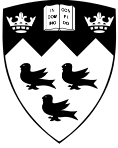

McGill University
School of Computer Science
Sable Research Group

Taming MATLAB

Sable Technical Report No. sable-2011-04

Anton Dubrau and Laurie Hendren

December 17, 2011

Contents

1 Introduction **4**

2 MATLAB - a dynamic language **6**

2.1 Basics 6

2.2 Matlab Type System 6

2.3 Matlab Functions and Specialization 7

2.4 Matlab Classes 9

2.5 Function Handles 9

2.6 Function Parameters and Arguments 9

2.7 Wild Dynamic Features 10

2.8 Summary 10

3 Framework for Builtins **11**

3.1 Learning about Builtins 11

3.1.1 Identifying Builtins: 11

3.1.2 Finding Builtin Behaviours: 12

3.2 Specifying Builtins 13

3.3 Specifying Builtin attributes 14

3.4 Summary 15

4 Tame IR **15**

4.1 Specialized AST nodes 16

4.2 Lambda Simplification 16

4.3 Switch simplification 16

5 Interprocedural Value Analysis and Call Graph Construction **17**

5.1 The Interprocedural Analysis Framework 18

5.2 Introducing the Value Analysis 19

5.2.1 Mclasses, Values and Value Sets: 19

5.2.2 Flow Sets: 20

5.2.3 Argument and Return sets: 20

5.2.4 Builtin Propagators: 21

5.3 Flow Equations 21

5.4 Structures, Cell Arrays and Function Handles 21
5.4.1 `struct`, `cell`: 22
5.4.2 `function_handle`: 22
5.5 The Simple Matrix Abstraction 22
5.6 Applying the Value Analysis 23

6 Related Work **24**

7 Conclusions and Future Work **25**

List of Figures

1 Overview of the MATLAB Tamer 5
2 Superior/inferior class relationships for MATLAB 8
3 Example mclass results for groups of Built-in binary operators 12
4 Subtree of builtin tree, showing all defined floating point builtins of MATLAB 14
5 Specializations of an assignment statement 16
6 Transforming `lambda` expressions 17
7 Transforming `switch` statements 17

List of Tables

I Results of Running Value Analysis 23

Abstract

MATLAB is a dynamic scientific language used by scientists, engineers and students worldwide. Although MATLAB is very suitable for rapid prototyping and development, MATLAB users often want to convert their final MATLAB programs to a static language such as FORTRAN. This paper presents an extensible object-oriented toolkit for supporting the generation of static programs from dynamic MATLAB programs. Our open source toolkit, called the MATLAB Tamer, identifies a large tame subset of MATLAB, supports the generation of a specialized Tame IR for that subset, provides a principled approach to handling the large number of builtin MATLAB functions, and supports an extensible interprocedural value analysis for estimating MATLAB types and call graphs.

1 Introduction

MATLAB is a popular numeric programming language, used by millions of scientists, engineers and students worldwide [13]. MATLAB programmers appreciate the high-level matrix operators, the fact that variables and types do not need to be declared, the large number of library and builtin functions available, and the interactive style of program development available through the IDE and the interpreter-style read-eval-print loop. However, even though MATLAB programmers appreciate all of the features that enable rapid prototyping, they often have other ultimate goals. Frequently their computations are quite computationally intensive and they really want an efficient implementation. Programmers also often want to integrate their MATLAB program into existing static systems. As just one example, one of our users wanted to generate FORTRAN code that can be plugged into a weather simulation environment.

This paper addresses the problem of how to provide the bridge between the dynamic realities of MATLAB and the ultimate goal of wanting efficient and static programs in languages like FORTRAN. It is not realistic to support all the MATLAB features, but our goal is to define and provide support for a very large subset of MATLAB which includes dynamic typing, variable numbers of input and output arguments, support for a variety of MATLAB data types including arrays, cell arrays and structs, and support for function handles and lambda expressions.

Providing this bridge presents two main challenges. The first is that MATLAB is actually quite a complex language which has evolved over many years and which has non-standard type rules and function lookup semantics. The second major challenge is properly dealing with the large number of builtin and library functions, which have also been developed over time and which sometimes have unexpected or irregular behaviour.

Our solution is an open-source extensible objected-oriented framework, implemented in Java, as presented in Fig. 1. The overall goal of the system is to take MATLAB programs as input and produce output which is suitable for static compilation, a process that we call *Taming* MATLAB. Given a `.m` file as input, which is the entry point, the MATLAB Tamer produces as output: (1) a Tame IR for all functions (both user and library) which are reachable from the entry point, (2) a complete call graph, and (3) an estimation of classes/types for all variables.

There are some features in MATLAB that are simply too wild to handle, and so our system will reject programs using those features, and the user will need to refactor their program to eliminate that feature. Thus, another important goal in our work is to define as large as possible subset of MATLAB that can be tamed without user intervention.

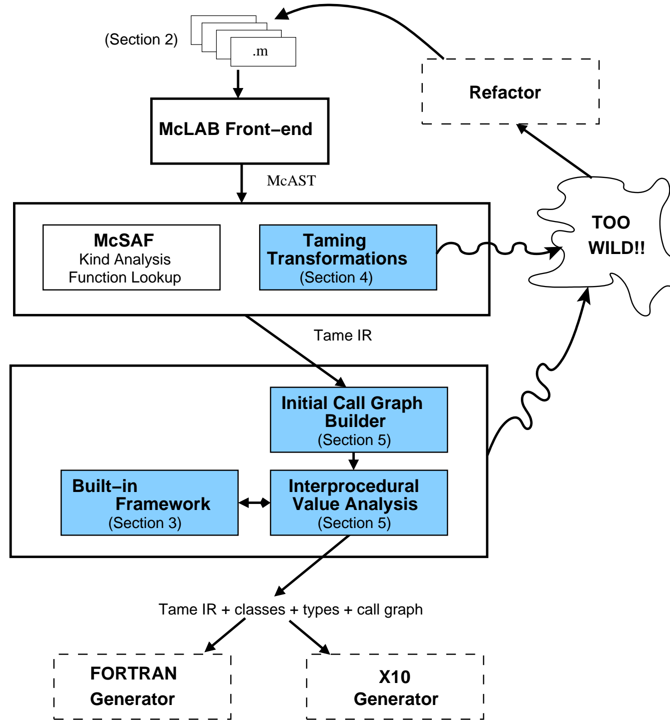

Figure 1: Overview of our Matlab Tamer. The shaded boxes indicate the components presented in this paper. The other solid boxes correspond to existing McLab tools we use, and the dashed boxes correspond to ongoing projects which are using the results of this paper.

The main contributions of this paper are as follows.

- We present an overall design and implementation for the Matlab Tamer, an extensible object-oriented framework which provides the bridge between the dynamic Matlab language and a static back-end compiler.
- We describe the key features of Matlab necessary for compiler developers and for tool writers to understand Matlab and the analyses in this paper. We hope that by carefully explaining these ideas, we can enable other researchers to also work on static tools for Matlab. Our discussion of Matlab features also motivates our choice of the subset of Matlab that we aim to tame.
- We provide a principled approach to understanding, grouping, and analyzing the large number of Matlab builtin functions.
- We developed extensions to the McSaf [4] framework to support a lower-level and more specialized Tame IR, suitable for back-end static code generation.
- We present an interprocedural flow analysis framework that computes both abstract values and the complete call graph. This flow analysis provides an object-oriented approach which allows for extension and refinement of the abstract value representations.

