Optimizing MATLAB through Just-In-Time Specialization *

Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge

School of Computer Science, McGill University, Montreal, QC, Canada
{mcheva,hendren,clump}@cs.mcgill.ca

Abstract. Scientists are increasingly using dynamic programming languages like MATLAB for prototyping and implementation. Effectively compiling MATLAB raises many challenges due to the dynamic and complex nature of MATLAB types. This paper presents a new JIT-based approach which specializes and optimizes functions on-the-fly based on the current types of function arguments.
A key component of our approach is a new type inference algorithm which uses the run-time argument types to infer further type and shape information, which in turn provides new optimization opportunities. These techniques are implemented in McVM, our open implementation of a MATLAB virtual machine. As this is the first paper reporting on McVM, a brief introduction to McVM is also given.
We have experimented with our implementation and compared it to several other MATLAB implementations, including the Mathworks proprietary system, McVM without specialization, the Octave open-source interpreter and the McFor static compiler. The results are quite encouraging and indicate that specialization is an effective optimization—McVM with specialization outperforms Octave by a large margin and also sometimes outperforms the Mathworks implementation.

1 Introduction

Scientists are increasingly using dynamic languages to prototype and implement their applications. MATLAB is particularly appealing because it has an interactive development environment, a rich set of libraries, and highly expressive semantics due to its dynamic nature. However, even though the dynamic nature of MATLAB may be convenient for scientists, it provides many challenges for effective and efficient compilation and execution. Furthermore, scientists would like to have reasonable performance as many scientific applications are computation-heavy and execute for a long time. Ideally this performance should be achieved without requiring a rewrite of MATLAB code to a more static language such as Fortran.

For good performance, we require an optimizing compiler that works directly on MATLAB programs. However, MATLAB poses several challenges. Firstly, MATLAB programs are normally developed incrementally, using an interactive

* This work was supported, in part, by NSERC and FQRNT.

development loop and mixing MATLAB scripts (a sequence of commands like those typed into the interactive loop prompt) with functions that are defined in separate source files. This means that code is dynamically-loaded and not all code is known ahead-of-time. Secondly, MATLAB's type system is both dynamic and intricate. The types of variables are not declared, but rather change as the computation proceeds. For example, it is not even straightforward to determine which values are scalars and which are arrays since a scalar assignment, such as `x = 1`, is assumed to define `x` as an 1×1 array. Furthermore, the size of an array dynamically increases as new values are written outside the current array bounds, and the effective base type of an array can change when an element of a more general type is written into it.

All of these challenges suggest that MATLAB is best optimized on-the-fly using a JIT compiler within a MATLAB Virtual Machine. We have developed a new open MATLAB VM called McVM which includes a JIT compiler built upon LLVM [1] which we briefly introduce in this paper. The main feature of the McVM JIT is a new on-the-fly specialization algorithm which specializes functions based on the run-time types of their arguments. This relies on a type and shape inference analysis which is specifically tailored to abstract the key features of the types in the function body. This type and shape analysis must be simple enough to work in the JIT context, but at the same time it must abstract the key features needed for optimization. Our approach is to combine 8 different simple abstractions, consisting of a variable's overall type, whether or not it is a scalar or a 2D matrix, its shape, and so on. The results of this type and shape inference analysis are then used to compile a specialized and optimized version of the function.

In order to determine the effectiveness of this argument-type-based specialization approach, we have implemented it and compared it against both McVM without specialization and three other existing MATLAB implementations: the Mathworks proprietary implementation, Octave[1] which is an open-source MATLAB interpreter, and McFor which is our group's static MATLAB-to-Fortran compiler. Initial results are quite encouraging and show that specialization works, provides good performance and that a reasonable number of specialized versions of functions are created.

The main contributions of this paper are:

McVM: an introduction of McVM giving our design criteria and an overview of the architecture of the system (Section 3);

Specialization: an introduction our approach for specializing functions on-the-fly based on the run-time types of function arguments (Section 4);

Type and Shape Inference: a new type and shape inference algorithm which approximates type and shape information based on argument types (Section 5); and

Experimental Validation: an experimental validation showing the overall effectiveness of McVM and the the effectiveness of specialization and type inference, in particular (Section 6).

[1] www.gnu.org/software/octave/

In the remainder of this paper we first describe the challenges of compiling MATLAB in Section 2, then we address each of our main contributions in Sections 3 through 6. We then discuss related work in Section 7 and give conclusions and future work in Section 8.

2 Optimization Challenges

MATLAB presents many challenges to an optimizing compiler. Traditional static optimization techniques do not work because of the highly dynamic nature and the complex semantics of the language. Dynamic loading of functions and scripts prevents us from assuming the entire program is known ahead of time, for example. One of the main challenges, however, is dealing with types, since the language is dynamically typed and follows intricate type rules.

