MIX10: A MATLAB TO X10 COMPILER

by

Vineet Kumar

School of Computer Science

McGill University, Montréal

Thursday, October 31st 2013

Abstract

TBD

Résumé

TBD

Acknowledgements

I would like to thank my supervisor Laurie Hendren, for her support and direction. She also helped greatly to finish the paper that constitutes the core of this thesis.

I would also like to thank the entire *Mc*LAB team. In particular I would like to acknowledge contributions by colleagues Jesse Doherty (M.Sc.) whose MCSAF framework is the starting point for the Tamer framework presented in this thesis, as well as Soroush Radpour (M.Sc.), who provided the kind analysis (with Jesse) and help with the lookup semantics and implementation. His MCBENCH framework provided insights into the usage of MATLAB.

Finally, I would like to thank my friends and family for their continued support in light of delays, in particular my mother Dorothee and my girlfriend JC.

Table of Contents

Appendices

List of Figures

List of Tables

Chapter 1
Introduction

MATLAB is a popular numeric programming language, used by millions of scientists, engineers as well as students worldwide[Mol]. 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 proto-typing, their applications are often quite compute intensive and time consuming. These applications could perform much more efficiently if they could be easily ported to a high performance computing system.

On the other hand, X10 is an object-oriented and statically-typed language which uses cilk-style arrays indexed by *Point* objects, and has been designed with well-defined semantics and high performance computing in mind. X10 compiler can generate C++ or Java code and supports various communication interfaces including sockets and MPI for communication between nodes on a parallel computing system.

In this thesis we present MIX10, a source-to-source compiler that helps to bridge the gap between MATLAB, a language familiar to scientists, and X10, a language designed for high performance computing systems. MIX10 statically compiles MATLAB programs to X10 and thus allows scientists and engineers to write programs in MATLAB (or use old programs already written in MATLAB) and still get the benefits of high performance computing without having to learn a new language. Also, systems that use MATLAB for

1

prototyping and C++ or Java for production, can benefit from MIX10 by quickly converting MATLAB prototypes to C++ or Java programs via X10. *Figure 1.1* shows the flow of compilation from MATLAB to executable code via MIX10. In particular, this thesis identifies the key challenges in compiling a dynamically-typed language like MATLAB to a statically-typed object-orinted high performance computing language like X10 and our approach to compiling MATLAB to X10.
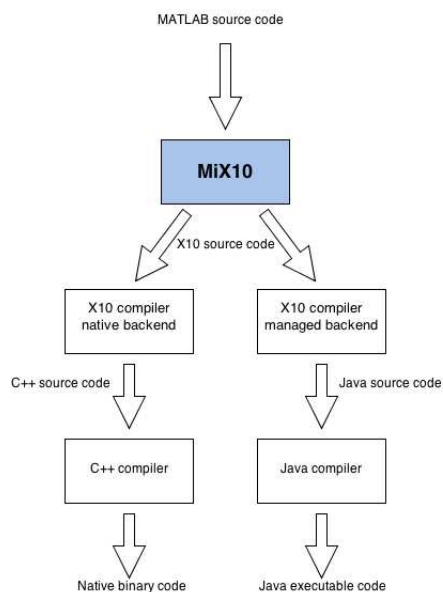
Figure 1.1 Compilation flow via MIX10

On one hand, all the aforementioned characteristics of MATLAB make it a very user-friendly and thus popular application to develop software among a non-programmer community. On the other hand, these same characteristics make MATLAB a difficult language to write a static compiler for. Lack of formal language specification, unconventional semantics and closed source make compiler writers' task even harder. Mathworks implementation of MATLAB is essentially an interpreter with a *JIT accelarator* which is generally slower than statically compiled languages. Built on top of *Mc**LAB*** frontend and static analysis tools, MIX10 provides static compilation for MATLAB via the C++ backend for X10 and thus ultimately aims to have better performance even with sequential code.

1.1 Contributions

The major contributions of this thesis are as follows:

Identifying key challenges: We have identified the key challenges in performing a semantics-preserving translation of MATLAB to X10.

Overall design of MIX10**:** We provide the design of a source-to-source translator, building upon the McLab front-end and analysis toolkits.

MIX10 **IR design:** In order to provide a convenient target for the first level of translation, we have defined a high-level MIX10 IR. This IR is used for code generation, X10 specific analyses, code simplifications and transformations.

Comparison of different kinds of X10 **arrays:** version 2.4 of X10 (latest version as of this writing) provides two kinds of multi-dimensional arrays, *region-based* arrays and *rail-backed* arrays.

Static analyses: We developed various static analyses to aid generation of better optimized code and to support wider range of MATLAB functionalities. Identification of complex numerical values, handling variables with type-conflicts, array-bounds checks and identification of non-mutable variables are the important analyses that we describe in this thesis.

Template-based builtin framework: MATLAB supports many builtin operations that can operate over a wide variety of run-time types. We have designed and implemented a template-based system that allows us to generate specialized X10 code for a collection of important builtin operations.

Code generation strategies for key language constructs: There are some very significant differences between the semantics of MATLAB and X10. A key difference is that MATLAB is dynamically-typed, whereas X10 is statically-typed. Furthermore, the type rules are quite different, which means that the generated X10 code must include the appropriate explicit type conversion rules, so as to match the MATLAB

semantics. Other MATLAB features, such as multiple returns from functions, a non-standard semantics for `for` loops, and a very general range operator, must also be handled correctly.

Working implementation and performance results: We have implemented the MIX10 compiler over various MATLAB compiler tools provided by *Mc**LAB***. In the process we also implemented some enhancements to these existing tools. We provide performance results for different X10 backends over a set of benchmarks and compare them with results from other MATLAB compilers including Mathworks MATLAB implementation and Octave.

1.2 Thesis Outline

This thesis is divided into 11 chapters, including this one and is structured as follows.

Chapter 2 provides an introduction to the X10 language and describes how it compares to MATLAB from the point of view of language design. *Chapter 3* gives a description of various existing MATLAB compiler tools upon which MIX10 is implemented. *Chapter 4* gives a high-level design of MIX10 and explains the design and need of MIX10 IR. In *Chapter 5* we introduce different types of arrays provided by X10 and describe code generation strategies for them. We also identify pros and cons of both kinds of arrays in the context of X10 as a target language. *Chapter 6* presents the template-based specialization framework for handling MATLAB builtin methods. *Chapter 7* describes the various static analyses that we implemented to help generate more optimized code. In *Chapter 8* we give details of code generation strategies for important MATLAB constructs. *Chapter 9* provides performance results for code generated using MIX10 for a suite of benchmarks. *Chapter 10* provides an overview of related work and *Chapter 11* concludes and outlines possible future work.

Chapter 2

Introduction to $X10$ and contrast with MATLAB

TBD

Chapter 3
Background

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. In this way 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 are actually implemented in native code. We call these functions builtins or builtin functions.

Every MATLAB operator (such as $+$, $*$) is also a builtin function; the operations are merely syntactic sugar for calling the functions that represent the operations (like `plus` for $+$, `mtimes` for `*`).

For an accurate static analysis of MATLAB programs one requires an accurate model of the builtins. In this section we describe how we have modelled 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[Rad12] to find all references to functions that occur in a large corpus of over three thousand MATLAB programs.[1] We recorded the frequency of use for every function and then, using the MAT-LAB function `exist`, which returns whether a name is a variable, user-defined function or builtin, we identified which of these functions is a builtin. This provided us with a list of builtin functions used in real MATLAB programs, with their associated frequency of use. The complete list can be found in *Appendix* **??**.

