and internal workings of our JIT compiler. We discuss the incremental strategy used to build our compiler. This is followed by discussions of its code generation strategy, our just-in-time code versioning system, and the additional optimizations our JIT compiler benefits from. The workings of the type inference system used by our JIT compiler are discussed in chapter 5.

4.7.1 Incremental Construction

We have chosen to build our JIT compiler on top of the interpreter, as an incremental process. This essentially means that our interpreter was designed, at the core, to be able to always fall back to interpreting sections of code it cannot compile, mixing sections of both compiled and interpreted code as part of the compilation of a given function. This is an incredibly convenient design, because it has allowed us to build the JIT compiler in multiple steps, while still being able to test its proper functioning at each point, and able to compile all programs the interpreter can run.

The starting point for our JIT was a compiler that interprets every single statement. For each statement it needs to compile, the JIT inserts a call to the proper interpreter function to interpret this statement. From this point, it is easy to add compilation support for any kind of statement, without ever needing to be able to compile all existing statement kinds, or even all possible forms of a given statement. For example, a early version of our JIT supported compilation of assignment statements assigning to one variable only; assignment statements assigning to multiple variables were interpreted.

The only drawback of this interpreter fallback system is that special care had to be taken to interface with the interpreter. Interpreter functions often take as input a statement or expression to evaluate along with an environment object containing the current variable bindings (see section 4.5). The JIT compiler is designed so as to store variables values in registers, with the appropriate type (i.e.: values known to be scalar integer are stored as integers, not as pointers to matrix objects). Thus, before an interpreter function is called

to evaluate a statement or expression, all the variables the statement or expression uses must be stored into value objects and bound into an environment object. In the case of assignment statements, the JIT must also keep track of which variables will then be written in that environment object. All of this is managed through a data structure we call the variable table.

4.7.2 Code Generation Strategy

Our JIT compiler is built on top of the LLVM compiler-building framework [Lat02]. This framework handles the low-level parts of the code generation. As input, it requires program code specified in a RISC-like Static Single Assignment (SSA) form, and translates it, at run-time, into machine-specific code. It also performs optimization passes on the code, both at the machine code level, and higher levels. It can, for example, perform constant propagation, dead code elimination and eliminate some redundant operations. As such, it greatly simplifies the construction of a JIT compiler by completely hiding much of the platform-specific details and providing low-level optimizations. Broadly speaking, our work in terms of code generation consists of translating our IIR tree into efficient input code for LLVM.

The JIT compiler generates code for a function in a recursive manner, by traversing its IIR tree. Specialized methods generate code for each kind of statement and expression as the tree is traversed. Whenever the JIT compiler does not know how to compile efficient code for a statement or expression, fallback code is generated which will invoke the interpreter to execute that specific statement or expression. At each step of the compilation process, the JIT needs to take special care to know how each live variable is stored. This is achieved through the variable table, which maps symbols (the variable names) to pairs of LLVM value objects and McVM type identifiers.

The variable table is needed because variables can be stored in different ways. They could be stored in an environment object (see section 4.5), or they could be stored in an LLVM virtual register (on the stack, essentially). If they are stored on the stack, they could either be stored as pointers to objects stored on the heap (necessary for matrix objects), or as

scalar integer or floating-point values, if we know enough about their type. The LLVM value object tells us how a variable is stored, but the McVM type identifier is needed to distinguish what actual language type the variable has. Note that the specific type may be unknown in some cases, if the available type information is insufficient to determine it (see chapter 5).

When interpreter fallback code is generated for an expression or a statement, the variables this expression or statement may use are written in an environment object, and the variable table is updated to reflect this. If these variables are needed at a later point, they will be read from the environment object and converted to the most efficient storage mode, at which point the variable table will be again updated to reflect this change. Note that if a function never needs to perform any interpreter fallback operations no environment object will be created for this function. Thus, the interpreter fallback mechanism does not incur any penalty for functions which do not need to use it.

Special care needs to be taken when compiling `if-else` and loop statements, because these produce multiple possible control-flow paths. The JIT compiler actually generates a variable table for each possible path. Thus, an `if-else` statement produces two variable tables which need to be merged into one. This is done such that if a variable is locally stored (not in an environment object) in any of the variable maps needing to be merged, it will be locally stored in the final variable map. The merging process also ensures that the optimal storage mode is selected for all variables. Note that in practice, the way variables are stored most often does not change at merge points.

In the case of loops, the process is similar, except that there may be many more loop exits due to the existence of a `break` statement in MATLAB. Control-flow paths also need to be merged at the loop entry. In this case, this is done so that the variable map states of all control-flow paths match that of the original control-flow path that entered the loop. This is because the loop code is generated first according to the entry control-flow path leading to the loop. Thus, for this code to remain valid, any other control-flow paths leading to the loop entry need to match the variable map state of the entry path. Our approach permits the compilation of loops in a single code generation pass.

One important optimization in our compiler is the use of type information to generate optimized code for binary expressions. MATLAB makes the distinction between scalar values and larger matrices invisible to the programmer, but we attempt to store scalar variables on the stack whenever possible, for performance reasons. Whenever the JIT compiler knows that a binary expression occurs between two scalar variables, it will attempt to generate specific machine instructions to perform this operation. This allows scalar additions, subtractions, multiplications and divisions to be compiled efficiently, without generating intermediate matrix objects to store the result.

4.7.3 Function Versioning

The researchers behind the Psyco [Rig04] and TraceMonkey [GES+09] virtual machines have realized that in order to efficiently compile programs written in a dynamic language, it is essential to expose information about concrete types used in the said programs. We have independently come to the same conclusion regarding the compilation of MATLAB programs and devised a just-in-time specialization scheme for our JIT compiler which makes use of both run-time and inferred information about program types. Our specialization scheme bears some similarity to the one used by the Psyco VM.

Our specialization strategy begins by "trapping" function calls made through the interpreter. Any command made in interactive mode runs through the interpreter. If the command is a call to a function (and not a script), the interpreter will (if JIT compilation is enabled) try to let the JIT compiler handle the call. When this happens, the JIT compiler will build an argument type string from the input arguments to the function. It will then attempt to locate a previously compiled version of the function matching this argument type string. If none is found, a new version will be compiled, specialized for the given argument types. Once compiled, the function will be called with the specified arguments.

When compiling a specialized version of a function for a specific input argument type string, the JIT compiler makes use of a type inference analysis pass (see chapter 5) to infer information about the possible types of variables at every point in the function body. The JIT compiler will then make use of this type information to generate more efficient code

than would be possible without it. Dispatching overhead can be eliminated, and some scalar variables can be stored directly on the stack, instead of being stored as objects (of unknown type) on the heap.

While compiling specialized function versions, the JIT compiler will also be able to determine information about the input parameters of functions called from the function being compiled. It can then compile specialized versions of the callees of the current function as well. Thus, our scheme specializes functions based on input argument types when they are called, or when the JIT compiler knows they could be called from the function currently being compiled. The vast majority of callees can be resolved at compilation time, and so most of the compilation and specialization happens when the root function of the call graph is called. If a new call to a function is made through the interpreter with input arguments for which no specialized version exists, a new one will be compiled. If the function has any callees, new versions may or may not be recompiled for those as well, depending on whether the input arguments to those would change or not.

```
1  function s = sumvals(start, step, stop)
2
3     i = start;
4     s = i;
5
6     while i < stop
7        i = i + step;
8        s = s + i;
9     end
10
11 end
```

Listing 4.1 The sumvals function

```
1  function s <scalar int> = sumvals(start <scalar int>, step <scalar
       int>, stop <scalar int>)
2
3     i <scalar int> = start;
4     s <scalar int> = i;
5
```

```
6     while i < stop
7        i <scalar int> = i + step;
8        s <scalar int> = s + i;
9     end
10
11 end
```

Listing 4.2 The type-annotated sumvals function

As an example of how our function versioning works, take a look at the `sumvals` function in listing 4.1. This function is meant to sum numerical values in the range going from `start` to `stop`, inclusively. Since MATLAB does not make variable types explicit, we do not know what types any of the variables (including the input parameters) have. The function could, in theory, sum over integers, floating-point values, matrices of integers or floating-point values, or even complex numbers. Thus, it cannot be efficiently compiled as-is. The JIT compiler would have to assume the worst-case, where we are summing over matrices, which clearly cannot fit on the stack and must be stored on the heap. It would also need to make calls to expensive dispatching code for every operation performed on any variable.

Now, imagine that this function is called from interactive mode as follows: `s = sumvals(1, 1, 6);`. From this point, we know that each argument to the call is a scalar integer. The JIT compiler can then use type inference to logically conclude that all variables in the function are in fact scalar integers. From this point, we can imagine a type-annotated version of the `sumvals` function (see listing 4.2). This function can be efficiently compiled. All variables are easily stored on the stack, and there is no need to make expensive dispatching calls, because there are efficient machine instructions to add and compare scalar integer values. Thus, a specialized version can be compiled based on the input argument types, which will likely be hundreds of times faster than its non-specialized counterpart would have been.

The obvious downside is that this scheme has the potential to generate many specialized versions of a function, with each requiring additional compilation time, and potentially impacting the performance of the instruction cache. We will see that this is not the case in practice (see chapter 6). From our observations, MATLAB programs tend to have few

long functions and fewer call sites than code written in other programming languages (see section 2.1.5). It also seems that MATLAB programmers tend to use functions for very specific purpose (e.g.: integrating a specific kind of function), and thus rarely call functions with very different input argument types. The main strength of our specialization scheme is that most function get specialized only once, and that the vast majority of the symbols in these functions can be typed if the input argument types are known.