Listing 1.1 shows an example of a simple program that illustrates some of the intricacies of the MATLAB type system. In this example, the `caller` function calls the `sumvals` function twice, with different argument types each time. The `sumvals` function is designed to sum numbers within a range of values. However, as this example illustrates, in MATLAB, it can be applied to both scalar types and arrays of values. Specifically, the variable `a` will be assigned the scalar integer value `5 * 10e11`, while `b` will be assigned the 1×2 floating-point array `1.0e12 * [0.8533 1.7067]`. These two values are then concatenated into `c`, a 1×3 array.

```
function s = sumvals(start, step, stop)
  i = start;
  s = i;

  while i < stop
    i = i + step;
    s = s + i;
  end
end

function caller()
  a = sumvals(1, 1, 10^6);
  b = sumvals([1 2], [1.5 3], [20^5, 20^5]);
  c = [a b];

  disp(c);
end
```

Listing 1.1. Implicit typing in MATLAB programs

Since the `sumvals` function can apply to either scalars or arrays, and the values operated on could be either integer, real or complex, compiling this program into efficient machine code can be challenging: type information is not explicit, and can change dynamically. A naive compiler could always store the variables inside the `sumvals` function as the widest available type (i.e.: complex matrices) or even generate code based on the idea that the type of all variables in the function are unknown, which is clearly very inefficient.

To generate efficient code, type inference is needed to extract implicit type information in the source program. In the case where `sumvals` is called with only

scalar integer inputs, it is possible to logically infer that all of the intermediate variables will also be scalar integers, and generate efficient code for this case. As for the case where `sumvals` is called with arrays as input, it should be possible to at least infer that complex values will never occur in the computation. This example motivates our approach of specialization based on the run-time types of arguments. Our approach will compile two different versions of the function based on the call signatures. This ensures that efficient code can be generated for each case. More details of our specialization technique are given in Section 4 and our type inference analysis is described in Section 5.

3 Design Overview

Our approach to optimization requires the ability to both interpret and compile multiple versions of code. The McVM virtual machine thus implements a mixed mode design, consisting of both interpreter and JIT components. The design is modular, making use of external front-end and low-level back-end components to simplify implementation complexity; Figure 1 shows the overall structure, which we now describe in more detail.

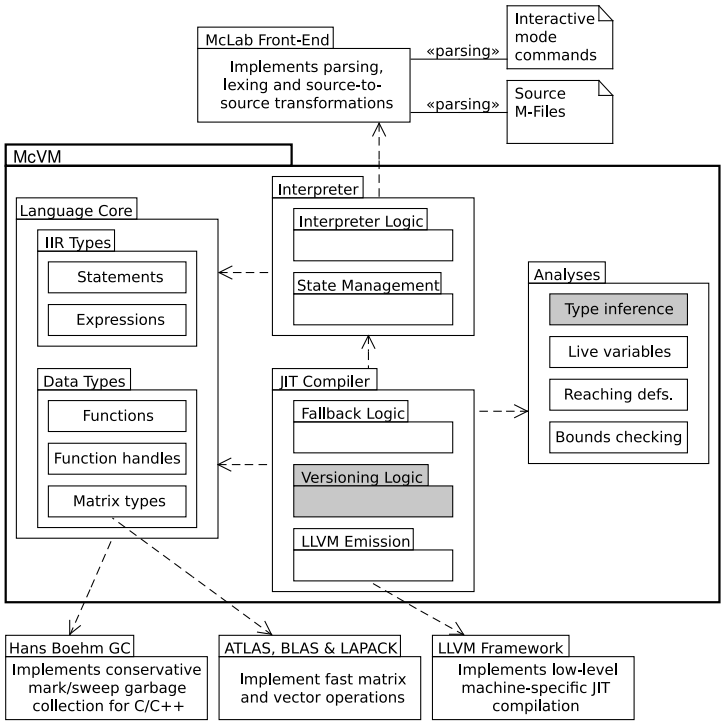

Fig. 1. Structure of the McVM Virtual Machine

The *McLab* front-end is used to parse interactive-mode commands and M-file source code, producing a common Abstract Syntax Tree (AST) representation

for both interpretation and compilation. The functionality of the interpreter is divided into interpretation logic and state management (housekeeping), while the JIT compiler manages the function specialization/versioning system, and generates low-level intermediate code for the statements it can compile. Our design allows for incremental and flexible development, with the JIT relying on the interpreter as fallback to evaluate code for which there is not yet compiler support.

At the core, McVM's implementation of matrix types depends directly on a set of mathematical libraries (ATLAS, BLAS and LAPACK) to perform fast matrix and vector operations. We use the Boehm garbage collector library for garbage collection [2], and our JIT compiler uses the LLVM framework to implement low-level JIT compilation and generate machine code [1]. The JIT compiler also implements several analyses to gain additional information about source programs being compiled, including basic analyses and optimizations such as live variables, reaching definitions, bounds check alimination, as well as type inference.

The McVM interpreter performs a straightforward, pre-order traversal of the internal AST in order to execute the input code. This interpretation approach is naive, but provides a correct, if low-performance execution that can serve both as a reference and act as a fallback when JIT compilation cannot be performed. The interpreter also serves housekeeping roles, providing essential run-time services. These include taking care of loading MATLAB files on-demand, executing interactive-mode commands, hosting library function bindings, maintaining bindings to global variables, and so forth.

The JIT compiler improves performance by translating high-level source code into a more efficient low-level form. A fundamental design goal in our VM was to aim for a simple and easily extensible design—similar to the `phc` compiler [3], our JIT compiler is built as an extension of the interpreter. The compiler can thus fall back to interpreting sections of code it cannot compile, mixing sections of both compiled and interpreted code in the execution of a given function. This allows for incremental JIT development, and also for language modifications to be more easily incorporated—new data types or statements can be added by modifying only the interpreter, relying on the fallback mechanism for any new features. The JIT compiler can later be modified, if necessary, to gain performance benefits from any additional optimization opportunities.