We selected approximately three hundred of the most frequent functions, excluding dynamic functions like `eval` and graphical user interface functions as our initial set of builtin functions. We also included all the functions that correspond to MATLAB operators, as well as some functions that are closely related to functions in the list.

3.1.2 Finding Builtin Behaviors

In order to build a call graph it is very important to be able to approximate the behavior 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 behavior is actually quite complex, and since the behavior of MATLAB 7 is the defacto specification of the behavior we decided to take a programmatic approach to determining the the behaviors.

We developed a set of scripts that generate 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

---

[1]This is the same set of projects that are used in [DHR11]. The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing.

Figure 6.1. The complete list of result tables can be found in *Appendix* **??**

	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 3.1 Example mclass results for groups of builtin 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` through `i64` represent the mclasses `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.
To save space we have not included the complete generated table, we have left out the columns and rows for unsigned integer mclasses and for handles.

As compared with type rules in other languages, these results may seem a bit strange. For example, the "-" entry for `plus(int16,int32)` in *Figure 6.1*(a) shows that it is an error to add an `int16` to and `int32`. However adding an `int64` to a `double` is allowed and results in an `int64`. Also, note that although the three tables in *Figure 6.1* are similar, they are not identical. For example, in *Figure 6.1*(a), multiplying a `logical` with a `logical` results in a `double`, but using the power operator with two `logical`

arguments throws an error. Finally, note that the tables are not always symmetrical. In particular, the f64 column and row in *Figure 6.1*(b) are not the same.

The reader may have noticed how the superior/inferior m-class relationships as shown in figure *Figure* **??** 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 behavior. 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 behavior, we arranged all of the builtins in a hierarchy - a part of the hierarchy is given in *Figure 6.2*. Leaves of the hierarchy correspond to actual builtins and internal nodes correspond to abstract builtins or a grouping of builtins which share some similar behavior.

To specify the builtins and their tree-structure, we developed a simple domain-specific language. A builtin is specified by values on one line. Values on every line are separated by semicolons. To specify a builtin, the first value has to be the name of the builtin.

If the builtin is abstract, i.e. it refers to a group of builtins, the parent group has to be specified as a second value. If no parent is specified, the specified builtin is a concrete builtin, belonging to the group of the most recently specified abstract builtin. This leads to a very compact representation, a snippet of which is shown in *Figure 6.3*.

Values after the second are used to specify properties or attributes of builtins. Attributes can be specified for abstract builtins, meaning that all children nodes will have that attribute. This motivates structuring all builtins in a tree - if similar builtin functions have the same attributes, then we may only have to specify properties once.

10

Figure 3.2 Subtree of the builtin tree, showing all defined floating point builtins of MAT-LAB. All internal nodes are abstract builtins, the names inside the boxes refer to actual functions. The full tree showing all defined builtins is available at http://www.sable.mcgill.ca/mclab/tamer.html.

```
...

# operates on floating point matrizes
floatFunction; matrixFunction

# proper float functions have a fixed arity, and all operands are floats
properFloatFunction; floatFunction

# unary functions operating on floating point matrizes
unaryFloatFunction; properFloatFunction

# elemental unary functions operating on floating point matrizes
elementalUnaryFloatFunction; unaryFloatFunction
sqrt
realsqrt
erf


...

# float functions with optional arguments or variable number of arguments
improperFloatFunction; floatFunction

...
```

Figure 3.3 Excerpt of the builtin specification, showing definitions for some of the floating point functions shown in *Figure 6.2*. The lines starting with a #-symbol are comments.

The builtin framework takes a specification like shown in *Figure 6.3* as input, and generates a set of Java classes, one for each builtin function, whose inheritance relationship reflects the specified tree. For an abstract builtin, the generated Java class is abstract as well. The builtin framework (the code that generates Java files from the builtin specification) is written in Python.

3.2.1 Builtin Visitor Class

Besides the builtin classes, the builtin framework also generates a visitor class in Java. It allows adding methods to builtins and thus to define flow equations for them using the visitor pattern - a pattern that is already extensively used in the MCSAF analysis framework[Doh11]. In fact, flow analyses themselves are written using the visitor pattern.

The generated visitor class (see *Figure 6.4*) can be used to make flow analyses implement flow equations for all builtins. In order to do so, one has to derive from the visitor class and fill in the class variables used as argument and return values for the case methods. Overriding case methods allows specifying desired flow equations for the corresponding builtin.

Note that the default case for every builtin is to call the parent case - this means that to specify behavior for similar builtins, one only needs to specify the abstract behavior of a group, and the flow analysis framework will automatically apply the correct (most specialized) behavior for a specific builtin. This further motivates the structuring of builtin functions into a tree.

For example, we may find that for some flow analysis, all the flow equations for all functions that are in the group 'UnaryFloatFunction' are the same. So we just need to override the `caseAbstractUnaryFloatFunction()` method, shown in *Figure 6.4*. When executing any case-method of a builtin in that group, its default implementation will call the parent's implementation until reaching `caseAbstractUnaryFloatFunction()`.

The analysis framework allows specification of flow equations for all AST-nodes. Since all the MATLAB operators have associated AST-nodes, one can specify flow equations for operators using the analysis framework. Our set of builtin functions includes all the MATLAB operators, so analysis writer may alternatively define flow equations for operators using the builtin framework, rather than the analysis framework. For the analyses presented in this thesis we have opted to do so, to have fewer flow equations for AST-nodes, and have all the behavior of builtin functions in one place.

Using this approach, an intraprocedural analysis that is aware of builtins will consist of a flow analysis class defining flow equations for AST-nodes, and a class defining flow equations for builtin functions. Both are defined as extensions of visitor classes - the flow analysis is a visitor class for the AST-node hierarchy, and the builtin visitor for the hierarchy of builtins.

```java
public abstract class BuiltinVisitor<A,R> {
  public abstract R caseBuiltin(Builtin builtin,A arg);

  ...

  //operates on floating point matrizes
  public R caseAbstractFloatFunction(Builtin builtin,A arg){
    return caseAbstractMatrixFunction(builtin,arg);
  }

  //proper float functions have a fixed arity, and all operands are floats
  public R caseAbstractProperFloatFunction(Builtin builtin,A arg){
    return caseAbstractFloatFunction(builtin,arg);
  }

  //unary functions operating on floating point matrizes
  public R caseAbstractUnaryFloatFunction(Builtin builtin,A arg){
    return caseAbstractProperFloatFunction(builtin,arg);
  }

  //elemental unary functions operating on floating point matrizes
  public R caseAbstractElementalUnaryFloatFunction(Builtin builtin,A arg){
    return caseAbstractUnaryFloatFunction(builtin,arg); }
  public R caseSqrt(Builtin builtin,A arg){
    return caseAbstractElementalUnaryFloatFunction(builtin,arg); }
  public R caseRealsqrt(Builtin builtin,A arg){
    return caseAbstractElementalUnaryFloatFunction(builtin,arg); }
  public R caseErf(Builtin builtin,A arg){
    return caseAbstractElementalUnaryFloatFunction(builtin,arg); }

  ...

  //float function with optional arguments or variable number of arguments
  public R caseAbstractImproperFloatFunction(Builtin builtin,A arg){
    return caseAbstractFloatFunction(builtin,arg); }

  ...

}
```

Figure 3.4 Excerpt of the visitor class `BuiltinVisitor` that is generated by the builtin framework using the specification shown in *Figure 6.3*. The comments are copied from the specification file by the builtin framework.