4.7.4 Additional Optimizations

We have already mentioned in section 4.7.2 that our JIT compiler can sometimes make use of type information to map scalar arithmetic operations to efficient machine instructions. This is not the only optimization that makes use of this information, however. We have also implemented optimized array access operations and optimized mapping of library function calls. These optimizations also rely on information provided by the type inference system and function versioning framework and try to generate optimized code when enough information is available to guarantee their soundness.

MATLAB possesses a sophisticated array indexing scheme that allows programmers to read or write to n-dimensional slices (sub-arrays) based on ranges of indices specified independently for each dimension. This system makes the life of MATLAB programmers easier, but requires fairly complex logic to work properly. In many cases, our JIT compiler uses the interpreter to evaluate complex array reads and writes, which comes with some overhead. However, when the JIT compiler can determine that a matrix is being written to or read from using scalar indices (e.g.: `x = a(i);` where `i` is a scalar), it knows that the value being read or written must also be a scalar. In this case, if the type of the matrix is known, it can generate optimized code to read or write the said value directly. It can also use the bounds check elimination analysis to eliminate unnecessary checks.

Library functions are implemented in our virtual machine as native C++ functions which take as input (and return as output) dynamically allocated arrays of pointers to data objects. This is sometimes largely inefficient because each call to these functions requires the arrays to be allocated. Also, in the case of scalar values stored on the stack, it means they must

be wrapped into objects before being passed along, and that return values may need to be unwrapped in the reverse fashion. Furthermore, the library functions we provide are also expected to be able to operate on both matrices and scalar values, and thus incur some overhead, because they must be able to loop over all the values present in a matrix.

To address these issues, we have devised a scheme where optimized versions of some library functions can be registered with the JIT compiler based on the types of input arguments they take as input, and the types of their return values. When a library function call is encountered, the JIT compiler will attempt to locate an optimized version of this function with input types and return value types matching those required in the current calling context. If one is found, it will be called instead of the general-purpose version of the library function. An obvious example where this is beneficial is in the case of functions like `abs` or `sin`. If these functions are called on scalar values, we can directly insert a call to the native C++ version of these library functions.

Chapter 5

Type Inference System

In this chapter, we explain in detail the type inference analysis used by the just-in-time type-based specialization mechanism built into our JIT compiler. We begin with a simple description of the analysis, what it infers, how it is different from other type inference schemes and how it is used by the JIT compiler. This is followed by a more formal explanation of the actual analysis process and its roots in abstract interpretation. We supply a detailed example of our type inference analysis being applied to a simple MATLAB program. Finally, we conclude with an explanation of the validation strategy used to empirically validate the correctness of the analysis.

5.1 Analysis Design

The type inference analysis our JIT compiler employs was designed to rapidly infer types as precisely as possible, without making too many assumptions about the programs being analyzed. It works on a per-function basis, with the assumption that the whole program is not necessarily known at run-time, and new functions could be loaded at any time. The analysis assumes that the set of possible types for each input argument of a given function are known, and infers the set of possible types for every variable at every point (before and after every statement) in the function, given those possible input argument types.

The type analysis can be run multiple times for the same function, given different possible input types each time, thus inferring the types for multiple versions of the same function. It can also be provided with different levels of information about the input types. It is possible, for example, to specify that we know nothing of the possible types of a given input argument. It is a conservative analysis, and thus its results are provably true in every case, for any input values, provided that the input type restrictions are met.

The analysis is based on abstract interpretation. That is, it is a forward flow analysis which propagates sets of possible types for variables through every possible branch of a given function, through every statement of that function. While doing this, it is simulating the effect that these statements would have on the possible types of the variables present in the function. The result of every operator is accounted for, and thus, the analysis is, in a way, "simulating" what would be happening at run-time. It performs this simulation in a such a way that the results are always valid, accounting for every branch that could possibly be taken at any point.

This is different from most other type inference analyses, which often rely on whole-program analysis, assuming the entire program is known at run-time, and use a constraint-solving mechanism to narrow down the possible types of variables. It is also different because it takes into account that the type of a given variable can be different at multiple points of a function. Finally, unlike some type analyses, it does not assume that the program is correct. Should there be a programming error (resulting in an exception being thrown at run-time) in a program's code, the analysis results are guaranteed to be correct up to that point.

Our JIT compiler uses data provided by this type inference analysis to implement a just-in-time function specialization scheme (see section 4.7.3). The more information the analysis provides about the concrete types of program variables, the more interpretive dispatching and storage overhead can be eliminated, and the faster the resulting compiled code will be. In practice, it can make an enormous difference in terms of real-world performance results (see chapter 6).

5.2 Flow Analysis Specifics

5.2.1 Abstract Domain

Abstract interpretation is a way to simulate the execution of programs over an abstract domain. In the real domain of MATLAB programs, variables at different program points are bound to actual values (data objects). In our abstract domain, variables instead map to sets of possible types that they could hold at different program points. These sets contain zero or more type object storing information about a potential type the specific variable can have.

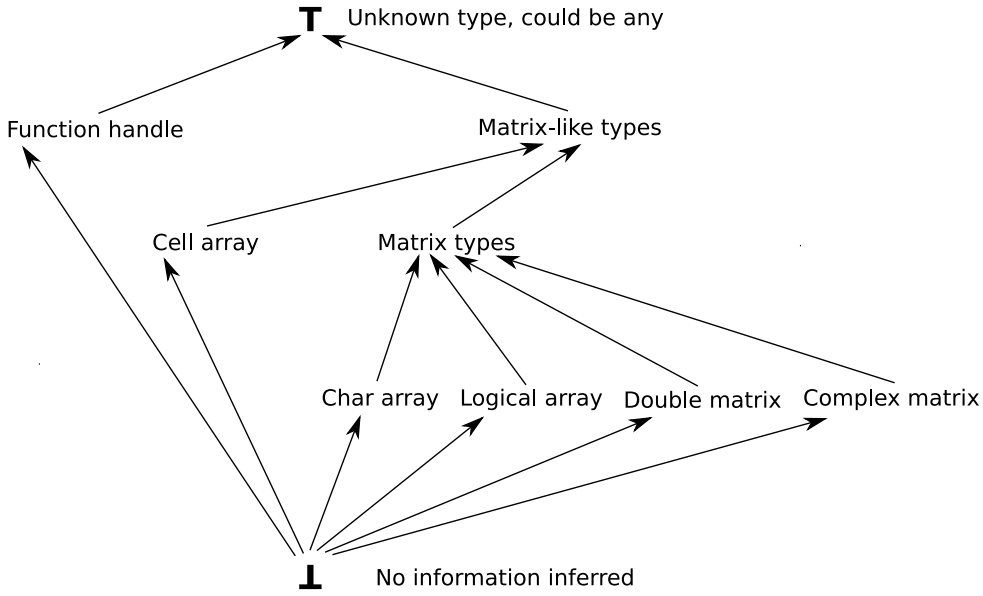

Figure 5.1 Hierarchical lattice of McVM types

Each type object in a set represents a specific MATLAB language type, such as character array, floating-point matrix, complex number matrix, etc. Figure 5.1 represents the hierarchical type lattice of McVM types these objects could have. If a type set contains multiple type objects, it means that the variable whose potential types are represented by the set at that program point could be one of the several types represented by each object in the set. The empty set is the $\perp$ element of the type lattice, representing situations where no infor-

mation has been computed yet. The set of all type objects is the $\top$ element of the lattice, representing the situation where the type of a variable cannot be determined.

Table 5.1 Description of type object fields

Field	Meaning/Description	Default
type	An element of the set of possible McVM data types.	Undefined
2D	Flag whose value applies to matrix types only. A True value indicates that the matrix has at most two dimensions. False means it is not known how many dimensions the matrix has.	False (unknown)
scalar	Flag whose value applies to matrix types only. A True value indicates that the matrix is a scalar. False means the matrix may not be scalar.	False (unknown)
integer	Flag whose value applies to matrix types only. A True value indicates that the matrix contains only integer values. False means the matrix may contain non-integer values.	False (unknown)
sizeKnown	Flag whose value applies to matrix types only. A True value indicates the size of the matrix is known. False means the size is not known.	False (unknown)
size	Applies to matrix types only. A vector of integers storing the dimensions of the matrix. This is only defined if the sizeKnown flag is set to True.	Undefined
handle	Applies to function handles types only. Stores a pointer to the function object the handle points to. This value can be null if the specific function is not known at inference time.	null (unknown)
cellTypes	Applies to cell array types only. Set of type objects representing the possible types the cell array stores.	Empty set (unknown)

The type objects are more than mere identifiers for language types; they also store various flags and attributes giving additional information about the possible values a variable may hold. Table 5.1 describes the fields stored in type objects. These fields cannot hold arbitrary

values. For example, if the `scalar` flag is set to True, then the `sizeKnown` flag must also be True. However, the `2D` flag does not necessarily indicate that the matrix size is known.