The JIT compiler performs actual code generation in conjunction with LLVM. During run-time, the input AST is first translated by our JIT compiler into a low-level, RISC-like Static Single Assignment (SSA) representation. From this, LLVM generates machine-specific executable code; LLVM also performs basic optimization passes on the code, such as constant propagation, dead code elimination and redundant operation elimination. 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.

Our fallback mechanism requires a high-level strategy to coordinate the transition from compiled code to interpretation and vice-versa. In particular, at each

step of the compilation process the JIT must track how and where each live variable is stored in order to appropriately transfer execution context. When interpreter fallback code is generated, instructions are issued to flush any register variables into memory for interpreter consumption. Upon returning to compiled execution, variables are copied back into their original registers. While spilling variables in this way is expensive, it has the advantage that the interpreter fallback mechanism does not impose extra penalties on compiled code in the case of functions which do not need to use it.

The McVM JIT compiler is able to compile and make use of specialized versions of functions based on call signatures. This corresponds to the two shaded boxes in Figure 1 labeled "Versioning Logic" and "Type Inference". In the next two sections we examine these two important components in more detail.

4 Just-In-Time Specialization

Exposing and using type information is central to most existing approaches to MATLAB optimization [4, 5]. McVM uses run-time type information to create multiple specialized versions of MATLAB functions. This allows for optimized function dispatch and improved code generation for many common operations, greatly reducing overhead costs necessary in a more generic design. Below we describe our precise versioning strategy, followed by core optimizations so enabled.

4.1 Function Versioning

Specialization requires creating type-specific versions of function bodies. This process is performed at run-time, by "trapping" commands issued through the interpreter (including calls made in the read-eval-print loop of the interactive mode). If the command is a call to a function (and not a script), the interpreter will try and pass control to the JIT compiler. When this happens, the JIT compiler builds an argument type string from the input arguments to the function, and attempts to locate a previously compiled version of the function with a matching argument type string. If none exists a new version will first be compiled, appropriately specialized to the given argument types. This removes significant dispatch overhead, allowing, for instance, scalar variables to be stored on the stack instead of as objects allocated on the heap. While compiling specialized function versions, the JIT compiler also considers functions called by the function being compiled, compiling them as direct calls to specialized versions as well. Thus entire executions can be specialized in a "deep" fashion.

As an example of how our function versioning works, consider the `sumvals` function shown earlier in Listing 1.1. This function is meant to sum numerical values in the range from start to stop, inclusively. In the absence of type information and specialization a compiler must make conservative assumptions, assuming iteration is potentially performed over arrays. Expensive heap storage

```
function s <scalar int> = sumvals(start <scalar int>, step <scalar int>, stop
      <scalar int>)

    i <scalar int> = start;
    s <scalar int> = i;

    while i < stop
        i <scalar int> = i + step;
        s <scalar int> = s + i;
    end

end
```

Listing 1.2. The type-annotated sumvals function

is thus required, as well as function calls to generically perform every operation (addition, comparison, etc.).

At an actual invocation of the function, however, such as in Listing 1.1: `a = sumvals (1, 1, 10^6);`, argument types are known to be scalar integers. This information is flowed through the function by our type inference, producing a type-annotated version as shown in Listing 1.2. From this, efficient code can be generated: all variables are easily stored on the stack, and there is no need to make expensive dispatches, because there are efficient machine instructions to add and compare scalar integer values.

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, should multiple versions be executed. We will see that this is not the case in practice (see Section 6). From our observations, MATLAB programs tend to have few long functions and fewer call sites than code written in other programming languages.

4.2 Additional Optimizations

Type-based specialization greatly simplifies basic arithmetic operations, allowing many uses of scalars to be implemented in just a few machine instructions. The type information, however, also facilitates the optimization of a number of other common operations, in particular certain array access operations, and use of library function calls. These optimizations improve performance by both taking advantage of type information, and eliminating cases where interpreter fallback is otherwise required.

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 behaviour is implemented through the interpreter, using the fallback mechanism to evaluate complex array reads and writes. When types are known, however, such as in `x = a(i);` where `i` is a scalar, optimized code can be generated to read or write the value directly. Type information includes array dimensions as well, eliminating the need for many dynamic array bounds 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 strategy is conservatively correct in the presence of unknown types, but can be inefficient because each call to these functions requires array allocation. Even for variables known to be scalar, the use of a generic library routine requires boxing and unboxing arguments and return values respectively, reducing the benefit from other optimizations.

To address these issues, we have devised a further simple specialization scheme for some library functions. Multiple, type-specific versions of library functions are first registered ahead-of-time in McVM. When a library function call is encountered, the JIT compiler will attempt to locate an appropriately specialized version, matching function argument and return types. An obvious example where this is beneficial is in the case of functions like `abs` or `sin`, where scalar data allows the direct use of the native C++ versions of these library functions.

5 Type and Shape Inference System

The McVM JIT compiler uses data provided by our type inference analysis to implement the just-in-time function specialization scheme described in Section 4. The more information the analysis provides about the concrete types and shapes of program variables, the more interpretive dispatching and storage overhead can be eliminated, and the faster the resulting compiled code will be, as demonstrated in Section 6.