The remainder of the paper is structured as follows. Sec. 2 introduces the key Matlab features, Sec. 3 describes our approach to Matlab builtins, Sec. 4 describes the Tame IR and transforma-

tions, Sec. 5 explains our extensible and interprocedural value analysis and call graph construction, Sec. 6 provides an overview of related work and Sec. 7 concludes.

2 MATLAB - a dynamic language

In this section we describe key MATLAB semantics and features to provide necessary background for compiler writers and tool developers to understand MATLAB and its challenges, and to motivate our approach of constructing a "tame" intermediate representation and MATLAB callgraph. In each subsection we give a description followed by annotated examples using the MATLAB read-eval-print loop. In the examples, ">>" indicates a line of user input, and the following line(s) give the printed output.

2.1 Basics

MATLAB was originally designed in the 1970s to give access to features of FORTRAN (like LINPACK, EISPACK) without having to learn FORTRAN [14]. As the name MATLAB(MATrix LABoratory) suggests, MATLAB is centered around numerical computation. Floating point matrices are the core of the language. However, the language has evolved beyond just simple matrices and now has a type system including matrices of different types, compound types including cell arrays and structs, and function references.

Given its origins, MATLAB is a language that is built around matrices. Every value is a *Matrix* with some number of dimensions, so every value has an associated array shape. Even scalar values are 1×1 matrices. Vectors are either $1 \times n$ or $n \times 1$ matrices and strings are just vectors of characters. Most operations are defined for matrices, for example `a * b` specifies matrix multiplication if both `a` and `b` are matrices. Operators are overloaded and sometimes refer to scalar operations, for example `a * b` specifies an element-wise multiplication if `a` is a matrix and `b` is a scalar.

```
>> size(3)              % the scalar 3 is a 1x1 matrix
    1     1
>> size([1 2 3])        % a 1x3 vector
    1     3
>> size([5; 6; 7; 8; 9]) % a 5x1 vector
    5     1
>> size('hello world') % a string, which is a 1x11 vector
    1    11
>> ['a' 'b'; 'e' 'f']  % a 2-dimensional matrix of characters
    ab
    ef
```

2.2 MATLAB Type System

MATLAB is dynamically typed - variables need not be declared, they will take on any value that is assigned to them. Every MATLAB value has an associated MATLAB class (henceforth we will use the name *mclass* when referring to a MATLAB class, in order to avoid confusion with the usual notion of a class). The mclass generally denotes the type of the elements of a value. For example, the mclass of an array of doubles is `double`. The default numeric mclass is `double`. While MATLAB also includes integer types, all numeric literals are doubles.

```
>> n = 1          % the input  literal  and the ouput look  like  an  integer
    1
>> class(n)       % however the mclass  is  is   really  double,  the  default
    double
>> class(1:100)  % the mclass  of  the  vector  [1,  2,   ...,   100]  is  double
    double
```

MATLAB has a set of builtin mclasses, which can be summarized as follows:

- `double`, `single`: floating point values
- `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `in32`, `int64`: integer values
- `logical`: boolean values
- `char`: character values (strings)
- `cell`: inhomogeneous arrays
- `struct`: structures
- `function handle`: references to functions

Given that by default any numerical value in MATLAB is a `double`, all values that are intended to be of a different numeric type have to be specifically converted. This also means that when combining a value of some non-double mclass with a value that is a `double`, the result will be of the non-double mclass. This leads to the surprising semantics that adding an `integer` and a `double` results in an `integer`, because that is the more specialized type.

```
>> x = 3; y = int8(5); % assign to x and y,   y  is   explicty  an  integer
>> class(x)            % the  class  of  x  is  double
    double
>> class(y)            % the  class  of  y  is  int8
    int8
>> class(x+y)          % the  result  of  x+y  is  int8,   not  double
    int8
```

2.3 MATLAB Functions and Specialization

A MATLAB function is defined in a `.m` file which has the same name as the function.[1] So, for example, a function named `foo` would be defined in a file `foo.m`, and that file needs to be placed either in the current directory, or in a directory on the MATLAB path. A `.m` file can also define subfunctions following the main (primary) function definition in a file, but those subfunctions are only visible to the functions within the file. Functions may also be defined in a `private/` directory, in which case they are visible only to functions defined in the parent directory.

MATLAB allows overriding or specializing operations and functions to operate on specific mclasses. This is accomplished by defining the function in a file inside a specially named directory which starts with the character `@` followed by the name of the mclass. For example, one could create a specialized function `firstWord` defined for Strings, by creating a file `@char/firstWord.m` somewhere on the MATLAB path. Functions that are specialized in such a way have precedence over non-specialized

[1] In the case where the name of the file and the function do not match, the name of the file takes precedence.

functions, but they do not have precedence over inner functions, subfunctions (defined in the same file) or private functions (defined in the `/private` directory). So, in our example, if there existed two definitions of `firstWord.m`, one general implementation somewhere on the MATLAB path, and one specialized implementation in a directory `@char` on the MATLAB path, then a call to `firstWord` with a `char` argument will result in a call to `@char/firstWord.m`, whereas a call with an argument with any other mclass, will result in a call to the general `firstWord.m` definition.

When calling a function that has mclass-specialized versions with multiple arguments of different mclasses, MATLAB has to resolve which version of the function to call. There doesn't exist a standard inheritance relationship between the builtin mclasses. Rather, MATLAB has the notion of a *superior* or *inferior* class. We were unable to find a succinct summary of these relationships, so we generated a MATLAB program which exercised all cases and which produced a `.dot` file describing all relationships, with all transitive relationships removed. Fig. 2 shows the relationships between different builtin mclasses, showing superior classes above inferior classes. Note that some mclasses have no defined relationship, for example, there are no defined inferior/superior relationships between the different integer mclasses. Further, note that `double`, being the default mclass, is inferior to integer mclasses. Also, the compound mclasses (struct and cell), are superior to all matrix mclasses.

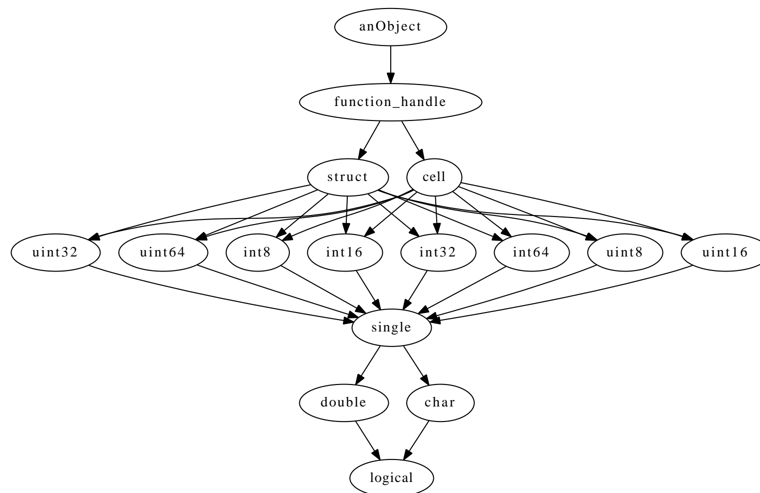

Figure 2: Superior/inferior class relationships for MATLAB

When resolving a call with multiple arguments, MATLAB finds the most superior argument, and uses its mclass to resolve the call. If multiple arguments have no defined superior/inferior relationships, MATLAB uses the leftmost superior argument. For example, if a function is called with three arguments with the mclasses (`double`, `int8`, `uint32`), in that order, MATLAB attempts to find a specialized version for mclass `int8`. If none is found, MATLAB attempts to find a non-specialized version.