For each statement in a program, our analysis will produce a mapping of symbols to sets of type objects representing the type that each variable in the current function may hold before the statement is executed. Formally, if O is the set of all possible type objects and S is the set of all symbols, then our analysis operates in the domain of subsets of M, where M is the set of all pairs of symbols and subsets of O (mappings of symbols to type sets):

$$M = \{(s,t) | s \in S, t \in P(O)\} \tag{5.1}$$

5.2.2 Merge Operator

A merge operator is required to implement inference rules for control flow statements. This is because when multiple control paths join at a given point in a program, our analysis needs to merge the mappings of symbols to type sets for each of these control flow paths into one single mapping. In our analysis, the merging of two type mappings is accomplished by performing, for each symbol, the joining of the type sets for each type mapping:

$$merge(M_1 \subseteq M, M_2 \subseteq M) = \{(s,t) | (s,t_1) \in M_1, (s,t_2) \in M_2, t = join(t_1,t_2)\} \tag{5.2}$$

The joining of type sets is accomplished by using set union as a merge operator and then applying a filter operator to the result. If one of the input values to the merge operator for type sets is $\top$, then the result will be $\top$ as well, because this value signifies that a variable's type cannot be determined. However, if one of the merge values is $\bot$, then the result will be the other value, because $\bot$ signifies that type information has not yet been determined. In the case where neither values are $\top$ or $\bot$, then the result is simply the union of both type

sets:

$$join(t_1 \subseteq P(O), t_2 \subseteq P(O)) = filter(t_1 \cup t_2) \tag{5.3}$$

The filter operator can be defined in terms of pseudocode (see listing 5.1). Its purpose is to merge all type objects in a type set having the same McVM type into one. It does so in a pessimistic way, that is, if any of the type objects to be merged has an unknown value for one of its flags, the merged type object will have the unknown value. Properties are only kept if all objects hold them as known. For example, if we are filtering a type set containing multiple `double` matrix type objects, the resulting type object will have the `integer` flag set to true only if all of the original matrix type objects did.

```
function TypeSet filter(TypeSet S)
  repeat until S is unchanged
    for each pair (s1, s2) in S
      if s1.type equals s2.type

        # Create a new type object with the same type
        s = new TypeObject();
        s.type = s1.type;

        # Merge the type object flags pessimistically
        # Using the boolean AND operator
        s.2D = s1.2D AND s2.2D;
        s.scalar = s1.scalar AND s2.scalar;
        s.integer = s1.integer AND s2.integer;
        s.sizeKnown = s1.sizeKnown AND s2.sizeKnown;

        # Set the function handle only if both handles match
        if s1.handle equals s2.handle
          s.handle = s1.handle
        else
          s.handle = null;
        end

        # Merge the cell array type sets
```

```
25        s.cellTypes = merge(s1.cellTypes, s2.cellTypes);
26
27        # Update the type set S
28        remove s1 from S;
29        remove s2 from S;
30        add s to S;
31
32      end
33    end
34
35    # Return the filtered type set
36    return S;
37  end
```

Listing 5.1 Pseudocode for the type set filter operator

5.2.3 Inference Rules

Our type inference analysis follows inference rules to determine the mapping of possible variable types after a given statement based on the possible types before that same statement. Each kind of statement has an associated type inference rule that takes the mapping of possible input types as input and returns the mapping of possible output types as output. Expression statements, such as `disp(3);` use the identity type mapping, that is, the output types they produce are the same as the input types.

The statements that are at the core of our type inference analysis are assignment statements. They are the only kind of statement that can define a variable, and thus, change its type. In the case of an assignment statement of the form `v = op(a, b);`, where `op` is an element of the set `R` of all possible binary operators, we have that the type of `v` is redefined as the set of possible output types of the operator being applied to the possible types of `a` and `b`, according to its own type rule:

$$typeRule_{v=op(a,b)}\left(M_{in} \subseteq M\right) = \left\{(s,t) \in M_{in} \middle| s \neq v\right\} \cup typeRule_{op(v,a,b)}\left(M_{in}\right) \tag{5.4}$$

$$typeRule_{op(v,a,b)}\left(M_{in} \subseteq M\right) = \left\{ (v,t) | t = outtype_{op}\left(\{(a,t) \in M_{in}\}, \{(b,t) \in M_{in}\}\right)\right\} \tag{5.5}$$

```
if cond
  trueStmt;
else
  falseStmt;
end
```

Listing 5.2 Type inference and branching

In the case of **if** statements, the type inference process is handled differently. The "true" and "false" branches of the statement are both treated as compound statements, as if all statements on either branch were one statement (see listing 5.2). The output type mappings are determined separately for both branches and then merged together into one mapping of the possible types at the output of the **if** statement itself:

$$typeRule_{if}\left(M_{in} \subseteq M\right) = merge\left(typeRule_{trueStmt}\left(M_{in}\right), typeRule_{falseStmt}\left(M_{in}\right)\right) \tag{5.6}$$

```
while cond
  loopStmt;
end
```

Listing 5.3 Type inference and loops

Handling of loop statements is slightly more complex. Because types at the input of the loop depend on types at the output, a fixed point must be iteratively computed. For the purpose of our type inference analysis, all loop statements are treated as **while** loops. As is the case for **if** statements, statements in the loop body are treated as one single compound statement (see listing 5.3). The type inference rule for **while** loops can be expressed in the form of pseucode, as in listing 5.4.

```
function TypeMapping typeRuleWhile(Statement loopStmt, TypeMapping
    inTypes)

 # Create a type mapping for the types at the continue points of
 # the loop. We initialize this mapping so all variable types are
 # initially bottom (uninferred)
 TypeMapping contTypes = {(s,t) | s in S, t = bottom};

 # Create a type mapping for the types at the "break" or exit
 # points of the loop
 TypeMapping exitTypes;

 # Repeat until a fixed point is reached for the output types
 repeat until (contTypes is unchanged) AND (exitTypes is unchanged)

  # Compute the types at the input of the loop statement
  loopInTypes = merge(inTypes, contTypes);

  # Add new lists to the top of the break and continue list stacks.
  # These are global variables which will be filled in as the
  # type inference analyzes the loop body statement.
  breakStack.push(new List());
  continueStack.push(new List());

  # Compute the types after the loop body by
  # applying the type rule for the statement
  bodyOutTypes = typeRule(loopStmt, loopInTypes);

  # Pop the break and continue lists from their stacks
  breakList = breakStack.pop();
  continueList = continueStack.pop();

  # Compute the type mapping for all continue points.
  # The types at the exit of the loop body are included
  contTypes = bodyOutTypes;
  for mapping in continueList
   contTypes = merge(contTypes, mapping);
```

```
37      end
38
39      # Compute the type mapping for all break points.
40      # The input types are included in case the loop
41      # executes zero times
42      exitTypes = inTypes;
43      for mapping in breakList
44        exitTypes = merge(exitTypes, mapping);
45      end
46
47    end
48
49    # Return the type mapping at the exit of the loop
50    return exitTypes;
51  end
```

Listing 5.4 Type inference rule for while loop statements

The type inference process for loops is complicated by the fact that there could be `break` or `continue` statements in the loop. We handle this by maintaining global lists of the type mappings associated with these statements for each iteration. These are then be included into the type mapping merging process of the fixed point computation. The `continue` type mappings correspond to the back-edge (control flow edge) going from the point after the loop body to the loop entry. The `break` type mappings must be taken into account to properly compute the possible types at the loop exit.

5.2.4 Handling Recursion

Our current type inference analysis allows type information to flow across call sites, but does not currently analyze recursive call chains. Its current behavior is to terminate the analysis when a call to a function which is currently being analyzed is encountered, at which point it returns no type information about the recursive call. This termination criterion ensures that there will be no infinite recursion in the type analysis itself.

We have found this solution to be acceptable for most of our benchmark programs, as most

of the heavy computations are done inside of loops, and not through recursion. For other dynamic languages, it may be desirable to propagate type information across recursive call sites. This will, however, introduce additional complications [FH88] as it becomes necessary to extract the non-recursive execution paths and provide type information about those. This is because the type analysis of a recursive function depends on the results of that same analysis. Thus, the analysis must be able to generate output for a non-recursive base case.

5.2.5 Inference Process

In terms of abstract interpretation, we wish to compute, for a given function, the least fixed point of the mapping of program statements and variables to sets of possible types before that given program point. The type inference process for a function begins with the type sets for the input parameters of the function being given. Because of the MATLAB semantics, the possible types of all other variables are initialized to $\top$. This is because undeclared variables could be globals, and thus, could potentially hold any type.

The body of the function is then analyzed. The function body itself is a compound statement. When inferring the types in a compound statements, the statements it contains are traversed in order, and the output type of each statement is stored in a global variable we will call `stmtTypeMapping`. This is a map (e.g.: hash map) of the type mapping at the output of each statement. Pseudocode is given for this in listing 5.5. When inferring types for compound statements, we are careful to store the type mappings associated with **`break`** and **`continue`** statements. These will be used when performing type inference on loops.

```
1 function TypeMapping typeRuleCompound(Statement list L, TypeMapping
     inTypes)
2
3   # For each statement in the list, in sequence
4   for each statement stmt in L
5
6     # Store an association of the statement
7     # with the type mapping that holds true before
8     # it into a global hash map. This is what our
```

```
9     # analysis produces as output.
10    add pair(stmt, inTypes) to global stmtTypeMapping;
11
12    # Compute the types after the statement
13    # these will be the input to the next statement
14    inTypes = typeRule(stmt, inTypes);
15
16    # If the statement is a break statement, we
17    # store the type mapping after the statement
18    # into the current global break list
19    if stmt is of type breakStmt
20      add inTypes to breakStack.top()
21    end
22
23    # If the statement is a continue statement, we
24    # store the type mapping after the statement
25    # into the current global continue list
26    if stmt is of type continueStmt
27      add inTypes to continueStack.top()
28    end
29
30  end
31
32  # Return the mapping at the output of the compound
33  return inTypes;
34 end
```

Listing 5.5 Type inference and compound statements

5.3 An Example

This section contains a step-by-step example to illustrate the workings of our type inference analysis. Our example begins with the program shown in listing 5.6, which comprises two functions: `caller` and `callee`. The `caller` function branches on its input argument, and calls the `callee` function with two different arguments. The `callee` function iteratively

computes a sum based on a counter variable which is divided by two at each iteration.

```
function r = caller(val)
  if val > 5
    r = callee(val);
  else
    val2 = 2 * val;
    r = callee(val2);
  end
end

function v = callee(start)
  v = 0;
  c = start;
  while c > 0.25
    v = v + c;
    c = c / 2;
  end
end
```