Our type inference analysis 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 analysis is an abstract interpretation style analysis, which implements a compositional forward analysis directly on the structured AST representation. The analysis computes an abstraction of the actual types and shapes of variables at each program point. The actual abstraction is a carefully designed combination of simple abstractions, where each element of the abstraction captures a key aspect of the variable's type or shape. For example the *isScalar* flag indicates when a variable is definitely a scalar variable. If this flag is true, then the JIT compiler can allocate it to a register, which is much more efficient than storing it as a matrix. Another key point of our analysis is that it is flow-sensitive, and we thus have type and shape information for each program point.

5.1 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 abstract types. These sets contain zero or more *type abstractions* summarizing all possible types and shapes the specific variable can

have. Each type abstraction is actually an 8-tuple: $\langle overallType, is2D, isScalar, isInteger, sizeKnown, size, handle, cellTypes \rangle$.

If an abstract type set contains multiple type abstractions, 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 type abstraction in the set. The empty set is the $\perp$ element of the type lattice, representing situations where no information 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.

The core of the abstraction is the first item of the 8-tuple, the *overallType*, which represents a specific MATLAB language type, such as character array, floating-point matrix, complex number matrix, etc. Figure 2 represents the hierarchical type lattice of McVM *overallType* values.

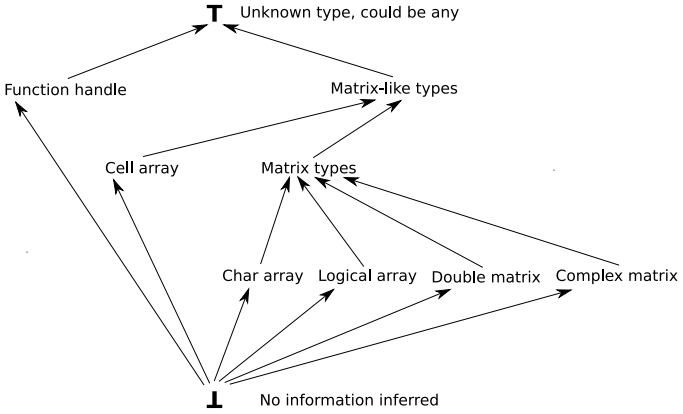

Fig. 2. Hierarchical lattice of McVM types

The remaining elements of each 8-tuple provide abstractions of different features of the type. Table 1 describes the fields stored in type objects. These fields cannot hold arbitrary values. For example, if the *isScalar* flag is set to True, then the *sizeKnown* flag must also be True. However, the *is2D* flag does not necessarily indicate that the matrix size is known.

For each statement in a program, our analysis produces a mapping of symbols to sets of type abstractions 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 abstractions 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):

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

5.2 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

Table 1. Description of type object fields

Field	Meaning/Description	Default
overallType	An element of the set of possible McVM data types.	Undefined
is2D	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)
isScalar	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)
isInteger	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.	$\perp$ (undefined)

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:

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

The joining of type sets is accomplished by using set union as a merge operator and then applying a filter operator to the result:

$$join(t_1,t_2) = filter\,(t_1 \cup t_2)$$

The filter operator takes a type set as input and returns a new type set in which all type objects having the same *overallType* value have been merged into one. It does so in a pessimistic way, that is, if one 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 for this flag. 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 input type objects did.

5.3 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:

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

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

As an example, we can look at the assignment `c = [a b];` in Listing 1.1. This represents the horizontal concatenation of arrays `a` and `b`. In this case, `a` holds the value `5 * 10e11`, which is a scalar integer value, and `b` holds the value `1.0e12 * [0.8533 1.7067]`, a 1×2 floating-point array. Thus, the type abstractions for `a` and `b` are:

$$\begin{aligned} type(a) = \{\langle & overallType = double, is2D = T, isScalar = T, isInteger = T, \\ & sizeKnown = T, size = (1,1), handle = null, cellTypes = \bot\rangle\} \end{aligned}$$

$$\begin{aligned} type(b) = \{\langle & overallType = double, is2D = T, isScalar = F, isInteger = F, \\ & sizeKnown = T, size = (1,2), handle = null, cellTypes = \bot\rangle\} \end{aligned}$$

The type rule associated with the horizontal concatenation operation allows us to infer that `c` will be a 1×3 floating-point array, that is:

$$\begin{aligned} outtype_{hcat}(type(a), type(b)) = \{\langle & overallType = double, is2D = T, isScalar = F, \\ & isInteger = F, sizeKnown = T, size = (1,3), \\ & handle = null, cellTypes = \bot\rangle\} \end{aligned}$$

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. 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:

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

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. Before we apply our type inference analysis, all loop statements are converted to `while` loops. As is the case for `if` statements, statements in the loop body are treated as one single compound statement. Special care is taken to properly deal with both `break` and `continue` statements.

5.4 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 inferred output type of each statement is stored in a global mapping (e.g.: hash map) of the types at the output of each statement.

6 Evaluation

In order to assess the performance of our virtual machine we compare the actual performance of McVM to that obtained by several related systems: 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 are intended to give us some idea of how well our current solution performs against competing implementations. The McFor numbers are provided as a rough "upper bound" on performance—Fortran compilers are known to perform very well on numerical computations, giving an indication of potential compiler performance for non-interactive code.