3.3 Builtin Function Categories

We categorize the MATLAB builtin functions according to many properties, such as mclass, arity, shape, semantics. To minimize the number of flow equations that need to be specified for analyses and properties, they may require different kinds of groupings for the builtins, based on the semantics of the analyses or property. Ideally, for every analysis there should be categories grouping builtins, so that the fewest possible flow equations have to be specified.

In general this is not possible, because we are using a tree to categorize builtins. Nevertheless we attempted to find as many useful categories as possible, which are partly inspired by potential needs for analysis, and partly by the similarities of existing builtin functions, and the categories we found.

Another motivation for the heavy use of categories is that our framework does not yet implement all MATLAB builtin functions, and we want to minimize the amount of work required to add a builtin. When adding builtins that fit in already existing categories, one can reuse the attributes and flow equations specified for these categories.

Effectively, we have made a survey of all the builtins, learning about their semantics, interfaces and mclass-behavior, and have retrofitted them with an object-hierarchy. This approach seems natural because we do generate object-oriented Java code for the builtins, which uses that same hierarchy.

In the following we list the categories we have used to group functions. We present every category along with their alternatives; the alternatives are mutually exclusive. We use naming conventions that attempt to follow MATLAB terminology, but some may only be valid for the builtin framework.

pure, impure

> Pure functions have no side effects, change no state, internal or otherwise, and always return the same result when called with the same arguments.

matrix, cell, struct, object, versatile

> Matrix functions operate on MATLAB values that are numerical, `logical` or `char`.

all arguments, operands and results should have these mclasses. For example, numerical functions are matrix functions.

Cell functions operate on cell arrays, struct functions operate on structures, object functions operate on objects.

Versatile functions operate on multiple kinds of the above categories. Some may operate on any MATLAB value. For example, query functions like `numel` only depend on the shape of the argument - since every MATLAB value has a shape, the function works on all arguments.

anyMatrix, numeric, float

These categories are sub-categories of the matrix category.

A function belonging to the anyMatrix category operates on numerical, `logical` or `char` arrays. Numeric Functions operate on numbers. They may also accept `char` or `logical` values, but these values will be coerced to `double`, so the actual operation and the result will be numerical.

Float functions only operate on floats, i.e. `single` or `double` values. Some of the functions in this category may also accept different arguments and coerce them to `double`.

proper/improper

Proper functions have strict arity, and the arguments are operands. As can be seen in *Figure 6.5*, a lot of numeric functions are proper. Almost all operators are proper functions (an exception is the colon operator).

Improper functions may operate on a variable number of operands, or allow optional parameters. Some may accept (optional) parameters specifying options for the computation to be performed - these option parameters are not operands and may be of a type that functions within its category do not accept as operands.

For example, the float function `eps` (machine epsilon) is improper: it allows zero arguments or one floating point argument, but it also supports the `char` values `'single'` and `'double'` as a sole argument. The function will always return a float value.

unary, binary

A unary function requires exactly one argument, a binary function requires exactly two.

elemental, array

The elemental category refers to element-wise functions, i.e. functions which operate on every element in an array independently. The result will have the same shape as the inputs. The array functions operate on the whole array at once. For example matrix multiplication belongs to the array category.

The notion of elemental and array functions corresponds to MATLAB's notion of array vs matrix operators, introduced in Sec. **??**. Note the different terminology to avoid re-using the term 'matrix'.

dimensionSensitive   Dimension-sensitive functions are of the form $f(M, [dim])$, i.e. they take some array as the first argument, and allow a second optional argument `dim`. This argument specifies the dimension along which to operate. By default the dimension will be the first non-singleton dimension.

dimensionCollapsing

A dimension-collapsing function is a dimension-sensitive function which will collapse every value along the operated dimension into one value, and return a new matrix with a corresponding shape. For example the `sum` function sums all values along the dimension it operates, turning them into single values. Other examples are the functions `prod`, `mean`, `mode`, `min` and `max`.

query

A query is a function that given some arguments, will return a scalar or a vector containing information about the argument(s). The computation summarizes the information contained in the arguments in some fashion.

toLogical, toDouble

These categories refer to the mclass of the result of the computation. We use these

as sub-categories of query. functions in the toDouble category will always return a `double` result, functions in the toLogical category will return `logical` results.

Besides the above general categories, we use ad hoc ones that attempt to group builtin functions according to their semantics, i.e. functions performing similar computation should be grouped together. For example in *Figure 6.5*, there are categories like 'trigonometric function' or 'factorization function'.

Within the tree-structure, categories are combined, creating more and more refined categories. For example, going down the tree one can reach the combination of categories termed `ElementalBinaryToLogicalMatrixQuery`. Functions in this combined category refer to query functions operating on matrices only, which take exactly two arguments, operate element-wise and will return values of mclass logical. The proliferation of these long names may explain some of our naming conventions, which are largely motivated by the desire for brevity, to keep combined categories manageable.

An example of a complete path along the builtin tree, showing further and further refinement of categories, is shown in *Figure 6.5*. It also shows alternative categories along the path.

3.4 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 attributes.

In the builtin specification language, an attribute is just a name, with a set of arguments that follow it. In the specification language the attributes are defined on the same line as the builtin itself. Starting with the third value, every value specifies an attribute. Internally we call attributes to builtins 'tags'.

A specific attribute 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.

Figure 3.5 An example showing all ancestors of a group of builtins, and all siblings for all these ancestors. This shows the refinement of categories from the top category of 'builtin' going to a specific builtin, and what the alternative categories are along the way.

It is possible to add new kinds of attributes to the builtin specification language. One merely has to provide a function [2] with a specific function interface that provides information about the specified builtin and the argument string for the attribute. The function has to return Java code that will be inserted in the generated Builtin class. The function may also update a list of interfaces that the generated builtin class implements. The name of that function is the name of the attribute as used in the builtin specification language. The argument to the attribute is an arbitrary string. It may, however, not contain a semicolon, because it is used to match the end of the attribute.

3.5   The Class and MatlabClass attribute

In order to build a complete callgraph, we need to know of what mclass a variable may be during runtime, due to the overloading lookup semantics introduced in Sec. **??**. To have complete knowledge of all possible mclasses for all variables at all times, we need to know how they behave with respect to mclasses. We opted to define all this information as attributes to builtins, defined in the builtin specification along with builtins themselves.

We defined an attribute called `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 object.

The mclass flow equation object itself is defined in the builtin specification as an argument to the `Class` attribute, using a small domain specific language that allows matching argument mclasses. It returns result mclasses based on matches. We have decided to build this little domain specific language because of the complexity of some builtins, and our desire to define mclass flow equations in a compact way.