Listing 5.6 Type inference analysis example: step 0

For the sake of the example, imagine that a user invokes the `caller` function with a 1x2 matrix of integers as its input argument (i.e.: `caller([5 6]);` is typed at the interactive prompt). Our type inference analysis is then invoked on the caller function for this specific argument type. It will then begin with the assumption that `val` has the 1x2 integer matrix type, as illustrated in listing 5.7. The analysis will then traverse statements on both the "true" and "false" branches of the **`if`** statement.

```
function r = caller(val <1x2 int>)
  if val > 5
    r = callee(val);
  else
    val2 = 2 * val;
    r = callee(val2);
  end
end
```

Listing 5.7 Type inference analysis example: step 1

On the “true” branch of the `if` statement, our analysis will conclude that the `callee` function is called with a 1x2 integer matrix argument. Thus, the analysis will recursively analyze the `callee` function for this argument type. The variable `v` will typed as a scalar integer value directly, since it is assigned constant, and the variable `c` will be assigned the same type as the `start` variable (see listing 5.8).

```
function r = caller(val <1x2 int>)
  if (val <1x2 int>) > 5
    r = callee(val <1x2 int>);
  else
    val2 = 2 * val;
    r = callee(val2);
  end
end

function v = callee(start <1x2 int>)
  v <scalar int> = 0;
  c <1x2 int> = (start <1x2 int>);
  while c > 0.25
    v = v + c;
    c = c / 2;
  end
end
```

Listing 5.8 Type inference analysis example: step 2

Analyzing the body of the `while` loop a first time, our analysis will conclude that inside the loop, `v` becomes a 1x2 integer matrix, since that is the resulting type of the addition of a scalar integer with a 1x2 integer matrix. However, it will conclude that `c` becomes a 1x2 matrix of real values, because dividing integer values by integers is not guaranteed to result in integer values (see listing 5.9).

```
function v = callee(start <1x2 int>)
  v <scalar int> = 0;
```

```
  c <1x2 int> = (start <1x2 int>);
  while (c <1x2 int>) > 0.25
    v <1x2 int> = (v <scalar int) + (c <1x2 int>);
    c <1x2 real> = (c <1x2 int>) / 2;
  end
end
```

Listing 5.9 Type inference analysis example: step 3

The new types of c and v at the end of the loop body will be merged with the types at the loop entry. The analysis will conclude that c is of type 1x2 real at the loop entry, while v is of type integer with unknown size, because it was scalar integer in the first iteration, and will be 1x2 integer in the second, and thus its size cannot be guaranteed across all iterations. However, inside the loop body, because adding an integer matrix of unknown size to a real matrix of size 1x2 results in a matrix of real numbers with size, the type of v will become a real number matrix of unknown size. A third iteration will show that the types inside the loop do not change at this point, and thus that a fixed point has been reached (see listing 5.10).

```
function v = callee(start <1x2 int>)
  v <scalar int> = 0;
  c <1x2 int> = (start <1x2 int>);
  while (c <1x2 real>) > 0.25
    v <real> = (v <real>) + (c <1x2 real>);
    c <1x2 real> = (c <1x2 real>) / 2;
  end
end
```

Listing 5.10 Type inference analysis example: step 4

At this point, the analysis of the `callee` function is completed. The analysis will conclude that for the 1x2 integer argument type, the return type of the function is real. This will be reflected in the `caller` function, in which the value `r` on the “true” branch will take the real type (see listing 5.11).

```
function r = caller(val <1x2 int>)
  if (val <1x2 int>) > 5
```

```
    r <real> = callee(val <1x2 int>);
  else
    val2 = 2 * val;
    r = callee(val2);
  end
end

function v <real> = callee(start <1x2 int>)
  v <scalar int> = 0;
  c <1x2 int> = (start <1x2 int>);
  while (c <1x2 real>) > 0.25
    v <real> = (v <real>) + (c <1x2 real>);
    c <1x2 real> = (c <1x2 real>) / 2;
  end
end
```

Listing 5.11 Type inference analysis example: step 5

Since the analysis of the "true" branch is completed, the analysis of the "false" branch will proceed. Trivially, `val2` will be assigned the 1x2 integer type, since multiplying a 1x2 integer matrix by a scalar integer preserves its type. Since the `callee` function is again being called with the 1x2 integer type, the results of the type analysis on this function for that type (which have been cached) will be reused. Thus, our analysis will conclude that `r` also has the real type along the "false" branch (see listing 5.12).

```
function r = caller(val <1x2 int>)
  if (val <1x2 int>) > 5
    r <real> = callee(val <1x2 int>);
  else
    val2 <1x2 int> = 2 * (val <1x2 int>);
    r <real> = callee(val2 <1x2 int>);
  end
end
```

Listing 5.12 Type inference analysis example: step 6

Since the `r` variable has the same real type along both branches of the **`if`** statement, our analysis will conclude that its type at the output of the `caller` function is also real (see listing 5.13). At this point, the type inference analysis is completed for the `caller` function with a 1x2 integer argument. The results of the analysis of both the `caller` and `callee` function for their specific argument types will be cached for later use.

```
function r <real> = caller(val <1x2 int>)
  if (val <1x2 int>) > 5
    r <real> = callee(val <1x2 int>);
  else
    val2 <1x2 int> = 2 * (val <1x2 int>);
    r <real> = callee(val2 <1x2 int>);
  end
end

function v <real> = callee(start <1x2 int>)
  v <scalar int> = 0;
  c <1x2 int> = (start <1x2 int>);
  while (c <1x2 real>) > 0.25
    v <real> = (v <real>) + (c <1x2 real>);
    c <1x2 real> = (c <1x2 real>) / 2;
  end
end
```

Listing 5.13 Type inference analysis example: step 7

5.4 Validation Strategy

Our JIT compiler relies on the type inference analysis to make optimization decisions. This often means eliminating run-time checks because type information has been statically inferred. For example, if we know that a variable will always be a floating-point matrix at a certain point in the program, we no longer need to check what type it actually is before operating on this variable at run-time. This poses one important design problem, however. Should the type analysis produce incorrect result, the JIT compiler could produce

erroneous code based on incorrect assumptions. This could in turn result in bugs that are rather difficult to track.

To avoid such situations, we have devised a validation strategy which essentially entails running a series of benchmarks (see section 6.1 for a list) through the interpreter while checking that the predicted types match the actual types at every point (before and after every 3-address form statement) in the execution of the said benchmarks. Should an error occur, the program point where this happens is reported, along with the erroneous type information, and the actual types that occur at run-time. To reduce the overhead of this validation process, the number of times a given statement will be validated during the execution of a function is limited to 128.

This validation strategy is by no means perfect, because our benchmarks do not expose all the possible programs that could arise. Thus, it is possible that some errors could slip through. In practice, however, we have found this strategy to be quite effective at catching bugs in our type analysis. We also believe that our benchmarks are fairly extensive as they make use of every language feature our VM supports and exhibit a fair level of code complexity.

An alternative (or complementary) validation strategy would have been to use unit testing to test our system. The difficulty with this, however, is that due to the complexity of our type inference system, there would be hundreds of cases to test in order to properly validate our type analysis. These unit tests would also need to generate code segments and validate the results of the analysis over these. Our empirical approach is simply using actual program code to validate the analysis instead of code generated by synthetic tests.

Chapter 6

Performance Study

In this chapter, we evaluate the performance of our JIT compiler as well as that of the various optimization strategies it employs. We begin with a description of our benchmarking strategy, including the benchmark programs, the platform on which benchmarking was performed, the versions of software used in performance measurements and the timing strategy used. This is followed by a comparison of objective performance numbers for our JIT compiler as well as competing implementations such as Mathworks MATLAB and GNU Octave.

The performance impact of our JIT compiler with various optimizations disabled is also examined so as to assess their respective impact. We then examine the factors that can explain performance differences between the various configurations of our JIT as well as competing implementations. We finally conclude with an examination of the performance of our type inference system and its impact on the global performance of our JIT compiler.

6.1 Benchmarking Strategy

In order to assess the performance of our virtual machine, we have chosen to compare the actual performance of McVM (in terms of running-time of benchmark programs) to that obtained by competing solutions such as Mathworks MATLAB, GNU Octave (the GNU

MATLAB environment) and McFor (a MATLAB to Fortran translator built by Jun Li, a member of the McLab team). The Octave and MATLAB performance numbers (shown in section 6.2 are intended to give us some idea of how well our current solution performs against competing implementations. The McFor (FORTRAN code) numbers are provided as a kind of "lower bound". FORTRAN compilers are known to perform very well for numerical computations, and thus, these numbers tell us how what kind of performance levels we could potentially hope to reach with future VM implementations.

We have also measured our JIT compilation times in order to establish whether they fall within reasonable norms. In order to establish how successful our attempts at code optimization have been, we have compared the performance of McVM while running in interpreted mode, as well as with the JIT compiler enabled, and with specific JIT compiler optimizations disabled. Furthermore, we show some profiling numbers intended to explain where specific performance bottlenecks occur, as well as type inference profiling data to help explain in which cases our type inference analysis does best and worse (see section 6.3).

We have performed our tests on a total of 20 benchmark programs, which are briefly described in table 6.1. Several of these are currently unsupported by the McFor FORTRAN translator as it lacks support for cell arrays, closures and function handles at this time. Table 6.2 provides some characteristic numbers for the 20 benchmarks supported by McVM, namely, the number of functions in each program, the total number of statements (in 3-address form), the maximum loop nesting depth in the entire program, and the total number of call sites found.

All of our benchmarking metrics were gathered on a system equipped with an Intel Core 2 Quad Q6600 processor (quad core, 2.4GHz) and 4GB of dual channel DDR2 RAM, running Ubuntu 9.04 (linux kernel 2.6.28). We have gathered our MATLAB performance numbers using MATLAB R2009a, and our GNU Octave numbers on Octave version 3.0.1. The FORTRAN code produced by McFor was compiled using the GNU FORTRAN compiler version 4.3.3. Because there is some variance when timing benchmarks (on the order of 10 to 20%), all benchmark timing measurements were made by running the benchmark programs a total of 30 times and averaging over all the values.