We have performed our tests on a total of 20 benchmark programs. These benchmarks are gathered from previous work on optimizing MATLAB[2], in the FALCON [6] and OTTER projects, Mathworks' CentralFile Exchange, Chalmers University, and from individual course work and student projects at McGill. 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. The left part of Table 2 provides characteristic numbers for each of the benchmarks supported by McVM. Number of functions and statements (3-address form) relate to the overall (static) input load on our system, while number of call sites directly affects specialization. Maximum loop nesting depth affects the theoretical efficiency of our dataflow analysis.

Not all benchmarks benefit equally from our optimizations of course, and in the following sections we show further profiling numbers intended to explain where specific performance bottlenecks occur. Section 6.2 describes the behaviour of the type inference system, while Section 6.3 gives data on the specialization system, including compiler overhead. 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 9.10

[2] http://www.ece.northwestern.edu/cpdc/pjoisha/MAT2C/

(linux kernel 2.6.31, 32-bit). We have gathered our MATLAB performance numbers using MATLAB R2009a, and our GNU Octave numbers on Octave version 3.0.5. The Fortran code produced by McFor was compiled using the GNU Fortran compiler version 4.4.1. Because of significant variance when timing benchmarks, attributable to i-cache effects and the garbage collector, all benchmark timing measurements are based on an average over 10 runs.

6.1 Baseline Performance

The rightmost columns of Table 2 show a comparison of benchmark running times under our four execution environments, as well as a version of McVM with the JIT and specialization disabled, giving absolute time as well as times normalized to the McVM JIT (values greater than 1 are running slower than McVM with JIT). As we can see, McVM with JIT performs better than MATLAB in 8 out of 20 benchmarks, sometimes by a fair margin. In the cases where it does worse than MATLAB, the running times can be relatively close (as with `nnet`), or, as exemplified by the `crni` benchmark, sometimes dramatically less; we discuss reasons for this poor performance in Section 6.2.

GNU Octave, possessing no JIT compiler, does rather poorly in general. It trails far behind MATLAB and outperforms McVM with JIT on only 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 well ahead of MATLAB and McVM, with the exception of the `clos` benchmark. This suggests that MATLAB and McVM both are still far from the "optimal" performance level.

6.2 Type Inference Efficiency

Our ability to optimize strongly depends on the behaviour of our type inference system. The leftmost part of Table 3 thus shows relevant run-time profiling information, dynamically weighted by the relative execution counts of the associated statements. The first data column gives the percentage of type sets that are at top, providing no type information, while column 3 shows the percentage of type sets which contain only one type, and so give exact type data. The third column shows the percentage of times where variables holding scalar values were known ahead of time to be scalar, and the fourth column is the percentage of times where the size of matrix variables was known by the type inference system.

In general the more type information our system has the better it will be able to optimize code generation. Knowledge of which variables are scalars is even more critical, however, 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% of scalar variables known. The behaviour of the `crni` benchmark can also be explained by this data. As can be seen in Table 3, scalars are known in only 68.7% of cases, one of the lowest such ratios. An examination of the code reveals this benchmark uses matrix "creation on

Table 2. Benchmark characteristics and comparison of running times. Columns 6–10 give absolute running times, while columns 11–14 are performance normalized to McVM JIT. The geometric mean was used for relative values (columns 11–14).

	Static Measurements				Performance (s)					Relative to McVM JIT			
Benchmark	Functions	Stmts	Loop Nesting	Call Sites	McVM JIT	MATLAB	McVM no JIT	Octave	McFor	MATLAB	McVM no JIT	Octave	McFor
adpt	2	196	2	6	13.4	2.66	12.6	45.9	0.72	0.20	0.94	3.42	0.05
beul	10	511	1	38	3.07	3.09	1.56	7.62	N/A	1.01	0.51	2.49	N/A
capr	5	214	2	10	3.51	8.10	1674	5256	1.26	2.31	478	1499	0.36
clos	2	58	2	3	6.84	0.75	13.6	17.5	7.87	0.11	1.99	2.56	1.15
crni	3	142	2	7	1321	6.95	1788	5591	3.56	0.01	1.35	4.23	0.00
dich	2	144	3	7	2.80	4.71	1149	4254	1.88	1.68	410	1517	0.67
diff	2	253	3	6	30.0	5.26	41.9	120	0.65	0.17	1.39	3.98	0.02
edit	2	130	2	6	54.9	11.0	81.4	394	0.13	0.20	1.48	7.17	0.00
fdtd	2	157	1	3	20.1	3.32	8.56	172	0.29	0.17	0.43	8.55	0.01
fft	2	159	3	8	12.8	16.2	2470	8794	9.13	1.27	193	689	0.72
fiff	2	120	2	4	5.37	6.97	1528	4808	0.99	1.30	285	895	0.18
mbrt	3	78	2	11	34.6	4.53	98.6	295	0.96	0.13	2.84	8.51	0.03
nb1d	3	194	2	11	4.10	9.85	4.24	43.9	0.74	2.40	1.03	10.7	0.18
nb3d	3	164	2	12	3.88	1.54	2.51	40.8	0.89	0.40	0.65	10.5	0.23
nfrc	5	151	2	11	15.7	4.94	26.0	80.3	N/A	0.32	1.66	5.13	N/A
nnet	4	186	3	16	6.95	6.35	7.32	26.5	N/A	0.91	1.05	3.81	N/A
play	6	364	2	29	3.37	8.68	4.24	29.0	N/A	2.57	1.26	8.60	N/A
schr	8	203	1	32	2.48	2.07	3.03	2.31	N/A	0.84	1.22	0.93	N/A
sdku	9	363	2	49	1.23	9.74	16.0	112	N/A	7.93	13.1	90.9	N/A
svd	11	308	3	42	8.24	2.38	7.02	10.9	N/A	0.29	0.85	1.33	N/A
mean	4.3	205	2.1	15.6	77.7	5.96	447	1505	2.24	0.49	3.91	15.4	0.08

assignment" to initialize its input data, resulting in several unknown types being propagated through the entire program. We examine ways to fix this weakness of our type inference system as part of future work.