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 added the `MatlabClass` attribute. It allows us to specify the exact MATLAB semantics - and thus provides an exact definition and documentation of MATLAB class semantics. Refer to *Figure 6.6* for an example usage of both a `Class` attribute and a `MatlabClass` attribute

---

[2]attribute functions are defined in processTags.py in the builtin framework

```
...

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

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

...
```

Figure 3.6 Excerpt of the builtin specification with the Class and MatlabClass attributes added
in. The Class attribute for `unaryNumericFunction` defines the mclass flow
equations for unary functions taking numeric arguments, and applies for all builtins
in the group. It specifies that given a numeric argument, the result will have the
same mclass (`numeric>0`). For `char` and `double` the result will be a double.
Note the `MatlabClass` attribute defined for `conj` and `sign`. These functions
have exact MATLAB semantics that differ from the default used by the builtin frame-
work: they disallow `logical` arguments (but not `char` arguments), using them will
result in an error.

showing slightly different behavior.

A detailed description of the domain-specific language used to represent mclass flow
equations is presented in *Appendix* **??**.

3.6 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. We have used
our analysis of the builtins to organize builtin functions into a tree structure, making it
easier to work with builtin functions.

This builtin framework is extensible both by allowing the quick addition of more builtin

21

functions; and by allowing to specify more information and behavior for builtin functions. This can be done either adding new properties to the framework itself; or by implementing visitor classes.

The compact representation of builtins also allows changing the organization of builtins. This means that the whole framework may evolve as our understanding of builtin functions and our requirements for analyses evolve.[3]

---

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

Chapter 4
Design of the $\mathrm{MIX10}$ compiler and $\mathrm{MIX10}$ IR

As indicated in *Figure* **??**, we build upon the MCSAF framework by adding taming transformations and by producing a more specialized Tame IR. The MCSAF framework provides us with a three-address form of the AST, reducing many complicated MATLAB constructs. We further reduce the AST to build the Tame IR. The main contributions of the Tame IR, beyond the three address form previously provided by MCSAF are:

- Rather than providing a reduced form of the AST, as provided by MCSAF, we implement the Tame IR as a complete set of new nodes. The interfaces of these nodes enforce the constraints of the IR.

- The Tame IR reduces the total number of possible AST nodes. In particular, we remove all expression nodes, and express their operations in terms of statements and function calls.

- The Tame IR reduces some complexity of MATLAB. Some of these reductions would not have been possible to be provided by the MCSAF framework, because it is completely semantics preserving. Because the tamer framework does impose constraints on MATLAB to make it amenable to static compilation, it is possible to further reduce the AST in ways that is not possible with semantics-preserving transformations. In particular, we simplify lambda expressions and remove switch statements; we also place all comments into empty statements, rather than have them annotated to statement nodes.

- The Tame IR specialize nodes according to their semantics, and provides nodes that signify the operation performed.

- The Tame IR provides information that is not available in the AST. In particular, it separates functions and variables, utilizing the kind analysis [DHR11].

Rather than implementing completely new nodes, all Tame IR nodes are extensions of existing AST nodes. This means that any Tame IR program tree is a valid AST as well. A program in the Tame IR is also a valid MATLAB program, with one exception, which is discussed in Sec. 4.1.1. This difference is removed when the Tame is pretty printed, which will produce valid MATLAB again.

The intention of the Tame IR is to make it easier to implement analyses, by reducing the number of nodes, by specializing nodes to signify their operation, and by providing some static information. By keeping the Tame IR an almost valid AST, any analysis written for the AST should work for the Tame IR as well; by keeping it valid MATLAB (at least when pretty printed), it should be easier to debug analyses and transformations. One goal for our overall Taming framework is to produce an IR whose operations are low-level enough to map fairly naturally to static languages like FORTRAN.

Besides providing the IR nodes and the transformations to build the Tame IR, we have also extended the visitor classes and flow analyses of the MCSAF framework so that it can be used to implement flow analyses that explicitly use the IR.

In the following sections we first introduce the Tame IR and its nodes, and then provide an overview of some of the transformations used to arrive at the Tame IR.

4.1 The Tame IR

The Tame IR consists of nodes that extend existing AST nodes. Some of these nodes extend the AST and merely enforce constraints that correspond to the three-address form semantics of the Tame IR. Some nodes are extensions of the AST nodes that do not change the interface at all, they merely exist to complete the Tame IR, so that a program may consist only of IR Statements.

For assignment nodes, however, we provide several specializations that correspond to many different operations that can be performed by an assignment statement. The AST only provides a single assignment statement with an expression on the lhs and rhs. This is what the three-address form of MCSAF provides as well, even when the three-address transformations will have reduced the actual structure of an assignment.

In the following sections, we present all the Nodes of the Tame IR. A complete grammar is given in appendix *Appendix C*.

4.1.1 Assignment Statements

For the Tame IR, we have extended the AST's assignment statement into several specialized versions, as seen in *Figure 4.1*. These all represent different operations in terms of assignment statements. Note in particular that we have different nodes for the syntactically identical array accesses and calls, given that the Tame IR differentiates between them, unlike the AST. In the following we describe all the different kinds of extensions of the assignment statement that are part of the Tame IR.
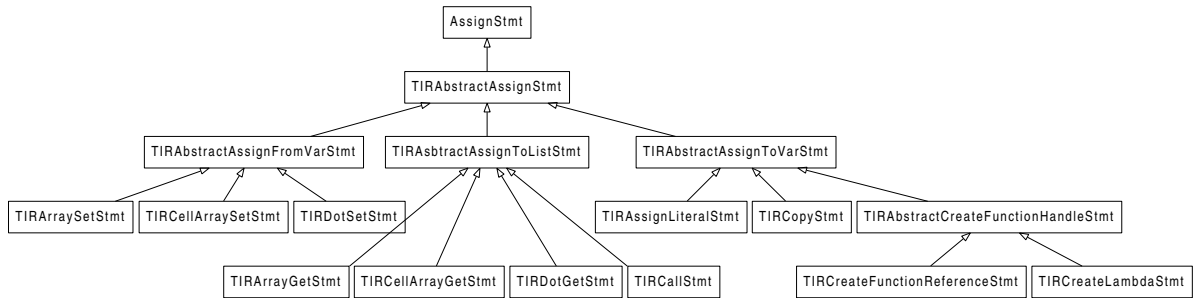
Figure 4.1 Specializations of an assignment statement

TIRAbstractAssignStmt

> An abstract class representing all assignment nodes of the Tame IR. This class extends the AST node `AssignStmt`. The analysis framework allows specifying flow equations for every node, including all the abstract nodes.

25

TIRAbstractAssignFromVarStmt

Assignments from variables are of the form

```
... = x
```

i.e. they have a rhs which is a name referring to a variable. This is an abstract node class representing the following nodes:

TIRArraySetStmt, TIRDotSetStmt, TIRCellArraySetStmt

The 'set'-assignments represent AssignFromVarStmts whose lhs are indexing operations, i.e. they represent assignment indexing operations that correspond to the MATLAB builtin function `subsasgn`. For example, they represent the following operations:

```
a(i,j)= x, a.s = x, a{i,j} = x
```

TIRAbstractAssignToListStmt

Assignments to lists are assignments with multiple possible target variables. I.e. they are assignments of the form

```
[v1, v2, v3, ... , vn] = ...
```

Within the Tame IR, it is allowed that the list of result variables is empty, which is not valid in MATLAB. This is the only deviation of the Tame IR from being valid MATLAB (the AST does not enforce this restriction). Empty lhs lists are used to represent expression statements. For example, within the Tame IR, a statement like

```
foo(3);
```

is represented as

```
[] = foo(3);
```

This allows us to represent all expressions in terms of statements, while having IR nodes that are merely extended AST nodes (in this case `AssignmentStmt`), while also not having multiple versions for statements, either as assignment or expression statements.