Table 6.1 Description and origin of our benchmark programs

Benchmark	Description	Origin
adpt	Adaptive quadrature by Simpson's Rule	FALCON Project
beul	Solves the heat equation using the backward Euler method	Anton Dubrau (McLab team)
capr	Computes the transmission line capacitance of a coaxial pair	Chalmers University of Technology
clos	Computes the transitive closure of a directed graph	OTTER Project
crni	Crank-Nicholson heat equation solver	FALCON Project
dich	Dirichlet solution to Laplace's equation	FALCON Project
diff	Young's two-slit diffraction experiment	MathWorks' Central File Exchange
edit	Computes the edit distance of two strings	MathWorks' Central File Exchange
fdtd	Finite Difference Time Domain (FDTD) technique	Chalmers University of Technology
fft	Computes the discrete fourier transform for complex data	Jun Li (McLab team)
fiff	Finite difference solution to the Wave equation	FALCON Project
mbrt	Generates the mandelbrot set	Anton Dubrau (McLab team)
nb1d	One-dimensional n-body simulation	OTTER Project
nb3d	Three-dimensional n-body simulation	Modified nb1d
nfrc	Generates a newton fractal	Anton Dubrau (McLab team)
nnet	Neural network learning AND, OR, XOR functions	Anton Dubrau (McLab team)
play	Recursive minimax search	Anton Dubrau (McLab team)
schr	Solves 2-D Schroedinger equation	Anton Dubrau (McLab team)
sdku	Sudoku solver	Andrew Casey (McLab team)
svd	Computes the singular value decomposition of a large matrix	Anton Dubrau (McLab team)

Table 6.2 Characteristics of our benchmark programs

Benchmark	Num. functions	Num. statements	Max. loop depth	Num. call sites
adpt	2	196	2	6
beul	10	511	1	38
capr	5	214	2	10
clos	2	58	2	3
crni	3	142	2	7
dich	2	144	3	7
diff	2	253	3	6
edit	2	130	2	6
fdtd	2	157	1	3
fft	2	159	3	8
fiff	2	120	2	4
mbrt	3	78	2	11
nb1d	3	194	2	11
nb3d	3	164	2	12
nfrc	5	151	2	11
nnet	4	186	3	16
play	6	364	2	29
schr	8	203	1	32
sdku	9	363	2	49
svd	11	308	3	42

6.2 Objective Performance

Table 6.3 shows a comparison of benchmark running times when running through McVM with the JIT compiler enabled, Mathworks MATLAB, McVM with the JIT compiler disabled, GNU Octave, and McFor (compiled FORTRAN code), while table 6.4 shows the same running times normalized relative to that of the McVM JIT (values greater than one representing running times slower than the McVM JIT). As we can see, McVM with JIT performs better than MATLAB in 6 out of 20 benchmarks, sometimes by a fair margin. In the cases where it does worse than MATLAB, the running times are still relatively close in most cases, except for the `crni` benchmark which performs rather poorly (we will examine why in section 6.3).

Table 6.3 Comparison of benchmark running times across multiple environments

Benchmark	McVM JIT (s)	MATLAB (s)	McVM no JIT (s)	Octave (s)	McFor (s)
adpt	1.26	0.29	1.30	4.57	0.19
beul	0.35	0.31	0.18	0.78	N/A
capr	0.37	0.81	181.30	544.97	0.12
clos	0.70	0.08	1.43	1.71	0.72
crni	137.06	0.71	190.24	577.45	0.33
dich	0.30	0.47	122.15	411.43	0.17
diff	3.23	0.53	4.43	11.29	0.06
edit	8.26	1.62	11.62	52.21	0.02
fdtd	2.45	0.34	0.96	16.17	0.03
fft	1.36	1.65	287.00	958.79	0.67
fiff	0.73	0.72	163.75	485.58	0.13
mbrt	3.95	0.47	10.47	28.84	0.09
nb1d	0.35	1.14	0.40	5.20	0.04
nb3d	0.32	0.11	0.17	2.23	0.05
nfrc	1.88	0.48	2.74	7.73	N/A
nnet	0.79	0.65	0.79	2.79	N/A
play	0.53	1.16	0.60	3.97	N/A
schr	1.24	0.87	0.89	0.97	N/A
sdku	0.63	1.87	3.17	21.01	N/A
svd	2.03	0.64	1.37	2.95	N/A

GNU Octave, possessing no JIT compiler, does rather poorly in general. It trails far behind MATLAB and outperforms McVM with JIT on a single benchmark. Interestingly, McVM in interpreted mode, although it performs much worse than the JIT on several benchmarks, actually performs better on some (this will also be discussed further). The McFor running times are generally far ahead of MATLAB and McVM, except for the `clos` benchmark, suggesting that MATLAB and McVM are still far from the "optimal" performance level.

In table 6.5, we show profiling numbers gathered using the McVM interpreter. These numbers are counts of how many matrices were created, how many matrix read operations were performed, the number of matrix multiplication operations executed, the total count of environment lookups, and the total number of function calls executed. These give us some idea about the behavior of our benchmark programs. Interestingly, we can clearly

Table 6.4 Benchmark running times relative to the McVM JIT performance

Benchmark	MATLAB	McVM no JIT	Octave	McFor
adpt	0.23	1.04	3.64	0.15
beul	0.90	0.53	2.24	N/A
capr	2.19	487.55	1465.51	0.32
clos	0.12	2.04	2.45	1.03
crni	0.01	1.39	4.21	0.00
dich	1.59	413.73	1393.52	0.57
diff	0.16	1.37	3.49	0.02
edit	0.20	1.41	6.32	0.00
fdtd	0.14	0.39	6.60	0.01
fft	1.22	211.25	705.73	0.49
fiff	0.99	225.24	667.92	0.17
mbrt	0.12	2.65	7.30	0.02
nb1d	3.30	1.17	15.06	0.10
nb3d	0.35	0.54	7.01	0.14
nfrc	0.26	1.45	4.11	N/A
nnet	0.83	1.00	3.53	N/A
play	2.18	1.13	7.44	N/A
schr	0.70	0.72	0.78	N/A
sdku	2.97	5.03	33.39	N/A
svd	0.32	0.68	1.45	N/A

see that the benchmarks with the highest number of matrices created are the slowest to run through the McVM and Octave interpreters. This is because allocating new matrices very often is both expensive and cache unfriendly.

Table 6.6 shows relative profiling counts of the number of matrices created, the number of slice reads, and the number of environment lookups when the JIT is enabled. These are actually ratios (percentages) of the values obtained in interpreted mode. This table is meant to show that the JIT compiler is able to reduce the occurrence of all these expensive operations in almost all cases, and never increases their occurrence. In particular, we see that the `fft` benchmark has its number of matrix slice reads reduced to 0.00%. This benchmark also happens to run over 200 times faster with the JIT compiler enabled.

We examine the relative performance of McVM with specific JIT optimizations disabled

Table 6.5 Interpreter profiling counts for our benchmark programs

Benchmark	Matrices created	Matrix slice reads	Matrix mult ops	Env. lookups	Function calls
adpt	103 K	9 K	1	72 K	5 K
beul	8 K	23	2801	8 K	9995
capr	26 M	7 M	1	31 M	3 K
clos	143 K	1	9	225 K	13
crni	29 M	7 M	1	31 M	1 K
dich	20 M	4 M	1	22 M	697 K
diff	791 K	1	1	791 K	154 K
edit	2 M	375 K	1	2 M	75 K
fdtd	4 K	6007	1	2 K	25
fft	43 M	9 M	1	48 M	56
fiff	29 M	5 M	1	28 M	6 K
mbrt	2 M	1	1	2 M	225 K
nb1d	42 K	4 K	1	68 K	4 K
nb3d	3 K	5248	367	3 K	1117
nfrc	434 K	1 K	1	219 K	46 K
nnet	119 K	18 K	4 K	79 K	1 K
play	91 K	6 K	6767	65 K	10 K
schr	1133	96	41	732	129
sdku	335 K	110 K	1	454 K	27 K
svd	74 K	9 K	4 K	62 K	5 K

in table 6.7. Columns show the relative performance (running time) of benchmarks as compared with the JIT compiler with all optimizations enabled. A number greater than one signifies a slowdown. Clearly, binary operation and array access optimizations have a tremendous optimization potential as they speed up several benchmarks by two orders of magnitude. Optimized library functions is able to speed up one benchmark by an order of magnitude, and shows promise. We note, however, that while these three optimizations do cause slowdowns in some cases, those are minor, and likely artifacts (generating code differently sometimes yields poorer cache performance, etc.).

The direct call mechanism involves directly compiling calls to program functions rather than performing them through the JIT. This is beneficial to benchmarks that perform many function calls (e.g.: `dich`), however, it can yield lower performance in cases where the

Table 6.6 Relative JIT profiling counts for our benchmark programs

Benchmark	Matrices created (%)	Matrix slice reads (%)	Env. lookups (%)
adpt	24.84	16.79	39.21
beul	85.47	47.83	114.19
capr	0.00	0.00	0.00
clos	0.00	100.00	0.00
crni	66.65	69.23	55.16
dich	0.00	0.00	0.00
diff	68.28	100.00	2.45
edit	64.99	39.98	81.56
fdtd	88.07	90.01	90.46
fft	0.00	0.00	0.00
fiff	0.01	0.00	0.00
mbrt	33.20	100.00	0.00
nb1d	75.47	0.01	14.66
nb3d	93.06	97.71	74.43
nfrc	42.53	100.00	19.80
nnet	85.82	100.00	81.94
play	72.24	99.83	45.84
schr	65.14	54.17	83.61
sdku	18.21	13.96	11.32
svd	84.79	100.00	60.00

types of input parameters to a function are unknown. A version of the function then gets compiled with insufficient type information, whereas the interpreter can extract exact type information on the fly when a call is performed.