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% scalar variables known and 96.9% singleton type sets. Most of these benchmarks makes heavy use of complex slice read operations operating on entire columns or rows of a matrix at a time, and these are currently implemented through our (expensive) interpreter fallback mechanism.

6.3 JIT Specialization

The benefit of JIT specialization depends on how well it improves the code as well as any introduced overhead. The rightmost three columns of Table 3 show the effect of JIT compilation on three profile measures, the number of matrices

Table 3. Profiled performance. All values are percentages.

Benchmark	Top sets	Singleton sets	Scalars known	Size known	JIT speedup	Matrices created	Slice reads	Env. lookups
adpt	4.18	95.8	100	90.0	-6.82	24.8	16.8	39.2
beul	55.2	44.8	71.3	29.5	-96.3	85.5	49.8	114
capr	0.01	100	100	82.8	99.8	0.00	0.00	0.00
clos	0.00	100	100	99.9	49.7	0.00	100	0.00
crni	19.1	71.4	68.7	54.8	26.1	66.7	69.2	55.2
dich	2.09	97.9	100	85.1	99.8	0.00	0.00	0.00
diff	14.3	82.1	66.7	66.7	28.2	68.3	100	2.45
edit	5.14	94.9	96.8	81.5	32.5	65.0	40.0	81.6
fdtd	0.01	100	100	49.8	-135	88.1	90.0	90.5
fft	0.00	100	100	80.3	99.5	0.00	0.00	0.00
fiff	0.01	100	100	86.1	99.6	0.01	0.00	0.00
mbrt	9.09	90.9	100	100	64.9	33.3	100	0.00
nb1d	5.84	94.2	88.1	34.5	3.33	75.6	0.00	14.9
nb3d	3.13	96.9	100	16.5	-54.6	94.0	98.3	76.2
nfrc	16.4	82.7	100	98.9	39.8	42.5	100	19.8
nnet	52.6	47.4	98.7	55.1	5.08	86.9	100	82.8
play	23.3	66.6	77.5	52.1	20.6	72.5	100	45.9
schr	31.8	55.3	99.5	41.7	18.3	65.5	54.0	84.6
sdku	14.8	85.2	83.8	49.7	92.3	7.55	5.69	4.65
svd	16.4	73.8	94.2	59.7	-17.4	84.7	100	60.2
mean	13.7	84.0	92.3	65.7	23.5	48.0	56.2	38.6

created, the number of slice reads, and the number of environment lookups, in each case presented as a percentage of the original, interpreted quantity. These are all expensive operations, and so large reductions should map to large improvements from JIT compilation. The `fft` benchmark, for instance, has 100% of its 789 million interpreter slice reads eliminated, and runs over 190 times faster with the JIT compiler enabled.

For a better understanding of the cost/benefit of different components of our system, we also evaluate the performance of McVM with specific JIT optimizations disabled. Relative to the McVM JIT compiler with all optimizations enabled, the five leftmost columns in Table 4 show the ratio of run-times of McVM with optimizations to arithmetic operations, array operations, function calls, specialized library functions, and the entire JIT selectively disabled (a number greater than one signifies a slowdown). Clearly, arithmetic operation and array access optimizations have a tremendous impact as they speed up several benchmarks by two orders of magnitude. In certain cases, such as `dich`, optimizing library functions also has a large impact.

The direct call mechanism has much less impressive benefits. It improves benchmarks that perform many function calls, but can also yield lower perfor-

Table 4. Relative JIT performance with specific optimizations disabled (columns 2–6), and overhead of the optimization system (columns 7–10). The geometric mean was used for relative values (columns 2–6).

Benchmark	Arith.	Array	Direct calls	Library	JIT	# functions	# versions	Compile (s)	Analysis (s)
adpt	1.43	1.12	0.97	1.07	0.94	2	2	0.86	0.79
beul	1.03	1.00	1.00	1.00	0.51	9	16	1.20	0.90
capr	590	428	1.73	1.05	478	5	5	0.50	0.43
clos	3.40	1.01	1.00	1.00	1.99	2	2	0.14	0.12
crni	1.63	1.27	0.75	0.99	1.35	3	3	0.32	0.26
dich	459	282	1.00	29.7	410	2	2	0.38	0.32
diff	2.20	1.03	1.01	0.96	1.39	2	2	1.19	1.10
edit	1.90	1.46	0.61	0.98	1.48	2	2	0.22	0.17
fdtd	1.25	1.10	1.01	0.87	0.43	2	2	0.48	0.38
fft	144	143	1.02	1.01	193	2	2	0.58	0.54
fiff	280	204	1.01	1.05	285	2	2	0.24	0.20
mbrt	3.57	1.05	1.05	0.99	2.84	3	3	0.14	0.11
nb1d	0.90	1.22	1.06	0.97	1.03	3	3	0.51	0.42
nb3d	0.66	1.07	1.08	0.97	0.65	3	3	0.57	0.46
nfrc	1.33	1.04	1.77	0.98	1.66	5	5	0.22	0.15
nnet	1.20	1.01	1.02	0.98	1.05	4	4	0.36	0.29
play	1.21	1.03	1.11	0.98	1.26	6	10	0.58	0.42
schr	1.47	1.00	1.02	1.00	1.22	8	9	0.55	0.45
sdku	1.42	1.67	1.13	0.97	13.1	9	11	1.08	0.85
svd	3.92	0.98	1.05	0.98	0.85	11	15	0.79	0.61
mean	4.56	3.28	1.04	1.17	3.91	4.2	5.2	0.55	0.45