The class specialization semantics for MATLAB means that if one intends to build a complete callgraph, i.e. resolve all possible call edges, one has to find all possible MATLAB classes for all arguments, and one must safely approximate the lookup semantics of functions, including the correct lookup of specialized functions using the mclass and the superior/inferior mclass relationships from Fig. 2.

2.4 MATLAB Classes

It is important to note that the mclass of a value does not completely define its type. For example, numeric MATLAB values may be real or complex, and matrices have an array shape. Both of these properties are defined orthogonally to the notion of its mclass. Although a computation can ask whether a value is complex or real, and can ask for the shape of an array, the lookup semantics solely depend on the mclass, which is effectively just a name. Within the MATLAB language, there is no dedicated class of values to represent mclasses. Usually, strings (char vectors) are used to denote mclasses. For example, `ones(3,2,'single')`, will call the builtin function 'ones' and create a 3×2 array of unit values of mclass `single`.

2.5 Function Handles

MATLAB values with mclass `function_handle` store a reference to a function. This allows passing functions as arguments to other functions. Function handles can either be created to refer to an existing function, or can be a lambda expression. Lambda expressions may also encapsulate state from the current workspace via free variables in the lambda expression.

```
>> f = @sin            % a function handle to a named function
f = @sin
>> g = @(x) exp(a*x)   % a lambda with a free variable "a"
g = @(x)exp(a*x)
```

Function handles, and especially lambdas, are useful in numerical computing, for example when calling numerical solvers, as illustrated below.

```
f = @(t,y) D*t + c; % set up derivative function
span = [0 1];      % set interval
y0 = [0:0.1:10]'; % set initial value
result = ode23s(f,span,y0); % use MATLAB library function to solve ODE
```

When building a callgraph of a program that includes function handles, one needs to propagate function handles through the program interprocedurally in order to find out which variables may refer to function handles, and to find associated call edges.

2.6 Function Parameters and Arguments

MATLAB uses call-by-value semantics, so that each parameter denotes a fresh copy of a variable.[2] This simplifies interprocedural analyses for static compilation as calling a function cannot directly modify local variables in the caller.

In MATLAB, function arguments are optional. That is, when calling a function one may provide fewer arguments than the function is declared with. However, MATLAB does not have a declarative way of specifying default values, nor does it automatically provide default values. That is, a parameter corresponding to an argument that was not provided will simply be unassigned and a runtime error will be thrown if an unassigned variable is read.

MATLAB does provide the function `nargin` to query how many arguments have been provided to the

[2] Actual MATLAB implementations only make copies where actually necessary, using either lazy copying when writing to an array with reference count greater than 1, or by using static analyses to determine where to insert copies [10].

currently executing function. This allows the programmer to use the value of nargin to explicitly assign values to the missing parameters, as illustrated below.

```
function [result1, result2] = myFunction(arg1,arg2)
   if (nargin < 1)
     arg1 = 0;
   end
   if (nargin < 2)
     arg2 = 1;
   end;
   ...
end
```

As shown above, MATLAB also supports assigning multiple return variables. A function call may request any number of return values simply by assigning the call into a vector of lvalues. Just like the function arguments, the return values don't all need to be assigned, and a runtime error is thrown if a requested return value is not assigned. MATLAB provides the `nargout` function to query how many results need to be returned.

Clearly a static compiler for MATLAB must deal with optional arguments in a sound fashion.

2.7 Wild Dynamic Features

Whereas features like dynamic typing, function handles, and variable numbers of input arguments are both widely used and possible to tame, there are other truly wild dynamic features in MATLAB that are not as heavily used, are sometimes abused, and are not amenable for static compilation.

These features include the use of scripts (instead of functions), arbitrary dynamic evaluation (`eval`), dynamic calls to functions using `feval`, deletion of workspace variables (`clear`), assigning variables at runtime in the caller scope of a function (`assignin`), changing the function lookup directories during runtime (`cd`) and certain introspective features. Some of these can destroy all available static information, even information associated with different function scopes.

Our approach to these features is to detect such features and help programmers to remove them via refactorings. Some refactorings can be automated. For example, McLAB already supports refactorings to convert scripts to functions and some calls to `feval` to direct function calls [15]. Other refactorings may need to be done by the programmer. For example, the programmer may use `cd` to change directory to access some data file, not being aware that this also changes the function lookup order. The solution in this case is to use a path to access the data file, and not to perform a dynamic call to `cd`. We have also observed many cases where dynamic `eval` or `feval` calls are used because the programmer was not aware of the correct direct syntax or programming feature to use.[3] For example, `feval` is often used to evaluate a function name passed as a String, where a more correct programming idiom would be to use a function handle.

2.8 Summary

In this section we have outlined key MATLAB features and semantics, especially concentrating on the definition of mclass and function lookup. Our approach is to tame as much of MATLAB as possible, including support for function pointers and lambda definitions. Capturing as much as possible of

[3] This is at least partly due to the fact that older versions of MATLAB did not support all of the modern features.

the evolved language is not just useful to allow access to a wider set of Matlab features for user code. Also, a significant portion of Matlab's extensive libraries are written in Matlab itself, and make extensive use of some of the features discussed above. Since we implement the Matlab lookup semantics, and allow the inclusion of the Matlab path, our callgraph will automatically include available Matlab library functions. Thus, implementing more features will also benefit users who do not make direct use of advanced features.

3 Framework for Builtins

One of the strengths of Matlab is in its large library, which doesn't only provide access to a large number of matrix computation functions, but packages for other scientific fields. Even relatively simple programs tend to use a fair number of library functions. Many library functions are actually implemented in Matlab code. Thus, to provide their functionality, the callgraph construction needs to include any Matlab function on the Matlab path, if it is available. Thus we can provide access to a large number of library functions as long as we can support the language features they use. However, hundreds of Matlab functions, termed builtin functions, are actually implemented in native code. We call these functions builtins or builtin functions. Every Matlab operator (such as +, ∗) is actually a builtin function; the operations are merely syntactic sugar for calling the functions that represent the operations (like 'plus', 'mtimes'). Thus, for an accurate static analysis of Matlab programs one requires an accurate model of the builtins. In this section we describe how we have modeled the builtins and how we integrate the analysis into the static interprocedural analysis framework.

3.1 Learning about Builtins

As a first step to build a framework of builtin functions, we need to identify builtins, and need to find out about their behavior, especially with respect to mclasses.

3.1.1 Identifying Builtins:

To make the task of building a framework for builtins manageable, we wanted to identify the most commonly used builtin functions and organize those into a framework. Other builtins can be added incrementally, but this initial set was useful to find a good structure.

To identify commonly used builtins we used the McBench framework [15] to find all references to functions that occur in a large corpus of over 3000 Matlab programs.[4] We recorded the frequency of use for every function and then using the Matlab function `exist`, which returns whether a name is a variable, user-defined function or builtin, we identified which one of these functions is a builtin function. This provided us with a list of builtin functions used in real Matlab programs, with their associated frequency of use. We selected approximately three hundred of the most frequent functions, excluding very dynamic functions like `eval` as our initial set of builtin functions.[5]

[4] This is the same set of projects that are used in [5]. The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing.

[5] The complete list can be found at `www.sable.mcgill.ca/mclab/tamer.html`

3.1.2 Finding Builtin Behaviours:

In order to build a call graph it is very important to be able to approximate the behaviour of builtins. More precisely, given the mclass of the input arguments, one needs to know a safe approximation of the mclass of the output arguments. This behaviour is actually quite complex, and since the behaviour of MATLAB 7 is the defacto specification of the behaviour we decided to take a programmatic approach to determining the the behaviours.