The last column of table 6.7 shows the relative performance of interpreted mode as compared to the JIT compiler with all optimizations enabled. As we can see, the JIT speeds up most benchmarks, sometimes by very large factors. However, it also causes some slowdowns in some cases. These cases often correspond to benchmarks where the JIT was not able to efficiently optimize the code. In the case of `nb3d`, for example, we can see in table 6.6 that the number of matrix slice reads was reduced by less than 3%. In such cases, the JIT can actually add additional type conversion overhead by constantly requiring local variables to be wrapped into objects for interpreter fallback.

Table 6.7 Relative JIT performance with specific optimizations disabled

Benchmark	No binary opts	No array opts	No direct calls	No library opts	No JIT
adpt	1.47	1.17	1.00	1.07	1.04
beul	1.01	0.99	0.99	0.98	0.53
capr	572.70	416.09	1.74	1.04	487.55
clos	3.42	1.00	1.00	1.01	2.04
crni	1.69	1.31	0.78	1.00	1.39
dich	448.91	285.35	1.00	30.06	413.73
diff	2.14	1.02	1.02	1.01	1.37
edit	1.87	1.45	0.61	1.00	1.41
fdtd	0.95	1.06	0.99	1.01	0.39
fft	194.80	171.26	1.00	1.02	211.25
fiff	233.63	165.63	1.02	1.05	225.24
mbrt	3.51	0.98	1.01	1.00	2.65
nb1d	1.06	1.28	1.02	1.01	1.17
nb3d	0.73	0.98	1.06	1.02	0.54
nfrc	1.24	0.98	1.68	1.00	1.45
nnet	1.17	0.99	1.03	1.00	1.00
play	1.15	0.99	1.07	1.00	1.13
schr	0.84	0.97	1.26	0.96	0.72
sdku	1.10	1.64	1.16	0.98	5.03
svd	0.79	0.92	1.06	0.95	0.68

We examine the total JIT compilation times for our benchmarks in table 6.8. This table also shows the number of functions present in each benchmark, how many specialized function versions were compiled in total for these benchmarks and how much time was spent by the compiler performing analyses (e.g.: type inference). As we would expect, in most cases, there are no more specialized versions than the number of functions, because the functions are always called with the same argument types, and thus no more than one version is compiled for each function. In cases where there are more, there are never more than twice as many versions as functions. This gives some credibility to our approach, at least when applied to environments such as MATLAB: there is no explosion of the number of specialized function versions in practice.

As we can see, most of the compilation time is spent performing analyses on the functions

Table 6.8 Compilation times of our benchmark programs

Benchmark	Num. functions	Num. versions	Compilation time (s)	Analysis time (s)
adpt	2	2	0.89	0.82
beul	9	16	1.21	0.91
capr	5	5	0.50	0.43
clos	2	2	0.18	0.15
crni	3	3	0.33	0.27
dich	2	2	0.33	0.28
diff	2	2	1.26	1.18
edit	2	2	0.26	0.21
fdtd	2	2	0.47	0.36
fft	2	2	0.59	0.55
fiff	2	2	0.23	0.19
mbrt	3	3	0.16	0.12
nb1d	3	3	0.47	0.38
nb3d	3	3	0.51	0.40
nfrc	5	5	0.24	0.16
nnet	4	4	0.36	0.29
play	6	10	0.58	0.42
schr	8	9	0.53	0.42
sdku	9	11	1.07	0.84
svd	11	15	0.78	0.59

to be compiled, as opposed to code generation. This suggests that this is an area where the performance of our compiler could be improved. The compilation times we have obtained are in some cases relatively long. Longer than the running times of benchmarks themselves in several instances. However, real scientific programs can run for hours. Seeing how our JIT compiler has yielded speedups of three orders of magnitude over GNU Octave and our interpreter in some cases, we believe that this compilation overhead will be easily amortized, as the JIT compiler could be saving hours of CPU time in the end.

We also note that little is known about the compilation strategy used by Mathworks MATLAB, since the implementation is not open source. We do not know what compilation strategy is used or when MATLAB parses and compiles benchmarks, and thus, cannot measure the MATLAB compilation times. For all we know, they may actually be comparable to

those of McVM. Note that the compilation times obtained were included in the timing of our benchmarks as the benchmarks were compiled during the first timing iteration. Thus, assuming MATLAB compiles functions as they are called during the first timing iteration (or found to be callees of functions being compiled), as with McVM, we are comparing the McVM and MATLAB running times on fair grounds.

6.3 Type Inference Efficiency

Table 6.9 Performance of the type inference system

Benchmark	Top sets (%)	Unary sets (%)	Scalars known (%)	Size known (%)	JIT speedup (%)
adpt	4.2	95.8	100.0	90.0	3.6
beul	55.2	44.8	71.3	29.6	-89.4
capr	0.0	100.0	100.0	82.8	99.8
clos	0.0	100.0	100.0	99.9	51.0
crni	19.1	71.4	68.7	54.8	28.0
dich	2.1	97.9	100.0	85.1	99.8
diff	14.3	82.1	66.7	66.7	27.0
edit	5.1	94.9	96.8	81.5	29.0
fdtd	0.1	99.9	100.0	49.8	-154.1
fft	0.0	100.0	100.0	80.5	99.5
fiff	0.0	100.0	100.0	86.1	99.6
mbrt	9.1	90.9	100.0	100.0	62.2
nb1d	5.8	94.2	88.4	34.7	14.7
nb3d	3.4	96.6	100.0	18.1	-86.7
nfrc	16.4	82.7	100.0	98.9	31.2
nnet	52.2	47.8	98.7	55.5	0.3
play	23.3	66.4	77.4	52.0	11.5
schr	31.8	54.9	99.5	41.5	-39.2
sdku	14.8	85.2	83.8	49.4	80.1
svd	16.5	73.7	94.0	59.7	-47.7

In this section we examine the efficiency of our type inference strategy and its impact on the performance of our JIT compiler. Table 6.9 shows profiling information relating to our type

inference system. The first four columns are percentages values measured by sampling the type sets of variables used by each statement at each execution of every program statement. That is, the profiling was done so as to take the number of times a statement is executed into consideration. The last column is a measurement of the percentage of speed improvement the JIT compiler achieves over interpreted code (negative values represent slowdowns).

The first column of the table tells us the percentage of type sets that are $\top$ (top, unknown types). The second column is the percentage of type sets containing only one type (the specific type of the variable is known). The third column shows the percentage of times where variables holding scalar values were known ahead of time to be scalar by the type inference system. The fourth column is the percentage of times where the size of matrix variables was known by the type inference system.

The higher the proportion of $\top$ type sets, the less type information our system knows. This means less information on which to base optimizations, and generally poorer performance. However, the knowledge of which variables are scalars is even more critical, as it lets the JIT compiler know which variables can be stored on the stack. As we can see, this matches our results: benchmarks with speedups of over 99% all have 100.0% of scalar variables known.

As discussed in section 6.2, the `crni` benchmark, while faster in the JIT compiler than in our interpreter, performs much worse in McVM than MATLAB (using our JIT, it has the highest running time of all our benchmarks). The reason for this is that it has relatively poor type information. As can be seen in table 6.9, scalars are known in only 68.7% of cases. This is because this benchmark uses matrix "creation on assignment" to initialize its input data (see section 7.2). This results in several unknown types being propagated through the entire program. We examine ways to fix this weakness of our type inference system in section 8.2.3.

While our JIT compiler is able to speed up most benchmarks, sometimes by very significant margins, some still show slowdowns over interpreted performance. These do not necessarily have poor type information. The `nb3d` benchmark, for example, has 100.0% scalar variables known and 96.6% unary type sets. Most of these poorly optimized benchmarks

make heavy use of matrix slice read operations (operating on entire columns or rows of a matrix at a time) which we currently have no optimization support for. Hence, the JIT generates expensive interpreter fallback code requiring many type conversions, and performs poorer than the interpreter itself. We discuss potential ways to eliminate this performance issue in section 8.2.1.

Chapter 7

Language Design Issues

There are a number of factors that make the implementation of an optimizing virtual machine for the MATLAB programming language a challenging task. Some of these issues stem purely from the design of the language. These are examined in this chapter, which is largely a discussion of the "flaws" or oversights of the language, as well as an argument as to why dynamic language ought to be designed with more regard for performance.

We first discuss the difficulties associated with the lack of an official language specification for MATLAB. We then proceed to describe some of the MATLAB language features which make optimization more challenging as well as those which unnecessarily complicate the understanding of the language and make the design of a compliant implementation more difficult.

7.1 Defining the MATLAB Language

As noted by the authors of the `phc` PHP compiler [BdVG09], is it difficult to implement a virtual machine or static compiler for a language when there is no official specification for the said language. As is the case with PHP, Mathworks provides no such specification for the MATLAB language. The language behavior is purely defined by the behavior of the Mathworks MATLAB implementation (which can change with each release), and the

limited amount of documentation they provide for this product.

Unfortunately, this documentation does not go into the full details of the language semantics. Some key points are left out. Thus, in the implementation of McVM, much of our work consisted of trying different program variants in MATLAB, and comparing the behavior of the reference MATLAB implementation to that of McVM. This was necessary since the reference MATLAB implementation really is the only complete, publically available specification of the language. Still, to this point, we cannot guarantee that the subset of the MATLAB language McVM supports is fully compliant with the reference MATLAB implementation.

7.2 Optimization Barriers

Some language features make optimization of the MATLAB programming language more challenging. Most of these challenges are perhaps unavoidable. It will probably always be more difficult to optimize dynamically typed languages than their statically typed counterparts, for example. However, some of these challenges stem from design choices that seem arbitrary, and perhaps unnecessary or easily avoidable. Sometimes, the way the language is designed makes optimization particularly difficult without really giving the programmer any additional flexibility.