mance in cases where the 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 with direct calls disabled.

Given our specialization strategy, compilation overhead is a concern—if types are highly variable, many function versions will be compiled, adding CPU and memory overhead. We thus measured the number of functions compiled, as well as the total number of specialized versions for each of our benchmarks. Columns 7 and 8 in Table 4 show that excessive specialization is not a problem in practice. In most cases functions are always called with the same argument types, and there are never more than twice as many versions as compiled functions.

The last two columns of Table 4 give the absolute compile-time overhead and its analysis-time constituent. As we can see, most of the compilation time is spent performing analyses on the functions to be compiled, as opposed to code generation. The slowest compilation time is associated with the `diff` benchmark. We attribute this to the large quantity of code contained in a triple nested loop in this benchmark, for which our analyses take longer to compute a fixed

point. In most cases these costs are not excessive and are easily overcome by the performance improvement, especially for longer running benchmarks.

7 Related Work

Our approach to optimizing MATLAB has concentrated on dynamic features of the language that interfere with more traditional optimization. This brings together more traditional work on compiling scientific, array-intensive languages and techniques for optimizing dynamic languages, and specifically dynamic specialization and type inference.

Previous compiler approaches to MATLAB have mainly focused on numerical performance, primarily in the context of static language subsets or contexts. As well as more traditional loop and array optimizations, code restructuring can be performed to ensure programs take good advantage of optimized instrinics [7]. Good performance can also be achieved by translating MATLAB code to other static languages, such as C [8] or Fortran 90 [6, 9], where further aggressive optimization or parallelization can be performed. A major source of complexity for almost all MATLAB optimizations, as in our case, is analyzing and understanding array properties, such as shape and size [10]. Elphick et al. identify similar typing and dynamic language concerns in their partial evaluation approach to optimizing MATLAB programs [5]. They develop *MPE*, an online system to partially evaluate MATLAB source functions into more efficient MATLAB code. Their design is intra-procedural and does not handle polyvariant types, but as such may provide an additional and orthogonal benefit to our approach.

Full VM approaches have also been applied, including JIT-based solutions. *MaJIC* combines JIT-compilation with an offline code cache maintained through speculative compilation of MATLAB code into C/Fortran [4]. They derive the most benefit from optimizations such as array bounds check removals and register allocation. The *Match* VM project translates MATLAB programs to a lower-level intermediate form which is then analyzed for dependencies and used to automatically parallelize computation [11]. The result is invisible to the user, and by relying on run-time estimates for scheduling avoids static array analysis requirements.

Program Specialization We use program specialization [12] in order to optimize effectively in the presence of imprecise type information. More specifically, we apply procedure cloning [13] to create specialized copies of function bodies in which we can make stronger typing assumptions. Such specialization techniques have previously been used offline to translate MATLAB code into optimized C or Fortran code [14]. Our design extends on run-time specialization techniques used by languages such as SELF [15] and is similar to the approach used to optimize the JIT compilation of generics for the C# language [16]. More general specialization designs have also been applied [17]. In practice this can yield very significant performance gains—Schultz and Consel report speedups of up to 300% for their specializing *JSpec* Java compiler [18].

Run-time specialization accommodates MATLAB's dynamic nature, and is a technique that has been applied in many other dynamic optimization contexts. The *Psyco* python virtual machine, for instance, implements specialization "by need" [19], a process similar to the online partial evaluation approaches applied to MATLAB [5] and Maple [20]. This specialization technique involves interleaving program specialization and execution; the specializer can request facts such as the type of variables while a procedure is executing, and depending on the result, potentially modify the compiled code to be more efficient. The design goal was to eliminate much of the interpretative overhead through the use of JIT compilation, without sacrificing the dynamic features of the language. Approaches such as *Psyco* differ from our system by working on fine-grain code fragments rather than functions, trading simpler code-generation and analysis requirements for smaller specialized sequences.

Similar to the Psyco effort, the *TraceMonkey* VM for the JavaScript language has focused on just-in-time specialization based on type information in order to increase performance [21]. The design 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. 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 throughout 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.

Type Inference Our specialization approach is facilitated by a type inference analysis [22], where we use a straightforward, if non-trivial dataflow analysis to determine type information. The problem, of course, has been examined in many contexts, and poses an efficiency and accuracy trade-off even in the case of statically typed languages, such as C++ [23] or Java [24]. In these cases relatively cheap *flow-insensitive* approaches to type analysis have been shown effective. In a more general and flow-sensitive sense the type inference problem can also be seen as a bidirectional dataflow analysis, propagating type information both along and against the direction of control flow [25]. In most such analyses types are considered static, although dynamic types may be reduced to static types through the use of a Static Single Assignment (SSA) representation.