When pretty-printed, an assignment with an empty lhs list will return an expression

statement.

TIRArrayGetStmt, TIRCellArrayGetStmt, TIRDotGetStmt

> The 'get'-assignments are assignments to lists that are represented by the MAT-LAB builtin function `subsref`, i.e. they have indexing operations on the rhs. Note that structure-referencing and cell-indexing may result in multiple return values that can be assigned, while array-indexing produces only one value. However, array-indexing is also used when calling a value of mclass `function_handle`. In that case, the referenced function gets called, possibly resulting in multiple return values. When any of the above operations is overloaded, the operation may also result in multiple return values.

TIRCallStmt

> Calls are assignments of the form

```
[r1, r2, ... , rn] = f(a1, a2, ..., an);
```

> Where r_i and a_i are variables. Note the similarity to the array-get statement. The difference is that f is a name that has to refer to a function.

TIRAbstractAssignToVarStmt

> These represent assignments of the form,

```
x = ...
```

> There is a name on the lhs representing a single variable. These are used for assignments where there always is exactly one variable on the lhs. This makes them simpler to analyze than the assignments to lists, because there don't need to be any checks for the existence of enough target variables, etc.

TIRAssignLiteralStmt

> Literal assignments are used to assign numerical and string literals to a variable, i.e. they may be used to represent the following statements:

```
x = 3, x = 'hi'
```

27

In MATLAB, `true` and `false` are not literals, but builtin functions. These functions actually allow arguments specifying matrix dimensions to produce logical matrices.

The assign-literal statements are the only place in the Tame IR where literals may occur; other statements usually operate on just variables.

TIRCreateFunctionHandleStmt

These assign-to-var statements allow the creation of function handles, either creating function pointers, or by creating an anonymous function using lambda. They are thus of the form

```
t = @f;
```

or

```
t = @(x1,x2,...)f(a1,a2,..,an,x1,x2,...);
```

where f is a name referring to a function. The variables a_i through a_n encapsulate workspace variables within the anonymous function, there may be 0 or more of such variables. The transformation from arbitrary lambda expressions to statements of the above form is discussed in detail in Sec. 4.3.

TIRCopyStmt

Copies are assignments of the form

```
x = y;
```

where *x* and *y* are names referring to variables.

4.1.2 Control Flow Statements

TIRIfStmt, TIRWhileStmt

The if and while statements in the Tame IR are almost the same as the corresponding statements in the AST. The only constraint, being a three-address form, is that the condition-expressions have to be names referring to variables.

TIRForStmt

The for statement in the Tame IR is of the form

```
for i = low:inc:hi

...

end
```

where *i*, *low*, *inc* and *hi* are names referring to variables. *inc* is optional.

TIRReturnStmt, TIRBreakStmt, TIRContinueStmt

These control flow statements are the same as their AST counterparts.

4.1.3   Other Statements

TIRGlobalStmt, TIRPersistentSmt

These statements allow declaring variables to be global or persistent. The Tame IR imposes the constraint on MATLAB that no variable may be used before a global definition. MATLAB merely issues a warning in this case. MATLAB does not allow using a persistent variable before the declaration.

TIRCommentStmt

In the AST, comments are annotated to AST-nodes. When replacing AST nodes with other AST nodes, one would thus have to ensure that comments are copied as well. In order to make transformations of the tree easier, we have opted to place all comments into empty statements, so that no other statement may have annotated comments.

4.1.4   Non-Statement Nodes

Besides all the above statement nodes, the Tame IR includes the following nodes which are not statements.

TIRNode, TIRStmt

These are interfaces. Any Tame IR node implements `TIRNode`. Any Statement of the Tame IR implements `TIRStmt`.

TIRFunction

`TIRFunction` is an extension of the function node of the AST. It ensures that all

29

statements inside the body are `TIRStmt` nodes. The functions also include information that is not readily available to AST function nodes, namely a simple symbol table separating names into functions and variables (the result of the kind analysis). It also provides the list of global and persistent variables declared inside the function body.

TIRStatementList

A simple extension of the `StatementList` that is part of the AST, to ensure that all elements are `TIRStmt` nodes.

TIRCommaSeparatedList

Used as a list of names for arguments to calls, and for indices of indexing operations, and targets in list-assignments. Besides names, indexing operations may include a colon (:), for example as used in the indexing operation

```
a(:,3)
```

Here, `:,3` would be represented as a comma-separated list.

As more of the MATLAB language is supported, more possible elements may get added, for example MATLAB's tilde expression '~', which allows discarding results of calls.

4.2 Tame IR Transformations

The Tame IR of an AST is built by transforming the three-address form produced by the MCSAF framework. Given this three-address form, most of the transformations produce equivalent nodes of the IR, merely checking constraints. To transform an incoming assignment statement, the transformations have to check what kind of assignment it is, and produce the appropriate IR assignment. All these transformations do not actually transform the underlying MATLAB code, they merely change the representation of it.

Besides these node-representation transformations, the Tame IR transformations also include some transformations that actually change the underlying MATLAB code. These

are presented below. Note how some of these transformations impose slight constraints on the MATLAB code, which are thus part of the Tame MATLAB language subset.

4.2.1 Reduction of Operations to Calls

The Tame IR has no operators. In order to transform to the Tame IR, all operators have to be transformed into calls to equivalent builtin functions. Note that users may already be using builtin functions rather than operators, so after the transformation, all operations are expressed in the same way. The list of operators thus transformed in presented in *Table 4.1*.

The missing short circuit logical operations (`&&` and `||`) are already reduced by the MCSAF framework into equivalent if-then-else statements.

The transformation does not reduce the indexing operators '()', '{}' and '.'. They do correspond to the builtin functions `subsref` and `subsasgn`, for indexing operations on the rhs and lhs, respectively. Note, however, that MATLAB uses the same indexing operator for all indexing operations. Consequently, MATLAB internally has to add arguments to specify the exact indexing operation used. This information is stored in a structure. For example, an indexing operation like

```
x = a(i,j);
```

May look like the following, if `subsref` was used explicitly:

```
s.type = '()';
s.subs = {i,j};
x = subsref(a,s);
```

Note the structure that contains the type of indexing (as a string), and the indices, which are themselves stored as a cell array. If the Tame IR reduced indexing operators, it would actually generate more complex code, which may be harder to analyze.

`x = a + b`	`x = plus(a,b)`

       (a) operation   (b) equivalent call

Figure 4.2 Transforming operations to calls

With this in mind, analyses have to be aware that it is possible not only to overload calls to functions, but also indexing operations. An accurate analysis thus has to check for overloaded functions for calls, as well as all 'get'-assignments and all 'set'-assignments.

After the operator reduction, analyses written for the Tame IR should utilize the builtin framework. That is, analysis writers should provide a flow analysis of the AST nodes using the MCSAF analysis framework, and flow equations for builtins using the builtin framework. This simplifies the flow analysis of the AST nodes themselves, because there are fewer nodes, and helps separating the definition of the flow equations of the AST-nodes from the definition of flow equations for builtin operations and functions.

binary numerical operators

+	plus
–	minus
*	mtimes
/	mrdivide
\	mldivide
^	mpower
.*	times
./	rdivide
.	ldivide
.^	power

other binary operators

&	and
|	or
<	lt
>	gt
<=	le
>=	ge
==	eq
=	ne

unary operators

–	uminus
+	uplus
.'	transpose
'	ctranspose
~	not

colon

:		colon
:	:	colon

Table 4.1 MATLAB operators and their corresponding builtin functions.