An instance of this which we have explained in some detail in section 3.6 is the `eval` construct, which can read or write any variable in a function's scope, potentially destroying almost all useful optimization information. McVM solved this by restricting `eval` to operate only on global variables. A better solution, however, may be to have an `eval` construct which can only operate on variables passed as input and returns an output directly instead of being allowed to assign values to outside variables. It is definitely possible to design a very flexible and useful `eval` construct without placing impractical constraints on the programmer or an optimizing compiler.

Another problematic construct also discussed in section 3.6 is the `cd` command, which can change the current global bindings in unpredictable ways. This problem truly stems

from language design more than anything else. We believe it would probably be more practical for MATLAB to have a "proper" module system as in Java or Python instead of basing lookups on the file system and relative paths. The `cd` command does not really grant programmers more flexibility, and is rather unintuitive (one may not expect all global bindings to change).

The `eval` construct is not the only case where a function can access the environment of a caller. Some MATLAB library functions, such as `feval` need to look into the caller's scope to work as they do in MATLAB. In the case of `feval`, it can be used to call a function based on its name. This is less problematic than the case of `eval`, but complicates the virtual machine design, and can also cause lost optimization opportunities. We also find that by allowing library functions to access variables without them being passed as arguments, MATLAB violates the idea that functions should act as well encapsulated units of functionality (the "black box" principle).

Another issue stems from the syntax of function calls and matrix indexing. In MATLAB, the expression `foo(a)` could either be a call to the function `foo` with argument `a`, or an indexing expression over the matrix `foo` with index `a`. Furthermore, function calls can be made without parentheses. The statement `a = b;` could be an assignment of `b` to `a`, or an assignment of the result of the function call to `b` with no arguments to `a`.

This makes it difficult to determine what variables are functions and what expressions are function calls, particularly when the variables are globally defined. This can, in some cases, result in lost optimization opportunities by making the types of variables more difficult to determine ahead of time. It also complicates the semantics of the language and makes them less obvious to the programmer. A simple fix would be to require parentheses for all function calls, and use different tokens for array indexing, such as square brackets (i.e.: `foo[a]`).

A last issue concerns the creation of matrices. As mentioned in section 6.3, in MATLAB, one can create a matrix by assigning at an index into a previously undeclared matrix. For example, if the variable `a` is currently not bound to anything, `a(5)= 1` will create a matrix with the value 1 at position 5. This is problematic for an optimizing compiler because `a`

may actually be a global variable bound to an existing matrix, in which case we do not know its type. We also find this to be problematic, because it could actually result in unexpected behaviors if one uses this method to attempt to declare a matrix when a global binding already exists.

7.3 Behavioral Inconsistencies

As previously stated, the MATLAB programming language has evolved "organically" over a number of years and does not have an official specification. The result of this unplanned expansion process is that in some areas, the language has unclear, unintuitive, imprecise or contradictory semantics. This generally complicates the implementation, because many "special cases" have to be taken care of to ensure MATLAB compatibility, but can also makes things more difficult for programmers by going against their expectations.

A simple example of behavior inconsistency is the way MATLAB handles comparisons between complex numbers. In MATLAB, the <, <=, > and >= operators only operate on the real part of a number, while the == and ~= (inequality) comparison operators operate on the whole number. The result is that, in MATLAB, the complex numbers `1 + 2i` and `1 + 3i` are not equal, yet neither one is less than or larger than the other. This is unintuitive and can easily violate the expectations of programmers.

More significant is the issue of implicit type conversions, as there is no true type hierarchy in MATLAB. Programmers familiar with C++ or Java may expect that adding an 8-bit integer matrix and a double precision floating-point matrix would yield a floating-point matrix (the higher precision type), but instead, the MATLAB result is an 8-bit integer matrix (the lower precision type). Hence, numerical precision is lost by default. Furthermore, one cannot apply arithmetic operators between integer matrices and single precision floating-point matrices. This behavior contradicts that of most common programming languages and is rather impractical.

MATLAB also contradicts itself in some ways. For example, in MATLAB, when one assigns outside of the bounds of a matrix, it will be automatically expanded to make the

assignment possible. However, if a scalar matrix A is expanded by executing `A(5)= 1;`, for example, the result will be a row vector of length 5. This is somewhat contradictory because in MATLAB, the first index is the row index, and matrices are stored in column major order. Thus, one would naturally expect the resulting vector to be a column vector.

A strange feature of MATLAB is the presence of the `end` expression. This expression is meant to represent the end of an array's bounds in an expression. What is rather strange, however, is that this expression is not only usable as an index, it can be used in sub-expressions, and its value changes depending on its position in the syntax tree. For example, if `A` is a matrix of size 4x7, then `A(1, floor(end/2))` gets us the element at position (1, 3), while `A(end, 1)` gets us the element at position (4, 1). This feature is often used to specify ranges going from a fixed value to the end of the array (e.g.: `A(2:end, 1)` extracts a portion of the first column of `A`).

To add to the confusion, the expression `A(end, end)` is valid in MATLAB R2009a, but `A(end, round(end/2))` is not, for reasons that are not explained in the MATLAB documentation (we suspect it may be the result of a parser bug, possibly because the `end` keyword also indicates the termination of code blocks). The Python programming language deals with this issue in part by allowing ranges with unspecified start or end indices, which automatically assume the value of the first or last array index (e.g.: `A(5:end)` is `A(5:)` in Python).

A more philosophical issue we have with the MATLAB semantics is that the behavior of functions can change in function of the number of output arguments assigned by the caller. Functions get the number of required output arguments as a hidden parameter when called and can alter their behavior based on it. This is again inconsistent with the behavior of most programming languages and requires special handling in a virtual machine's implementation. However, we also question the value of this feature. Clearly, in many cases, it will not help programmers write less code.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

These has been relatively little work done in the compiler research community as to the optimization of scientific and dynamic programming languages. To this day, most dynamic languages are interpreted because of the inherent difficulty in generating efficient compiled code for them. Mathworks has succeeded at creating a powerful, fast and flexible dynamic programming language for the scientific and engineering communities. However, their implementation of MATLAB is closed source and its internal workings remain a trade secret.

Through the McVM project, we have designed and implemented an optimizing virtual machine comprising a JIT compiler for a non-trivial subset of the MATLAB language. This virtual machine incorporates a powerful just-in-time type-based program specialization mechanism and additional optimizations which allow it to reach performance up to three orders of magnitude faster than competing MATLAB implementations such as GNU Octave.

One of our goals for the McVM virtual machine is to make it a viable product for end-users who seek a free and fast environment to develop software with the MATLAB language.

However, it is already of interest to the research community. McVM will soon be released under the liberal BSD open source license, and made available to other researchers. This will make it possible for others to use it as a testbed for novel compiler optimizations and language ideas.

We also hope that the success of our just-in-time program specialization scheme will inspire other researchers and implementers to investigate the use of such techniques in other dynamic languages. Much of the ideas used to improve the performance of our virtual machine, as well as the future research directions we suggest in section 8.2 would be easily applied to languages such as Python, Ruby or JavaScript, and some of these may well be key to the implementation of dynamic languages whose performance can compete with that of statically compiled languages.

8.2 Future Work

In this section we look into possible improvements to McVM which would address some of the more important performance issues with our current implementation. This includes matrix computation optimizations as well as improvements to our JIT such as the development of a lower level intermediate representation and a smarter type inference system to avoid performance degradation when sufficient type information cannot be inferred through traditional means.

We also examine longer term performance improvement strategies which could be applied to McVM as well as virtual machines and JIT compilers for other dynamic languages. These strategies include the use of adaptive optimizations as well as the implementation of a dynamic recompilation system. Finally, we discuss the idea of designing an improved dynamic language for scientific computation based on our experience with MATLAB.

8.2.1 Matrix Computation Optimizations

Our virtual machine currently treats matrix operations in a very generic way. When two matrices are added or multiplied, for example, a new matrix object is allocated to store the result. When a compound expression involving multiple matrices is evaluated, it is split into 3-address form, and each intermediate result is stored in a separated matrix that is allocated on the fly. This approach works reasonably well for strings, but it is obviously not the most effective way to handle matrix operations.

In programs that make extensive use of matrix computations, it is often the case that matrices are iteratively updated inside of loops. In this kind of scenario, our current approach allocates new matrices for every loop iteration. In such cases, it would likely be beneficial to reuse the same memory space for a given matrix variable, as well as for temporary results. This could be achieved through an analysis that maps matrix variables to pre-allocated “matrix registers”, thereby reducing allocation overhead and improving memory locality.

Another possibility for optimization would be to map some matrix operations directly to BLAS/LAPACK library calls. We currently do not use the full potential of these libraries. They expose some of the more common compound matrix operations which we currently implement using one or more intermediate steps. They also make it possible to operate directly on a single row or column of a matrix without first extracting the values from the said matrix. This could provide significant speed gains in some situations.

8.2.2 Secondary Intermediate Representation

At this point, the JIT compiler generates LLVM code based directly on our IIR tree form and the results of data flow analyses. It then relies on LLVM to perform the last optimization steps. However, LLVM lacks high-level information necessary to perform some optimizations. For example, it cannot know that some environment writes and reads (associated with interpreter fallback) are redundant or unnecessary. It is also unaware that some type conversions required by the JIT compiler could be eliminated.

It is likely that the code we generate could be optimized further if instead of generating LLVM code directly, we instead generated code in another intermediate form that exposes more higher-level information. This intermediate form could then be optimized to eliminate some redundant operations, possibly using very simple "peephole" pattern matching techniques. Dataflow analyses would also be more effective at a higher level. The optimized intermediate form would then be translated to LLVM code in a separate pass.