Type inference on dynamic languages brings additional complexity. Constructs like `eval`, MATLAB's `cd`, as well as dynamic loading and reflection features, make it difficult or 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 [26] and Ruby [27]. These approaches show potential to detect type errors ahead of time, but they do not address the aforementioned problems. Our approach, on the other hand, can operate on programs whose call graphs are not fully known ahead of time.

8 Conclusions and Future Work

Our experience with McVM demonstrates that online specialization is an effective and viable technique for optimizing MATLAB programs. Although other specialization and partial evaluation approaches have been applied to MATLAB [4, 5] and similar dynamic language contexts [19, 21], we provide an efficient and full JIT solution. Our approach focuses on optimizing code generation, uses a coarse-grained strategy that minimizes specialization overhead, and is specifically designed to accommodate complex dynamic language properties. Combined with an effective type and shape inference strategy, McVM is able to achieve performance up to three orders of magnitude faster than competing MATLAB implementations such as GNU Octave, and in several cases faster than the commercial product.

Further improvements to performance are possible in a number of ways. 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 repeatedly added to integer matrices, that 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 either an optimized path or default code chosen as appropriate. Speculative approaches have been successful based on external compilation [4], and a JIT-based solution has potential to yield further significant speed gains.

References

1. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Comp. Sci. Dept., U. of Illinois at Urbana-Champaign (Dec 2002)
2. Boehm, H., Spertus, M.: Transparent programmer-directed garbage collection for C++ (2007)
3. Biggar, P., de Vries, E., Gregg, D.: A practical solution for scripting language compilers. In: SAC '09, ACM (2009) 1916–1923
4. Almási, G., Padua, D.: MaJIC: compiling MATLAB for speed and responsiveness. In: PLDI '02, ACM (2002) 294–303
5. Elphick, D., Leuschel, M., Cox, S.: Partial evaluation of MATLAB. In: GPCE '03, Springer-Verlag (2003) 344–363
6. Derose, L., Rose, L.D., Gallivan, K., Gallivan, K., Gallopoulos, E., Gallopoulos, E., Marsolf, B., Marsolf, B., Padua, D., Padua, D.: FALCON: A MATLAB interactive restructuring compiler. In: LCPC '95, Springer-Verlag (1995) 269–288
7. Birkbeck, N., Levesque, J., Amaral, J.N.: A dimension abstraction approach to vectorization in matlab. In: CGO '07, IEEE Computer Society (2007) 115–130

8. Joisha, P.G., Banerjee, P.: A translator system for the MATLAB language: Research articles. Softw. Pract. Exper. **37**(5) (2007) 535–578
9. Rose, L.D., Padua, D.: A MATLAB to Fortran 90 translator and its effectiveness. In: ICS '96, ACM (1996) 309–316
10. Joisha, P.G., Banerjee, P.: An algebraic array shape inference system for MATLAB®. ACM Trans. Program. Lang. Syst. **28**(5) (2006) 848–907
11. Haldar, M., Nayak, A., Kanhere, A., Joisha, P., Shenoy, N., Choudhary, A., Banerjee, P.: Match virtual machine: An adaptive runtime system to execute MATLAB in parallel. In: ICPP2000. (2000) 145–152
12. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, Inc. (1993)
13. Cooper, K.D., Hall, M.W., Kennedy, K.: Procedure cloning. In: Computer Languages. (1992) 96–105
14. Chauhan, A., McCosh, C., Kennedy, K., Hanson, R.: Automatic type-driven library generation for telescoping languages. In: SC '03. Volume 1. (1917) 58113–695
15. Chambers, C., Ungar, D.: Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. SIGPLAN Not. **24**(7) (1989) 146–160
16. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET Common Language Runtime. In: PLDI '01, ACM New York, NY, USA (2001) 1–12
17. Shankar, A., Sastry, S.S., Bodík, R., Smith, J.E.: Runtime specialization with optimistic heap analysis. SIGPLAN Not. **40**(10) (2005) 327–343
18. Schultz, U., Consel, C.: Automatic program specialization for Java. ACM Trans. Program. Lang. Syst. **25**(4) (2003) 452–499
19. Rigo, A.: Representation-based just-in-time specialization and the Psyco prototype for Python. In: PEPM '04, ACM (2004) 15–26
20. Carette, J., Kucera, M.: Partial evaluation of Maple. In: PEPM '07, ACM (2007) 41–50
21. Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: PLDI '09, ACM (2009) 465–478
22. Duggan, D., Bent, F.: Explaining type inference. In: Science of Computer Programming. (1996) 37–83
23. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA '96, ACM (1996) 324–341
24. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: OOPSLA '00, ACM (2000) 281–293
25. Singer, J.: Sparse bidirectional data flow analysis as a basis for type inference. In: Web proceedings of the Applied Semantics Workshop. (2004)
26. Joisha, P.G., Banerjee, P.: Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In: APL '01, ACM (2001) 7–21
27. Furr, M., An, J.h.D., Foster, J.S., Hicks, M.: Static type inference for Ruby. In: SAC '09, ACM (2009) 1859–1866