4.3 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 single expression (rather than, say, statement lists), that the MCSAF framework leaves intact in their original form, due to the difficulty of reducing a lambda expression while still maintaining the full MATLAB semantics. For the Tame IR 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 *Figure 4.3*(a) to the code in *Figure 4.3*(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).

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

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

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

(a) lambda   (b) transformed lambda

Figure 4.3 Transforming `lambda` expressions

4.4   Switch simplification

As illustrated in *Figure 4.4*(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 comparisons for the same case block, MATLAB allows using cell arrays of case expressions, for example {2, 3} in *Figure 4.4*(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 *Figure 4.4*(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
```

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

(a) switch   (b) transformed switch

Figure 4.4 Transforming `switch` statements

4.5   Summary

We have provided a simplified IR that can be used to represent MATLAB, which enables implementing more simplified flow analyses, working together with the builtin framework, and which should help facilitate static compilation of MATLAB programs.

Chapter 5
$X10$ **Arrays**

This chapter introduces the interprocedural analysis framework. We have previously introduced the builtin framework and the Tame IR. In the next chapter we will introduce the value analysis, an interprocedural analysis that uses all these tools to build a callgraph with annotated type information. In order to implement this interprocedural analysis, we have developed the interprocedural analysis framework.

The interprocedural analysis framework 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.

In the following sections we will introduce the interprocedural analysis framework as an extension of the intraprocedural analysis framework, and how it works in tandem with callgraph objects and the lookup objects, as well as how the framework deals with recursion. To help potential analysis writers, we have indicated the names of Java classes that correspond to the contexts in bold.

5.1 The Function Collection Object

In order to represent callgraphs we use an object which we call **FunctionCollection**. It is, as the name suggests, a collection of nodes representing functions, indexed by so function reference objects. Objects of type **FunctionReference** act as unique identifiers for functions. They store a function's name, in which file it is contained if applicable, and what kind of function it is (primary function, subfunction, nested function, builtin function, constructor). For nested functions, it stores in which function it is contained. Function reference objects give enough information to load a function from a file.

Nodes in the function collection not only store the code of the function and a function reference; they also provide information about its environment. The node provides a MAT-LAB function lookup object which is able to completely resolve any function call coming from the function. It includes information about the MATLAB path environment and other functions contained in the same file. The lookup information is provided given a function name, and optionally an mclass name (to find overloaded versions); and will return a function reference allowing the loading of functions.

The lookup information allows us to build a callgraph knowing only an entry point and a path environment, and using semantics for finding functions that correspond to the way MATLAB finds functions at runtime. This is bridging the gap between a dynamic language and static compilers, which usually require specifying what source code files are required for compilation.

The simple function collection uses only the lookup information contained in its nodes to built an approximation of a callgraph, which is naturally incomplete. We have used it for the development of the Tamer framework, as it provides a simple way to generate a callgraph which excludes discovering overloaded calls and propagation of function handles. We have implemented slightly different versions of the function collection, which are described in the table below.

SimpleFunctionCollection	A simple callgraph object built using MATLAB lookup semantics excluding overloading. Function Handles are loaded only in the functions where the handle is created. Obviously this an incomplete callgraph, but may be used by software tools that do not need a complete callgraph, and where the simplicity can be useful.
IncrementalFunctionCollection	callgraph that does the same lookup as the FunctionCollection, but does not actually load functions until they are requested. This is used to build the callgraph
CompleteFunctionCollection	callgraph that includes call sites for every function node and correctly represents overloading can calling function handles. This is produced by the Tamer using interprocedural analyses. This callgraph can be used to build further interprocedural analysis that are not extensions of the value analysis. It can also be used as a starting point for static backends.

Table 5.1 The different kinds of Function Collection objects.

5.2 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, called **InterproceduralAnalysisNode**, 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. An InterproceduralAnalysisNode is generic in the intraprocedural

analysis, the context and the result - these have to be defined by an actual implementation of an interprocedural analysis.

Every interprocedural analysis has an associated FunctionCollection object, which may initially contain only one function acting as the entry point for the program (i.e. when building a callgraph using an IncrementalFunctionCollection). The interprocedural analysis requires a context (argument flow set) for the entry point to the program.

Algorithm

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.

5.2.1 Contexts

In order to implement an interprocedural analysis, one has to define a context object. These may be the flow information of the arguments of a call; but it could be any information. The analysis itself is context-sensitive, meaning that if there are multiple calls to one function with different contexts, they are all represented by different interprocedural analysis nodes. The interprocedural analysis framework never merges contexts, which would have to be done by the specific analysis if desired.

Interprocedural analysis nodes are cached. Thus if a function/context pair is called a second time, the information will be readily available.

Note that in order to completely resolve calls, the flow information and the contexts have to include mclass information for variables and arguments. In order to resolve calls to function handles, the contexts have to store which arguments may refer to function handles

(and which functions they refer to).

Once the complete callgraph is built, further analyses don't need to flow mclass information, because all possible calls are resolved. But this information may still be useful to obtain more accurate analysis result, by knowing which information to flow into which calls for ambiguous call sites (see Sec. 5.2.3) - that is why the value analysis presented in *Chapter* **??** allows extending the flow-sets, to allow flowing information for different analyses together in one analysis, and get a more precise overall result.

5.2.2 Call Strings

When analyzing a function f for a given context c_f, and encountering a call to some other function g, the interprocedural analysis framework suspends the analysis of f in order to analyze the encountered call. The flow analysis has to provide a context c_g for the call to g, and an intraprocedural analysis will be created that will analyze g with c_g.
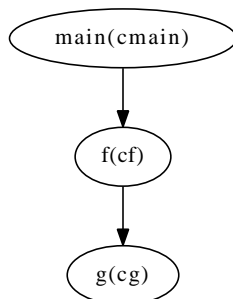
Figure 5.1 A small program where *main* calls f calls g. The call string for $g(c_g)$ in this example may be $main(c_{main}) : f(c_f) : g(c_g)$.

The set of currently suspended functions (in *Figure 5.3 main* and f), which are awaiting results of encountered calls that need to be analyzed correspond to the callstack of these functions at runtime, at least for non-recursive programs. We call the chain of these functions, together with their contexts a **CallString**. Every function/context pair, i.e. the associated interprocedural analysis node, has an associated call string, which corresponds to one possible stack trace during runtime. Note that interprocedural analysis nodes are cached, and may be reused. Thus in the above example, if the main function also calls g

41

with context c_g, the results of the interprocedural analysis node created for the call encountered in function f will be reused.
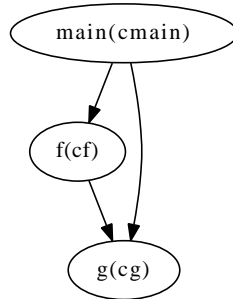
Figure 5.2 Here, *main* also calls g, also with context c_g. Since the interprocedural analysis node for $g(c_g)$ is reused, the call string will be reused as well.

Since the interprocedural analysis node is reused, it will have the same call string. So the call string is not an exact representation of a call stack for every call, it is merely the exact representation of one possible call stack that will reach a given function/context pair. Note that for purposes of error reporting, the call string can be presented to the user as a stack trace.

5.2.3   Callsite

Any statement representing a call may actually represent multiple possible calls. For example a call to a function g may be overloaded, so if arguments may have different possible mclasses, different functions named g may be called. Also, because it is up to an actual analysis to define its notion of what a context is, it is possible that an analysis may decide to produce multiple contexts for one call to a function f. This would create specialized versions of a function from a single call (this is actually possible in the value analysis presented in *Chapter* **??**). A third way in which a statement may represent multiple possible calls is via function handles. An TIRArrayGet statement may trigger a call if the represented array is actually found to be a function handle (we call the variable accessed in a TIRArrayGet statement an 'array' simply because it is used in an array-indexing operation, but it could be any variable). If that function handle may refer to multiple possible functions at runtime,