8.2.3 Smarter Type Inference

Knowledge of specific data types is important to optimization. Unfortunately, the need to be conservative in our type inference analysis means that unknown types dominate in merges. The result is that once "unknown" types are introduced, they often propagate and undermine the type inference efforts. Our code generation strategy is then left with very little information to operate on. In many cases, however, even if the type of a variable cannot be determined with 100% certainty, it may be possible to mitigate the impact of unknown types by predicting the most likely outcome.

A speculative design enables heuristic judgements. It is likely, for example, that if a variable is constantly added with integer matrices, it is also an integer matrix. Our code generation system could use these "best guesses" to generate an optimized code path. The types of variables can then be tested during execution and the optimized path chosen if appropriate. If the predicted type turns out to be wrong when the code executes, a default unoptimized path can be executed instead. We believe that such an approach would be likely to yield important speed gains, because the added overhead can be very small and the potential gains very significant.

8.2.4 Adaptive Optimizations

In our JIT compiler, optimizations are currently always applied in every case where they are applicable. This, however, is not always optimal, because some optimizations can, in some cases, reduce actual performance. Thus, it is perhaps desirable to implement a

system to try and predict which optimizations will yield performance gains. This could potentially be done by building a predictive model based on profiling information or code features. At run-time, we could let functions run for some time in interpreted mode before JIT compiling them, so as to gather profiling information, which would then be used to query the predictive model in order to make optimization decisions.

This idea is not novel. Most commercial optimizing JIT compilers analyze code and performance to make optimization decisions [AAF+05]. Our approach is essentially a combination of two existing ideas. Cavazos and O'Boyle have devised a system where code features are used to predict which optimizations will benefit a given method [CO06]. The features they use, such as counts of specific type of Java bytecode instructions, are meant to give the predictive model a "portrait" of what kind of operations a given method performs. Their approach uses a predictive model based on logistic regression (trained offline), and they have shown speedups of up to 33% on some Java benchmarks.

A similar approach proposed by Cavazos et. al uses hardware performance counters (e.g.: number of cache misses, floating-point operation statistics, etc.) to build a model of what optimizations are warranted by specific behaviors of a running program [CFA+07]. Their approach showed up to a 17% gain over the highest optimization setting of a commercial optimizing compiler. We believe that by combining this approach with profiling based on method features, it may be possible to achieve greater performance gains than can be achieved with either technique separately. Profiling data reveals important information about a program's actual behavior, while code features can be used to fine tune optimizations for each method of a program.

8.2.5 Dynamic Recompilation

We have mentioned in section 3.6 that we had placed restrictions on the power of some dynamic MATLAB features, notably the `eval` and `cd` constructs. These restrictions are due to the fact that our system only compiles functions once and makes optimization decisions at compilation time. Thus, our system must ensure that optimization decisions cannot be invalidated by dynamic features of the language.

One way to get around this limitation would be to make use of on stack replacement (see section 2.6) to dynamically recompile and replace functions once some of the assumptions made at compilation time become invalidated. This way, dynamic language constructs such as `eval` and `cd` can be left unrestricted and aggressive optimizations are still be performed. The obvious tradeoff, however, is that dynamic recompilation will incur some overhead.

8.2.6 Language Design

As examined in chapter 7, there are a number of issues that make it difficult to optimize the MATLAB language, and make it sometimes unintuitive for programmers. The MATLAB programming language also has no publically available specification document, making it difficult to write compatible implementations. These problems could potentially be fixed by Mathworks, but changing the syntax and semantics of an existing language with such a large following is impractical and unlikely to happen.

It would perhaps be desirable to implement a new dynamic language that shares the advantages of MATLAB in terms of productivity gains and convenience, but is designed with performance, consistency and intuitiveness in mind. New features such as improved reflectivity and metaprogramming capabilities could also be introduced. It may be advantageous if such a project came from the academic world, as it would make it more likely for the reference language implementation to be open source, and for there to be a publically available specification.

Designing a new programming language is obviously challenging. It is easy to design a language to be easy to optimize by introducing additional constraints, but there is a certain balance to be achieved between ease of optimization and convenience for the programmer. Dynamic languages are usually designed with programmer productivity as their primary goal and very little regard for performance. As discussed in chapter 7, some language design features of MATLAB complicate optimization while being rather arbitrary choices. We believe it should be possible to design a programming language that is easier to optimize without compromising its flexibility.

Bibliography

[AAF+05] Matthew Arnold, Matthew Arnold, Stephen J. Fink, Stephen J. Fink, David Grove, David Grove, Michael Hind, Michael Hind, Peter F. Sweeney, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. In *Proceedings of the IEEE, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation*, 2005.

[ABC+06] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the international symposium on Code Generation and Optimization (CGO)*, 2006, pages 295–305.

[AP02] George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceedings of the 2002 ACM conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, 2002, pages 294–303.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BdVG09] Paul Biggar, Edsko de Vries, and David Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, Honolulu, Hawaii, U.S.A, 2009, pages 1916–1923.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, San Jose, California, United States, 1996, pages 324–341.

[BS07] Hans-J. Boehm and Michael Spertus. Transparent programmer-directed garbage collection for C++, 2007.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 1977 ACM symposium on Principles Of Programming Languages (POPL)*, 1977, pages 238–252.

[CFA+07] John Cavazos, Grigori Fursin, Felix V. Agakov, Edwin V. Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the 2007 international symposium on Code Generation and Optimization (CGO)*, 2007, pages 185–197.

[CHK92] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. In *ICCL'92, Proceedings of the 1992 International Conference on Computer Languages*, 1992, pages 96–105.

[CO06] John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of the 2006 ACM conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Portland, Oregon, USA, 2006, pages 229–240.

[CU89] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *Proceedings of the 1987 symposium on interpreters and interpretive techniques*, 24(7):146–160, 1989.

[DB96] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[DRG+95] L. Derose, L. De Rose, K. Gallivan, K. Gallivan, E. Gallopoulos, E. Gallopoulos, B. Marsolf, B. Marsolf, D. Padua, and D. Padua. FALCON: a MATLAB interactive restructuring compiler. In *Proceedings of the 1995 conference on Languages and Compilers for Parallel Computing (LCPC)*, 1995, pages 269–288. Springer-Verlag.

[ELC03] Daniel Elphick, Michael Leuschel, and Simon Cox. Partial evaluation of MATLAB. In *Proceedings of the 2003 international conference on Generative Programming and Component Engineering (GPCE)*, Erfurt, Germany, 2003, pages 344–363. Springer-Verlag.

[FAFH09] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, Honolulu, Hawaii, 2009, pages 1859–1866.

[FH88] P. Fairfield and M. A. Hennell. Data flow analysis of recursive procedures. *ACM SIGPLAN notices*, 23(1):48–57, 1988.

[FQ03] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the 2003 international symposium on Code Generation and Optimization (CGO)*, San Francisco, California, 2003, pages 241–252. IEEE Computer Society, Washington, DC, USA.

[GES+09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, 2009, pages 465–478.

[GL02] Roberta Gori and Giorgio Levi. An experiment in type inference and verification by abstract interpretation. In *Revised Papers from the 2002 international*

workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI), 2002, pages 225–239. Springer-Verlag, London, UK.

[GV08] Dayong Gu and Clark Verbrugge. Phase-based adaptive recompilation in a JVM. In *Proceedings of the 2008 IEEE/ACM international symposium on Code Generation and Optimization (CGO)*, Boston, MA, USA, 2008, pages 24–34.

[HNK+00] Malay Haldar, Anshuman Nayak, Abhay Kanhere, Pramod Joisha, Nagaraj Shenoy, Alok Choudhary, and Prithviraj Banerjee. Match virtual machine: an adaptive runtime system to execute MATLAB in parallel. In *Proceedings of the 2000 International Conference on Parallel Processing (ICPP)*, 2000, pages 145–152.

[JB01] Pramod G. Joisha and Prithviraj Banerjee. Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In *Proceedings of the 2001 conference on APL*, New Haven, Connecticut, 2001, pages 7–21.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Lat02] Chris Lattner. LLVM: an infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[LL96] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the 1996 ACM conference on Programming Language Design and Implementation (PLDI)*, 1996, pages 137–148.

[MY99] Hidehiko Masuhara and Akinori Yonezawa. Generating optimized residual code in run-time specialization. In *1999 international colloquium on partial evaluation and program transformation*, 1999, pages 83–102.

[PC95] John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Proceedings of the 1995 workshop for Languages and Compilers for Parallel Computing (LCPC)*, 1995, pages 566–580.

[Rig04] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for Python. In *Proceedings of the 2004 ACM symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM)*, Verona, Italy, 2004, pages 15–26.

[RP96] Luiz De Rose and David Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proceedings of the 1996 International Conference on Supercomputing (ICS)*, Philadelphia, Pennsylvania, United States, 1996, pages 309–316.

[RP06] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 2006 ACM symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Portland, Oregon, USA, 2006, pages 944–953.

[SC03] Ulrik Schultz and Charles Consel. Automatic program specialization for Java. *ACM Transactions on programming languages and systems*, 25:2003, 2003.

[SD04] Kermit Sigmon and Timothy A. Davis. *MATLAB Primer, Seventh Edition*. Chapman & Hall/CRC, December 2004.

[Sin04] Jeremy Singer. Sparse bidirectional data flow analysis as a basis for type inference. In *Web proceedings of the Applied Semantics workshop (APPSEM)*, 2004.

[SYK+05] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM transactions on programming languages and systems*, 27(4):732–785, 2005.

[VE01] Michael J. Voss and Rudolf Eigemann. High-level adaptive program optimization with ADAPT. In *Proceedings of the 2001 ACM symposium on Principles and Practices of Parallel Programming (PPoPP)*, Snowbird, Utah, United States, 2001, pages 93–102.