We developed a set of scripts that generated random MATLAB values of all combinations of builtin mclasses, and called selected builtins using these arguments. If different random values of the same mclass result in consistent resulting mclasses over many trials, the scripts record the associated mclass propagation for builtins in a table, and collect functions with the same mclass propagation tables together. Examples of three such tables are given in Fig. 3.[6]

	i8	i16	i32	i64	f32	f64	c	b
i8	i8	-	-	-	-	i8	i8	-
i16	-	i16	-	-	-	i16	i16	-
i32	-	-	i32	-	-	i32	i32	-
i64	-	-	-	i64	-	i64	i64	-
f32	-	-	-	-	f32	f32	f32	f32
f64	i8	i16	i32	i64	f32	f64	f64	f64
c	i8	i16	i32	i64	f32	f64	f64	f64
b	-	-	-	-	f32	f64	f64	f64

(a) `plus`, `minus`, `mtimes`, `times`, `kron`

	i8	i16	i32	i64	f32	f64	c	b
i8	i8	-	-	-	-	-	i8	-
i16	-	i16	-	-	-	-	i16	-
i32	-	-	i32	-	-	-	i32	-
i64	-	-	-	i64	-	-	i64	-
f32	-	-	-	-	f32	-	f32	f32
f64	i8	i16	i32	i64	f32	f64	f64	f64
c	i8	i16	i32	i64	f32	f64	f64	f64
b	-	-	-	-	f32	f64	f64	-

(b) `mpower`, `power`

	i8	i16	i32	i64	f32	f64	c	b
i8	i8	-	-	-	-	i8	i8	-
i16	-	i16	-	-	-	i16	i16	-
i32	-	-	i32	-	-	i32	i32	-
i64	-	-	-	i64	-	i64	i64	-
f32	-	-	-	-	f32	f32	f32	f32
f64	i8	i16	i32	i64	f32	f64	f64	f64
c	i8	i16	i32	i64	f32	f64	f64	f64
b	-	-	-	-	f32	f64	f64	-

(c) `mldivide`, `mrdivide`, `ldivide`, `rdivide`, `mod`, `rem`, `mod`

Figure 3: Example mclass results for groups of Built-in binary operators. Rows correspond to the mclass of the left operand, columns correspond to the mclass of the right operand, and the table entries give the mclass of the result. The labels i8 to i64 represent the classes int8 through int64, f32 is single, f64 is double, c is char, and b is logical. Entries of the form "-" indicate that this combination is not allowed and will result in a runtime error.

As compared with type rules in other languages, these results may seem a bit strange. For example, the "-" entry for `plus(int16,int32)` in Fig. 3(a) shows that it is an error to add an int16 to and int32. However adding an int64 to a double is allowed and it results in an int64. Also, note that although the three tables in Fig. 3 are similar, they are not identical. For example, in Fig. 3(a), multiplying

[6] To save space we have not included the whole table, we have left out the columns and rows for unsigned integer mclasses and for handles. All result tables can be found at `www.sable.mcgill.ca/mclab/tamer.html`

a logical with a logical results in a double, but using the power operator with two logicals is an error. Finally, note that the tables are not always symmetrical. In particular, the `f64` column and row in Fig. 3(b) are not the same.

The reader may have noticed how the superior/inferior m-class relationships as shown in figure Fig. 2 seem to resemble the implicit type conversion rules for MATLAB builtin functions. For example, when adding an integer and a double, the result will be double. However, it is not sufficient to model the implicit MATLAB class conversion semantics by just using class-specialized functions and their relationships. Many MATLAB builtins perform explicit checks on the actual runtime types and shapes of the arguments and perform different computations or raise errors based on those checks.

Through the collection of a large number of tables we found that many builtins have similar high-level behaviour. We found that some functions work on any matrix, some work on numeric data, some only work on floats, and some work on arbitrary builtin values, including cell arrays or function handles.

3.2 Specifying Builtins

To capture the regularities in the builtin behaviour we arranged all of the builtins in a hierarchy - a part of the hierarchy is given in Fig. 4. Leaves of the hierarchy correspond to actual builtins and upper levels correspond to abstract groups which share some sort of similar behaviour. The motivation is that some flow analyses need only specify the abstract behaviour of a group, and the flow analysis framework will automatically apply the correct (most specialized) behaviour for a specific builtin.

To specify builtins and their relationships, we developed a simple domain-specific language. One just needs to specify the name of a builtin. If the builtin is abstract (i.e. it refers to a group of builtins), the parent group has to be specifies. If no parent is specified, the specified name is an actual builtin, belonging to the group of the most recently specified builtin. This leads to a very compact representation, allowing builtins to be specified on one line each, as illustrated by the following snippet of the specification.

```
floatFunction; matrixFunction
properFloatFunction; floatFunction
unaryFloatFunction; properFloatFunction
elementalUnaryFloatFunction; unaryFloatFunction
sqrt
realsqrt
erf
...
improperFloatFunction; floatFunction
...
```

The builtin framework takes a specification like above, and generates a set of Java classes, whose inheritance relationship reflects the specified tree. It also generates a visitor class, which allows annotating methods to Builtins using the visitor pattern - a pattern that is already extensively used in the MCSAF framework [4].

We categorize the MATLAB builtin functions according to many properties, such as mclass, arity, shape, semantics. This means that different analyses or attributes can be specify at exactly the required category. It also means that when adding builtins that do fit in already existing categories,