then the function handle access may refer to multiple possible calls.
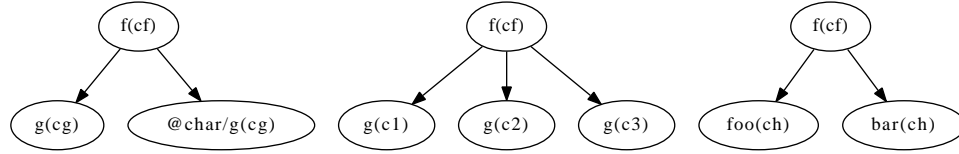
Figure 5.3 This figure shows examples how it is possible for one single call site to refer to multiple possible calls. This may be due to overloading, creation of multiple contexts for a single call, or function handles.

In order to be able to represent multiple possible call edges coming out of a statement, we associate any statement that includes any calls with a **Callsite** object. This callsite can store multiple possible call edges as function/context pairs, which we call a "call" in the interprocedural analysis framework. An intraprocedural analysis, in order to request the result of a call, has to request a callsite object for a calling statement. It may then request arbitrary calls from that callsite object, which will all get associated with the calling statement.

5.2.4 Recursion

The interprocedural analysis framework supports simple and mutual recursion by performing a fixed point iteration within the first recursive interprocedural analysis node. In order to identify recursive and mutually recursive calls we use the call strings introduced in Sec. 5.2.2. While we established that there is no guarantee which stack trace the call string represents, we know that it will always represent one possible stack trace. Since the call stacks of all recursive and mutual recursive calls must include the function, we merely need to check, for any call, whether it already exists in its call string.

If it does, we have identified a recursive call, and must perform a fixed point iteration. To do so, we label the intraprocedural analysis node associated with the recursive call (i.e. the call to $f(c_f)$ in *Figure 5.4*) as recursive. This will trigger the fixed point iteration. Because we need a result for the recursive call to continue analyzing, an actual analysis implementation has to provide a default value as a first approximation, which may be just
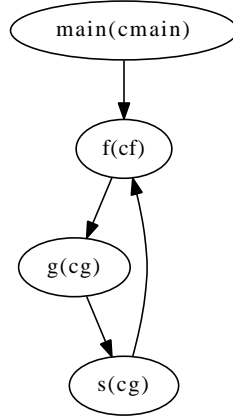
Figure 5.4 Example of a recursive program. The call in $s(c_s)$ to $f(c_f)$ triggers the fixed point iteration of $f(c_f)$. $f(c_f)$ is the first recursive interprocedural analysis node.

bottom. Once the intraprocedural contained in the interprocedural analysis node associated with the recursive call is completed, the result is stored as a new partial result. The analysis is then recomputed, using this new partial result for the recursive call. When a new partial result is the same as a previous partial result, we have completed the fixed point iteration. Note that the computation resulting in the new partial result uses the previous partial result for its recursive call - but since they are the same, we have made a complete analysis using the final result for the recursive call.

Note that the while the fixed point iteration is being computed, all calls below the recursive call (i.e. the calls $g(c_g)$ and $s(c_s)$ in *Figure 5.4*) always return partial results. Thus we cannot cache the nodes and their results, and have to continuously invalidate all the corresponding interprocedural analysis nodes.

Note that the analysis treats calls to the same function with different contexts as different functions. No fixed point iteration is performed to resolve recursive calls with different contexts, because they represent different underlying intraprocedural analyses. Thus it is possible to create infinite call strings, as shown in *Figure 5.5*. It is up to the actual analysis implementation to ensure this does not happen. A simple strategy would be to ensure that there are only a finite number of possible contexts for every function. Another strategy is for the intraprocedural analysis to check the current call string before requesting a call, to
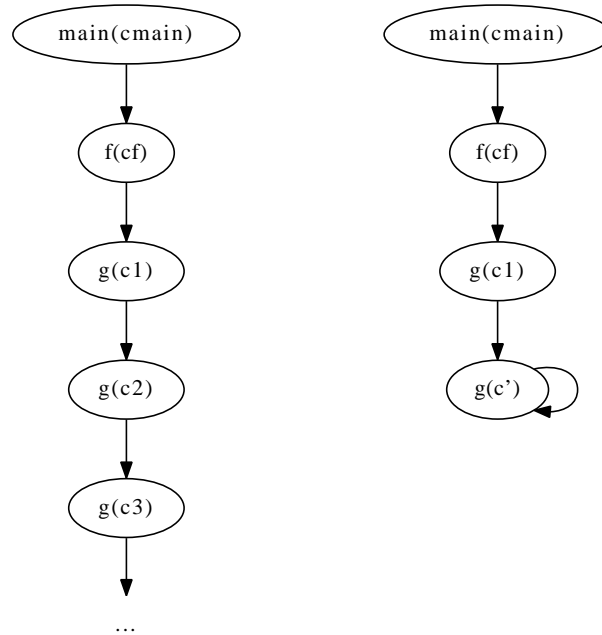
Figure 5.5 Example of a recursive program, showing how recursive calls with different contexts can create infinite chains of calls on the left. An interprocedural analysis implementation has to catch such cases and create a finite number of contexts, as shown on the right, where the contexts c_2 and onward are replaced with c'. In this case the interprocedural analysis framework will perform a fixed point iteration on $f(c')$.

ensure that the function to be called does not already exist in the call string. If it does, the intraprocedural analysis should push up the context to a finite representation (shown in *Figure 5.5*).

5.3 Summary

We have presented an interprocedural analysis framework that we hope is flexible enough to allow different kinds of full-program analyses, while powerful enough to deal with issues such as recursion and ambiguous call sites. This analysis framework is a key component of our value analysis (presented in the next chapter), and the overall callgraph construction of the Tamer.

Chapter 6
Framework for handling builtin functions

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. In this way 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 are actually implemented in native code. We call these functions builtins or builtin functions.

Every MATLAB operator (such as $+$, $*$) is also a builtin function; the operations are merely syntactic sugar for calling the functions that represent the operations (like `plus` for `+`, `mtimes` for `*`).

For an accurate static analysis of MATLAB programs one requires an accurate model of the builtins. In this section we describe how we have modelled the builtins and how we integrate the analysis into the static interprocedural analysis framework.

6.1   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.

6.1.1 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[Rad12] to find all references to functions that occur in a large corpus of over three thousand MATLAB programs.[1] 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 of these functions is a builtin. This provided us with a list of builtin functions used in real MATLAB programs, with their associated frequency of use. The complete list can be found in *Appendix* **??**.

We selected approximately three hundred of the most frequent functions, excluding dynamic functions like `eval` and graphical user interface functions as our initial set of builtin functions. We also included all the functions that correspond to MATLAB operators, as well as some functions that are closely related to functions in the list.

6.1.2 Finding Builtin Behaviors

In order to build a call graph it is very important to be able to approximate the behavior 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 behavior is actually quite complex, and since the behavior of MATLAB 7 is the defacto specification of the behavior we decided to take a programmatic approach to determining the the behaviors.

We developed a set of scripts that generate 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

---

[1]This is the same set of projects that are used in [DHR11]. The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing.

Figure 6.1. The complete list of result tables can be found in *Appendix* **??**

	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 6.1 Example mclass results for groups of builtin 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` through `i64` represent the mclasses `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.
To save space we have not included the complete generated table, we have left out the columns and rows for unsigned integer mclasses and for handles.

As compared with type rules in other languages, these results may seem a bit strange. For example, the "-" entry for `plus(int16,int32)` in *Figure 6.1*(a) shows that it is an error to add an `int16` to and `int32`. However adding an `int64` to a `double` is allowed and results in an `int64`. Also, note that although the three tables in *Figure 6.1* are similar, they are not identical. For example, in *Figure 6.1*(a), multiplying a `logical` with a `logical` results in a `double`, but using the power operator with two `logical`

arguments throws an error. Finally, note that the tables are not always symmetrical. In particular, the f64 column and row in *Figure 6.1*(b) are not the same.

The reader may have noticed how the superior/inferior m-class relationships as shown in figure *Figure* **??** 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 behavior. 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.

6.2 Specifying Builtins

To capture the regularities in the builtin behavior, we arranged all of the builtins in a hierarchy - a part of the hierarchy is given in *Figure 6.2*. Leaves of the hierarchy correspond to actual builtins and internal nodes correspond to abstract builtins or a grouping of builtins which share some similar behavior.

To specify the builtins and their tree-structure, we developed a simple domain-specific language. A builtin is specified by values on one line. Values on every line are separated by semicolons. To specify a builtin, the first value has to be the name of the builtin.

If the builtin is abstract, i.e. it refers to a group of builtins, the parent group has to be specified as a second value. If no parent is specified, the specified builtin is a concrete builtin, belonging to the group of the most recently specified abstract builtin. This leads to a very compact representation, a snippet of which is shown in *Figure 6.3*.

Values after the second are used to specify properties or attributes of builtins. Attributes can be specified for abstract builtins, meaning that all children nodes will have that attribute. This motivates structuring all builtins in a tree - if similar builtin functions have the same attributes, then we may only have to specify properties once.

50

Figure 6.2 Subtree of the builtin tree, showing all defined floating point builtins of MAT-LAB. All internal nodes are abstract builtins, the names inside the boxes refer to actual functions. The full tree showing all defined builtins is available at http://www.sable.mcgill.ca/mclab/tamer.html.

51