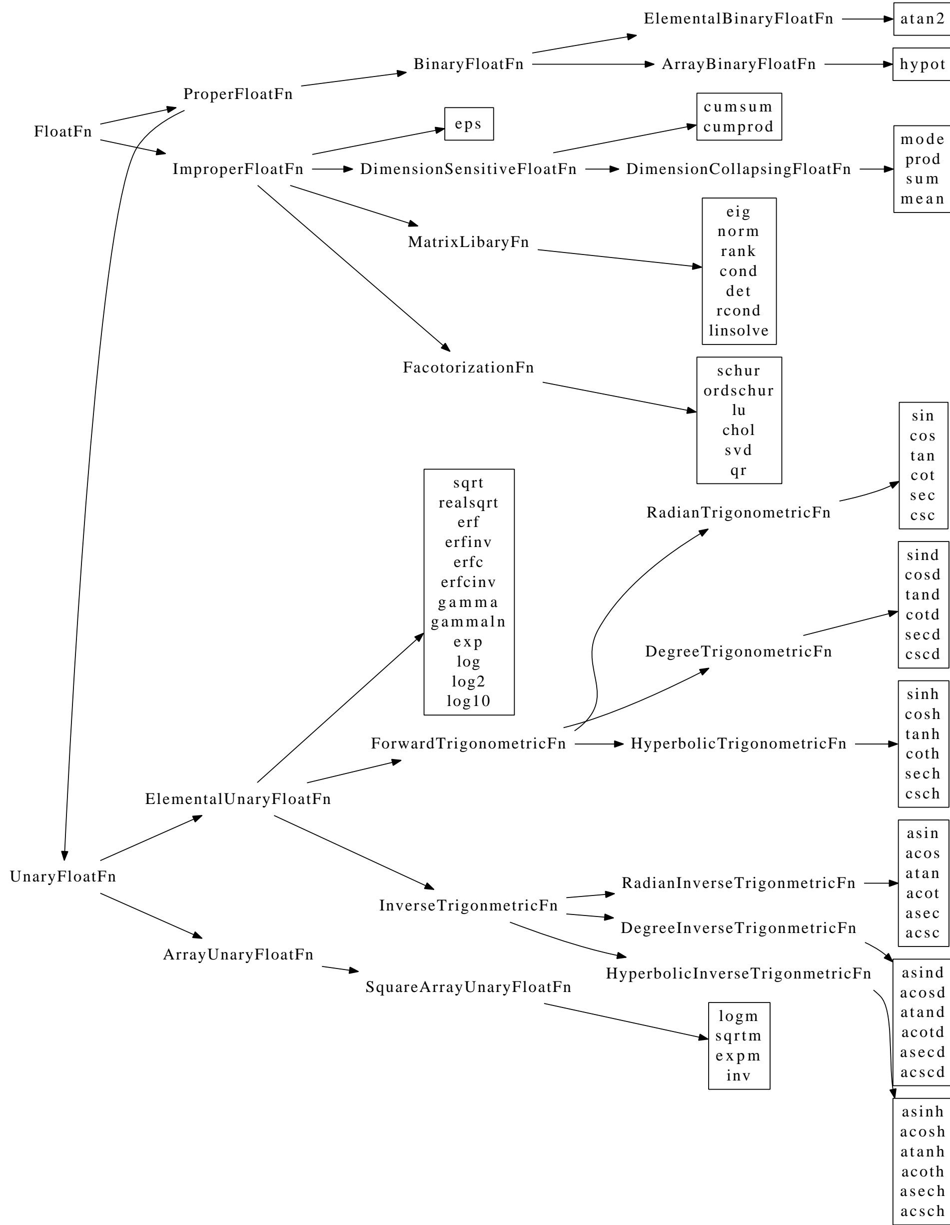

Figure 4: Subtree of builtin tree, showing all defined floating point builtins of Matlab

one does not need to add all required attributes or flow equations.

3.3 Specifying Builtin attributes

It is not sufficient to just specify the existence of builtins; their behavior needs to be specified as well. In particular, we need flow equations for the propagation of mclasses. Thus the builtin specification language allows the addition of properties. A property is just a name, with a set of arguments that follow it. A specific property can be defined for any builtin, and it will trigger the addition of more methods in the generated Java code as well as the inclusion of interfaces. In this

way, any property defined for an abstract builtin group is defined for any builtin inside that group as well, unless it gets overridden.

The first property we defined was the property `Class`. When specified for a builtin, it forces the inclusion of the Java interface `ClassPropagationDefined` in the generated Java code, and will add a method that returns an mclass flow equation. The mclass flow equation itself is specified as an argument to the `Class` attribute using a small domain specific language that allows matching argument mclasses, and returns result mclasses based on matches. An example snippet is given below which shows the specification of m-lass flow equations for unary functions taking numeric arguments. Functions in that group accept any numeric argument and return a result of the same mclass (`numeric>0`), a char or logical argument will result in a `double`.

```
unaryNumericFunction; properNumericFunction;; Class(numeric>0, char|logical>double)

elementalUnaryNumericFunction; unaryNumericFunction; abstract
real
imag
abs
conj;; MatlabClass(logical>error,natlab)
sign;; MatlabClass(logical>error,natlab)
```

We have noticed some irregularities in the pure Matlab semantics, and our specification sometimes removes those. In order to keep a record of the differences we use the `MatlabClass` specification which allows us to specify the exact Matlab semantics - and thus provides an exact definition and documentation of Matlab class semantics. In the example above, we specify that the functions `conj` and `sign` have different Matlab semantics: they disallow `logical` arguments, which will result in an error.

3.4 Summary

We have performed an extensive analysis of the behavior of Matlab builtin functions. Based on that we developed a framework that allows to specify Matlab builtin functions, their relationships and properties such as flow equations in a compact way. This framework is extensible both by allowing the quick addition of more builtin functions; and by allowing to specify information and behavior for builtin functions. This can be done either adding new properties to the framework itself; or by implementing visitor classes. [7]

4 Tame IR

As indicated in Fig. 1, we build upon the McSaf framework by adding taming transformations and by producing a more specialized Tame IR. To produce an easily analyzable Tame IR we have made three important additions: (1) generating more specialized AST nodes, (2) translating `switch` statements to equivalent conditional statements, and (3) transforming lambda expressions to analyzable equivalents. One might wonder why these transformations are not already part of McSaf. The important point is that McSaf must handle all of Matlab, whereas for our Tame

[7]The complete specification of builtins, documentation of the specification and diagrams of all builtins is available at www.sable.mcgill.ca/mclab/tamer.html

IR we can make restrictions that are reasonable for the purposes of static compilation. This allows us to make the Tame IR more specialized and enables more simplifying transformations.

4.1 Specialized AST nodes

One goal for our Taming framework was to produce an IR that is very simple to analyze, and has operations that are low-level enough to map fairly naturally to static languages like FORTRAN. As one example, in McSAF there is only one kind of assignment statement, assigning from an expression to an lvalue expression. For the Tame IR, we have many more specialized cases as illustrated in Fig. 5. We also extended McSAF's analysis framework to recognize these new IR nodes, so flow equations can be specified for all these new nodes. Note how the Tame IR has a different statement for a function call or an array indexing operation. In MATLAB these use the same syntax, a parameterized expression. We use kind analysis to resolve names to being a function or variable, but there are rare cases when this is not possible. Tame MATLAB will reject these cases.

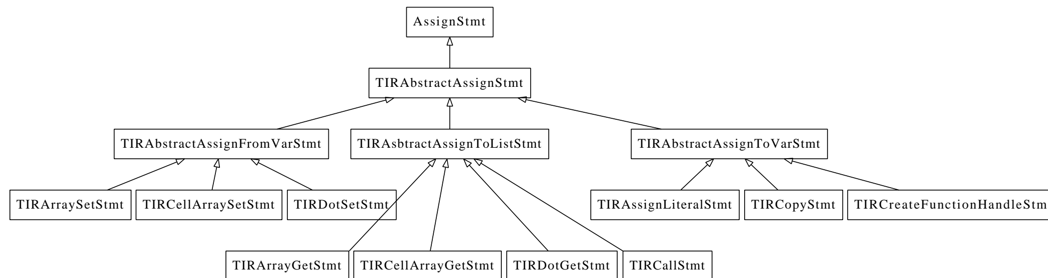

Figure 5: Specializations of an assignment statement

4.2 Lambda Simplification

MATLAB supports lambda expressions. In order to be compatible with the Tame IR, their bodies need to be converted to a three address form in some way. MATLAB lambda expressions are just a single expression (rather than, say, statement lists), so we extract the body of the lambda expression into an external function. The lambda expression still remains, but will encapsulate only a single call, all whose arguments are variables. For example, the lambda simplification will transform the expression in Fig. 6(a) to the code in Fig. 6(b). The new lambda expression encapsulates a call to the new function `lambda1`. Note that the first two arguments are variables from the workspace, the remaining ones are the parameters of the lambda expression. In the analyses, we can thus model the lambda expression using partial evaluation of the function `lambda1`. To make this transformation work, the generated function must return exactly one value, and thus Tame MATLAB makes the restriction that lambda expressions return a single value (of course that value may be an array, struct or cell array).

4.3 Switch simplification

As illustrated in Fig. 7(a), MATLAB has support for very flexible switch statements. Unlike in other languages, all case blocks have implicit breaks at the end. In order to specify multiple case

```
function outer
  ...
  f = @(t,y) D*t + c
  ...
end
```

(a) lambda

```
function outer
  ...
  f = @(t,y) lambda1(D,c,t,y)
  ...
end

function r = lambda1(D,c,t,y)
  r = D*t + c
end
```

(b) transformed lambda

Figure 6: Transforming `lambda` expressions

comparisons for the same case block, Matlab allows using cell arrays of case expressions, for example `{2, 3}` in Fig. 7(a). Indeed, Matlab allows arbitrary case expressions, such as `c` in the example. If `c` refers to a cell array, then the case will match if any element of the cell array matches. Without knowing the static type and size of the case expressions, a simplification transformation is not possible. Thus, to enable the static simplification shown in Fig. 7(b) we add the constraint for the Tame Matlab that case-expressions are only allowed to be syntactic cell arrays.

```
switch n
  case 1
    ...
  case {2, 3}
    ...
  case c
    ...
  otherwise
    ...
end
```

(a) switch

```
t = n
if (isequal(t,1))
    ...
elseif (isequal(t,2) || isequal(t,3))
    ...
elseif (isequal(t,c))
    ...
else
    ...
end
```

(b) transformed switch

Figure 7: Transforming `switch` statements

5 Interprocedural Value Analysis and Call Graph Construction

The core of the Matlab Tamer is the *value analysis*. It's an extensible monolithic context-sensitive inter-procedural forward propagation of abstract Matlab values. For every program point, it estimates what possible values every variable can take on. Most notably it finds the possible set of mclasses. It also propagates function handle values. This allows resolution of all possible call edges, and the construction of a complete call graph of a tame Matlab program.

The value analysis is part of an extensible interprocedural analysis framework. It contains a set of modules, one building on top of the other. All of them can be used by users of the framework to build analyses.

- The **interprocedural analysis framework** (section 5.1) builds on top of the Tame IR and the McSaf intraprocedural analysis framework. It allows the construction of interprocedural analyses by extending an intraprocedural analysis built using the McSaf framework. This framework works together with a callgraph object implementing the correct Matlab look up semantics. An analysis can be run on an existing callgraph object, or it can be used to build new callgraph objects, discovering new functions as the analysis runs.

- The **abstract value analysis** (section 5.2), built using the interprocedural analysis framework, is a generic analysis of abstract Matlab values. The implementation is agnostic to the actual representation of abstract values, but is aware of Matlab mclasses. It can thus build a callgraph using the correct function lookup semantics including function specialization.

- We provide an implementation of **composite values** like cell arrays, structures and function handles, which is generic in the implementation of abstract matrix values (section 5.4). This makes composite values completely transparent, allowing users to implement very fine-grained abstract value analyses by only providing an abstraction for Matlab values which are matrices.

- Building on top of all the above modules and putting everything together, we provide an abstraction for all Matlab values, which we call simple values (section 5.5). Since it includes the function handle abstractions, this can be used by users to build a complete tame Matlab callgraph. This is the **concrete value analysis**, whose results are presented in section 5.6.

5.1 The Interprocedural Analysis Framework

The interprocedural Analysis framework is an extension of the intraprocedural flow analyses provided by the McSaf framework. It is context-sensitive to aid code generation targeting static languages like Fortran. Fortran's polymorphism features are quite limited; every generated variable needs to have one specific type. The backend may thus require that every Matlab variable has a specific known mclass at every program point. Functions may need to be specialized for different kinds of arguments, which a context-sensitive analysis provides at the analysis level.

An interprocedural analysis is a collection of interprocedural analysis nodes, which represent a specific intraprocedural analysis for some function and some context. The context is usually a flow representation of the passed arguments. Every such interprocedural analysis node produces a result set using the contained intraprocedural analysis.

Every interprocedural analysis has an associated callgraph object, which may initially contain only one function acting as the entry point for the program. The interprocedural analysis requires a context or argument set for the entry point function.

The analysis starts by creating an interprocedural analysis node for the entry point function and the associated context, which triggers the associated intraprocedural flow analysis. As the intraprocedural flow analysis encounters calls to other functions, it has to create context objects for those calls, and ask the interprocedural analysis to analyze the called functions using the given context. The call also gets added to the set of call edges associated with the interprocedural analysis node.

As the interprocedural has to analyze newly encountered calls, the associated functions are resolved, and loaded into the callgraph if necessary. The result is a complete callgraph, and an interprocedural analysis.

The interprocedural analysis framework supports simple and mutual recursion by performing a fixed point iteration within the first recursive interprocedural analysis node.

5.2 Introducing the Value Analysis

The abstract value analysis is a forward propagation of generic abstract MATLAB values. The mclass of any abstract value is always known.

A specific instance of a value analysis may use different representations for values of different mclasses. For example, function handle values may be represented in a different way than numeric values. This in turn means that values of different Matlab classes can not be merged (joined).

5.2.1 Mclasses, Values and Value Sets:

To define the value analysis independently of a specific representation of values, We first define the set of all mclasses:

$$C = \{\texttt{double}, \texttt{single}, \texttt{logical}, \texttt{cell}, \ldots\}$$

For each mclass, we need some lattice of values that represent estimations of MATLAB values of that class:

$$V_{mclass} = \{v : v \text{ represents a MATLAB value with mclass } mclass\}, mclass \in C$$

We require that merge operations are defined, so $\forall v_1, v_2 \in V_{mclass}, v_1 \wedge v_2 \in V_{mclass}$.

We can not join values of different mclasses, because their actual representation may be incompatible. In order to allow union values for variables, i.e. to allow variables to have more than one possible mclass, we estimate the value of a MATLAB variable as a set of pairs of abstract values and their mclasses, where the mclasses are disjoint. We call this a value set. More formally, we define a value set as:

$$\begin{aligned} ValueSet = \{&(mclass_1, v_1), (mclass_2, v_2), \ldots, (mclass_n, v_n) : \\ &class_i \neq class_j, class_i \in C, v_i \in V_{class_i}\} \end{aligned}$$

Or the set of all possible value sets given a set V of lattices for every mclass.

$$S_V = \{\{(mclass_k, v_k) : mclass_i \neq mclass_j, v_i \in V_{mclass_i}, k \in 0..n\} : 0 \leq n \leq |C|\}\}$$

This is a lattice, with the join operation which is the simple set union of all the pairs, but for any two pairs with matching mclasses, their values get joined, resulting in only one pair in the result set.

While the notion of a value set allows the analysis to deal with ambiguous variables, still building a complete callgraph and giving a valid estimation of types, having ambiguous variables is not conducive to code generation for a language like FORTRAN. So

```
if (...); t = 4; else; t = 'hi'; end
```

results in t having the abstract value $\{(\texttt{double}, 4), (\texttt{char}, \texttt{'hi'})\}$. This example is not tame MATLAB.

5.2.2 Flow Sets:

We define a flow set as a set of pairs of variables and value sets, i.e.

$$flow = \{(var_1, s_1), (var_2, s_2), ..., (var_n, s_n) : s_i \in S_V, var_i \neq var_j\}$$

and we define an associated look-up operation

$$flow(var) = s \text{ if } (var, s) \in flow$$

This is a lattice whose merge operation resembles that of the value sets.

Flow sets may be *nonviable*, representing non-reachable code (for statements after errors, or non-viable branches). Joining any non-*bottom* flow set with the *nonviable* set results in the viable flow set. joining *bottom* and *nonviable* results in *nonviable*.

5.2.3 Argument and Return sets:

The context or argument set for the interprocedural analysis is a vector of values representing argument values. Arguments are not value sets, but simple values $v \in V_c$ with a single known mclass c. When encountering a call, the analysis has to construct all combinations of possible argument sets, construct a context from that and analyze the call for all such contexts. For example, if we reach a call `r = foo(a,b)`, with a flow set

$$\{(a, \{(\texttt{double}, v_1), (\texttt{char}, v_2)\}, (b, \{(\texttt{logical}, v_3)\})\},$$

the value analysis constructs two contexts, from (v_1, v_3) and (v_2, v_3), and analyzes function `foo` with each context. Note how the dominant argument for the first context is `double`, whereas it is `char` for the second. If there exist mclass specialized versions for `foo`, then this results in call edges to, and analysis of, two different functions.

More formally, for a call $func(a_1,\ a_2,\ \cdots\ ,\ a_n)$ at program point p, with the input flow set f_p, we have the set of all possible contexts

$$allargs = f_p(a_1) \times f_p(a_2) \times \cdots \times f_p(a_n) = \prod_{1 \leq i \leq n} f_p(a_i)$$

the interprocedural analysis needs to analyze $func$ with all these contexts and merge the result,

$$R = \bigwedge_{arg \in allargs} analyze(func, arg)$$

To construct a context, the value analysis may simplify (push up) values to a more general representation. For example, if the value abstraction includes constants, the push up operation may turn constants into *top*. Otherwise, the number of contexts for any given function may grow unnecessarily large.

The result of analyzing a function with an argument set is a vector of value sets, where every component represents a returned variable. They are joined by component-wise joining of the value sets. In the value analysis we require that for a particular call, the number of returned variables is the same for all possible contexts.

5.2.4 Builtin Propagators:

Every implementation of the value abstractions needs to provide a builtin propagator, which provides flow equations for builtins. If B is the set of all defined builtin functions $\{\texttt{plus}, \texttt{minus}, \texttt{sin}, \ldots\}$, then the builtin propagator P_V for some representation of values V_C is a function mapping a builtin and argument set to a result set.

$$P_V : B \times \bigcup_{n \in \mathbb{N}} V^n \rightarrow \bigcup_{n \in \mathbb{N}} (S_V)^n$$

The builtin framework provides tools to help implement builtin propagators by providing builtin visitor classes. The framework also provides attributes for builtin functions, for example the class propagation information attributes.

5.3 Flow Equations

In the following subsection we will show a sample of flow equations to illustrate the flow analysis. We assume a statement to be at program point p, with incoming flow set f_p. The flow equation for program point p results in the new flow set f'_p

- var_t `=` var_s: $\quad f'_p = f_p \setminus \{(var_t, f_p(var_t))\} \cup \{(var, f_p(var_s))\}$
- var `=` l, where l is a literal with mclass c_l and value representation v_l:
  $$f'_p = f_p \setminus \{(var, f_p(var))\} \cup \{(var, \{(c_l, v_l)\})\}$$
- `[`t_1`,`t_2`,...,`t_m`]` `=` $func$`(`a_1`,`a_2`,...,`a_n`)`, a function call to some function $func$:
  with
  $$call_{func,arg} = \begin{cases} P_V(b, args) & \text{if } func \text{ with } args \text{ refers to a builtin } b \\ analyze(f, args) & \text{if } func \text{ with } args \text{ refers to a function } f \end{cases}$$
  we set
  $$R = \bigwedge_{args \in f_p(a_1) \times f_p(a_2) \times \cdots \times f_p(a_n)} call_{func,args}$$
  then
  $$f'_p = f_p \setminus \bigcup_{i=1}^{m} \{(t_i, f_p(t_i))\} \cup \bigcup_{i=1}^{m} \{(t_i, R_i)\}$$

Note that when analyzing a call to a function in an m-file, the argument values will be pushed up. For calls to builtins, the actual argument values will be used, effectively in-lining the behaviour of builtin functions.

5.4 Structures, Cell Arrays and Function Handles

We implemented a value abstraction for structs, cell arrays and function handles. This abstraction is again modular, this one with respect to the representation of matrix values (i.e. values with mclass `double`, `single`, `char`, `logical` or integer). Structures, cell arrays and function handles act as containers for other values, making them effectively transparent. A user may provide a fine-grained abstraction for just matrix values and combine it with abstraction of composite values to implement a concrete value analysis.

5.4.1 `struct`, `cell`:

For structures and cell arrays, there are two possible abstractions:

- *tuple*: The exact index set of the `struct/cell` is known and every indexing operation can be completely resolved statically. Then the value is represented as a set of pairs $\{(i_1, s_1), (i_2, s_2), .., (i_n, s_n) : i_k \in I, s_n \in S_V\}$, where I is an index set - integer vectors for cell arrays, and names for structs.

- *collection*: Not all indexing operations can be statically resolved, or the set of indices is unknown. In this case, all value sets contained in the struct or cell are merged together, and the representation is a single value set $s \in S_V$.

The usual representation for a structure is a tuple, because usually all accesses (dot-expressions) are explicit in the code and known. Cell arrays are usually a collection, because the index expressions are usually not constant. But cell arrays tend to have homogeneous mclass values, so there is some expectation that any access of a `struct` or `cell` results in some unambiguous mclass and thus allows static compilation.

5.4.2 `function_handle`:

As explained in section 2.5, function handles can be created either by referring to an existing function, or by using a lambda expression to generate an anonymous function using a lambda expression. The lambda simplification (presented in section 4.2) reduces lambda expressions to single calls.

We model all function handles as sets of function handle pairs. A function handle pair consists of a reference to a function and a vector of partial argument value sets. A function handle value may thus refer to multiple possible function/partial argument pairs.

Given some flow set f_p defined at the program point p,

`g = @sin` results in

$$f'_p = f_p \setminus (g, f_p(g)) \cup \{(g, \{(\texttt{function_handle}, \{(\texttt{sin}, ())\})\})\}$$

`g = @(t,y) lambda1(D,c,t,y)` results in

$$f'_p = f_p \setminus (g, f_p(g)) \cup \{(g, \{(\texttt{function_handle}, \{(\texttt{lambda1}, (f_p(D), f_p(c)))\})\})\}$$

Note that function handles get invoked at array get statements, rather than calls. That is because the tame IR is constructed without mclass information, and will correctly interpret a function handle as a variable. When the target of an array get statement is a function handle, the analysis inserts one or more call edges at that program point, referring to the functions contained in the function handle.

5.5 The Simple Matrix Abstraction

Using the value abstraction for structures, cell arrays and function, we implemented a concrete value abstraction by adding an abstraction for matrix values, which we call simple matrix values. On top of the required mclass, this abstraction merely adds constant propagation for scalar doubles, strings (char vectors), and scalar logicals.

This allows the analysis of MATLAB code utilizing optional function arguments using the builtin function `nargin`, and some limited dynamic features utilizing strings. For example, a call like `ones(n,m,'int8')` can be considered tame.

This implementation represents the concrete value analysis that is used to construct complete callgraphs.

5.6 Applying the Value Analysis

In order to exercise the framework, we applied it to the set of benchmarks we have previously used for evaluating McVM/McJIT [10], a dynamic system. The benchmarks and results are given in Table I. About half of the benchmarks come from the FALCON project [16] and are purely array-based computations. The other half of the benchmarks were collected by the McLAB team and cover a broader set of applications and use more language features such as lambda expressions, cell arrays and recursion. The columns labeled #Fn correspond to the number of user functions, and the column labeled #BFn corresponds to the number of builtin functions used by the benchmark. Note the high number of builtins. The column labeled "Wild" indicates if our system rejected the program as too wild. Only the sdku benchmark was rejected because it used the `load` library function which loads arbitrary variables from a stored file. It is likely that we should provide a tamer version of load. The column labeled "Mclass" indicates "unique" if the interprocedural value propagation found a unique mclass for every variable in the program. Only three benchmarks had one or more variables with multiple different mclasses. We verified that it was really the case that a variable had two different possible classes in those three cases.

Name	Description	Source	#Fn	#BFn	Features	Wild	Mclass
adpt	*Adaptive quadrature*	Numerical Methods	1	17		no	unique
beul	*Backward Eurler*	McLAB	11	30	lambda	no	unique
capr	*Capacitance*	Chalmers EEK 170	4	12		no	unique
clos	*Transitive Closure*	Otter	1	10		no	unique
crni	*Tridiagonal Solver*	Numerical Methods	2	14		no	unique
dich	*Dirichlet Solver*	Numerical Methods	1	14		no	unique
diff	*Light Diffraction*	Appelbaum (MUC)	1	13		no	unique
edit	*Edit Distance*	Castro (MUC)	1	6		no	unique
fdtd	*Finite Distance Time Domain*	Chalmers EEK 170	1	8		no	unique
fft	*Fast Fourier Transform*	Numerical Recipes	1	13		no	**multi**
fiff	*Finite Difference*	Numerical Methods	1	8		no	unique
mbrt	*Mandelbrot Set*	McLAB	2	12		no	unique
mils	*Mixed Integer Least Squares*	Chang and Zhou	6	35		no	unique
nb1d	*1-D Nbody*	Otter	2	9		no	unique
nb3d	*3-D Nbody*	Otter	2	12		no	unique
nfrc	*Newton Fractal*	McLAB	4	16		no	unique
nne	*Neural Net*	McLAB	3	16	cell	no	unique
play	*Minimax Search*	McLAB	5	26	recursive, cell	no	**multi**
rayt	*Raytracer*	Aalborg (Jensen)	2	28		no	unique
sch2	*Sparse Schroed. Eqn Solver*	McLAB	8	32	cell, lambda	no	unique
schr	*Schroedinger Eqn Solver*	McLAB	8	31	cell, lambda	no	unique
sdku	*Sodoku Puzzle Solver*	McLAB	8		load	**yes**	
sga	*Vectorized Genetic Algorithm*	Burjorjee	4	30		no	**multi**
svd	*SVD Factorization*	McLAB	11	26		no	unique

Table I: Results of Running Value Analysis

Although the main point of this experiment was just to exercise the framework, we were very encouraged by the number of benchmarks that were not wild and the overall accuracy of the basic interprocedural value analysis. We expect many other analyses to be built using the framework,

with different abstractions. By implementing them all in a common framework we will be be able to compare the different approaches.

6 Related Work

There are several categories of related work. First, we have the immediate work upon which we are building. The McLab project already provided the front-end and the McSaf [4] analysis framework, which provided an important basis for the Tamer. We also learned a lot from McLab's previous McFor project [11] which was a first prototype Matlab to Fortran95 compiler. McFor supported a smaller subset of the language, did not have a comprehensive approach to the builtin functions, and had a much more ad hoc approach to the analyses. However, it really showed that conversion of Matlab to Fortran95 was possible, and that Fortran95 is an excellent target language. In this paper we have gone back to the basics and defined a much larger subset of Matlab, taken a more structured and extensible approach to building a general toolkit, tackled the problem of a principled approach to the builtins, and defined the interprocedural analyses in a more rigorous and extensible fashion. The next generation of McFor can now be built upon these new foundations.

Although we were not able to find publicly available versions, there have been several excellent previous research projects on static compilation of Matlab which focused particularly on the array-based subset of Matlab and developed advanced static analyses for determining shapes and sizes of arrays. For example, FALCON [16] is a Matlab to Fortran90 translator with sophisticated type inference algorithms. Our Tamer is targeting a larger and more modern set of Matlab that includes other types of data structures such as cell arrays and structs, function handles and lambda expressions, and which obeys the modern semantics of Matlab 7. We should note that FALCON handled interprocedural issues by fully inlining all of the the code. MaJIC [2], a MATLAB Just-In-Time compiler, is patterned after FALCON. It uses similar type inference techniques to FALCON, but are simpler to fit the JIT context. MAGICA [8, 9] is a type inference engine developed by Joisha and Banerjee of Northwestern University, and is written in Mathematica and is designed as an add-on module used by MAT2C compiler [7]. We hope to learn from the advanced type inference approaches in these projects and to implement similar approximations using our interprocedural value analysis.

There are also commercial compilers, which are not publicly available, and for which there are no research articles. One such product is the *MATLAB Coder* recently released by MathWorks [12]. This product produces C code for a subset of Matlab. According to our preliminary tests, this product does not appear to support cell arrays except in very specific circumstances, nor does it support a general form of lambda expressions, and was therefore unable to handle quite a few of our benchmarks. However, the key differences with our work is that we are designing and providing an extensible and open source toolkit for compiler and tool researchers. This is clearly not the main goal of proprietary compilers.

There are other projects providing open source implementations of Matlab-like languages, such as Octave [1] and Scilab [6]. Although these add valuable contributions to the open source community, their focus is on providing interpreters and open library support and they have not tackled the problems of static compilation. Thus, we believe that our contributions are complementary.

Other dynamic languages have had very successful efforts in defining compilable subsets. For

example RPython [3] uses a similar approach to ours, defining a reduced set of python that can be statically compiled, while providing an analysis and compiler to compile that subset.

7 Conclusions and Future Work

This paper has introduced the MATLAB Tamer, an extensible object-oriented framework for supporting the translation from dynamic MATLAB programs to a Tame IR, call graph and class/type information suitable for generating static code. We provided an introduction to the features of MATLAB in a form that we believe helps expose the semantics of mclasses and function lookup for compiler and tool writers. We tackled the somewhat daunting problem of handling the large number of builtin functions in MATLAB by defining an extensible hierarchy of builtins and a small domain-specific language to define their behaviour. We defined a Tame IR and added functionality to McSAF to produce the IR and to extend the analysis framework to handle to new IR nodes introduced. Finally, we developed an extensible interprocedural analysis framework and an extensible value analysis that we used for estimating the mclass of every variable.

Our initial experiments with the framework are very encouraging and we are now working on using the framework to implement back-ends, and we hope that others will also use the framework for a variety of static MATLAB tools.[8] We also plan to continue developing the value analysis to add richer abstractions for shape and other data structure properties. Finally, as a part of a larger project on benchmarking MATLAB, we hope to expand our set of benchmarks and to further examine which features might be tamed, and to extend our set of automated refactorings.

References

[1] GNU Octave. `http://www.gnu.org/software/octave/index.html`.

[2] G. Almási and D. Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 294–303, New York, NY, USA, 2002. ACM.

[3] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. Rpython: a step towards reconciling dynamically and statically typed oo languages. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64, New York, NY, USA, 2007. ACM.

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

[5] J. Doherty, L. Hendren, and S. Radpour. Kind analysis for MATLAB. In *In Proceedings of OOPSLA 2011*, pages 99–118, 2011.

[6] INRIA. Scilab, 2009. `http://www.scilab.org/platform/`.

[7] P. G. Joisha. a MATLAB-to-C translator, 2003.

[8] The URL for a distribution will be released with the final paper.

[8] P. G. Joisha and P. Banerjee. Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In *APL '01: Proceedings of the 2001 conference on APL*, pages 7–21, New York, NY, USA, 2001. ACM.

[9] P. G. Joisha and P. Banerjee. Static array storage optimization in MATLAB. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 258–268, New York, NY, USA, 2003. ACM.

[10] N. Lameed and L. J. Hendren. Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In *Proceedings of the International Compiler Conference (CC11)*, pages 22–41, 2011.

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

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

[13] C. Moler. The Growth of MATLAB and The MathWorks over Two Decades. `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf`.

[14] C. Moler. The Origins of MATLAB. `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/dec04.html`.

[15] S. Radpour. Understanding and Refactoring MATLAB. Master's thesis, McGill University, January 2012.

[16] L. D. Rose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.