```
...

# operates on floating point matrizes
floatFunction; matrixFunction

# proper float functions have a fixed arity, and all operands are floats
properFloatFunction; floatFunction

# unary functions operating on floating point matrizes
unaryFloatFunction; properFloatFunction

# elemental unary functions operating on floating point matrizes
elementalUnaryFloatFunction; unaryFloatFunction
sqrt
realsqrt
erf

...

# float functions with optional arguments or variable number of arguments
improperFloatFunction; floatFunction

...
```

Figure 6.3 Excerpt of the builtin specification, showing definitions for some of the floating point functions shown in *Figure 6.2*. The lines starting with a #-symbol are comments.

The builtin framework takes a specification like shown in *Figure 6.3* as input, and generates a set of Java classes, one for each builtin function, whose inheritance relationship reflects the specified tree. For an abstract builtin, the generated Java class is abstract as well. The builtin framework (the code that generates Java files from the builtin specification) is written in Python.

6.2.1 Builtin Visitor Class

Besides the builtin classes, the builtin framework also generates a visitor class in Java. It allows adding methods to builtins and thus to define flow equations for them using the visitor pattern - a pattern that is already extensively used in the MCSAF analysis framework[Doh11]. In fact, flow analyses themselves are written using the visitor pattern.

52

The generated visitor class (see *Figure 6.4*) can be used to make flow analyses implement flow equations for all builtins. In order to do so, one has to derive from the visitor class and fill in the class variables used as argument and return values for the case methods. Overriding case methods allows specifying desired flow equations for the corresponding builtin.

Note that the default case for every builtin is to call the parent case - this means that to specify behavior for similar builtins, one only needs to specify the abstract behavior of a group, and the flow analysis framework will automatically apply the correct (most specialized) behavior for a specific builtin. This further motivates the structuring of builtin functions into a tree.

For example, we may find that for some flow analysis, all the flow equations for all functions that are in the group 'UnaryFloatFunction' are the same. So we just need to override the `caseAbstractUnaryFloatFunction()` method, shown in *Figure 6.4*. When executing any case-method of a builtin in that group, its default implementation will call the parent's implementation until reaching `caseAbstractUnaryFloatFunction()`.

The analysis framework allows specification of flow equations for all AST-nodes. Since all the MATLAB operators have associated AST-nodes, one can specify flow equations for operators using the analysis framework. Our set of builtin functions includes all the MATLAB operators, so analysis writer may alternatively define flow equations for operators using the builtin framework, rather than the analysis framework. For the analyses presented in this thesis we have opted to do so, to have fewer flow equations for AST-nodes, and have all the behavior of builtin functions in one place.

Using this approach, an intraprocedural analysis that is aware of builtins will consist of a flow analysis class defining flow equations for AST-nodes, and a class defining flow equations for builtin functions. Both are defined as extensions of visitor classes - the flow analysis is a visitor class for the AST-node hierarchy, and the builtin visitor for the hierarchy of builtins.

```java
public abstract class BuiltinVisitor<A,R> {
  public abstract R caseBuiltin(Builtin builtin,A arg);

  ...

  //operates on floating point matrizes
  public R caseAbstractFloatFunction(Builtin builtin,A arg){
    return caseAbstractMatrixFunction(builtin,arg);
  }

  //proper float functions have a fixed arity, and all operands are floats
  public R caseAbstractProperFloatFunction(Builtin builtin,A arg){
    return caseAbstractFloatFunction(builtin,arg);
  }

  //unary functions operating on floating point matrizes
  public R caseAbstractUnaryFloatFunction(Builtin builtin,A arg){
    return caseAbstractProperFloatFunction(builtin,arg);
  }

  //elemental unary functions operating on floating point matrizes
  public R caseAbstractElementalUnaryFloatFunction(Builtin builtin,A arg){
    return caseAbstractUnaryFloatFunction(builtin,arg); }
  public R caseSqrt(Builtin builtin,A arg){
    return caseAbstractElementalUnaryFloatFunction(builtin,arg); }
  public R caseRealsqrt(Builtin builtin,A arg){
    return caseAbstractElementalUnaryFloatFunction(builtin,arg); }
  public R caseErf(Builtin builtin,A arg){
    return caseAbstractElementalUnaryFloatFunction(builtin,arg); }

  ...

  //float function with optional arguments or variable number of arguments
  public R caseAbstractImproperFloatFunction(Builtin builtin,A arg){
    return caseAbstractFloatFunction(builtin,arg); }

  ...

}
```

Figure 6.4 Excerpt of the visitor class `BuiltinVisitor` that is generated by the builtin framework using the specification shown in *Figure 6.3*. The comments are copied from the specification file by the builtin framework.

54

6.3   Builtin Function Categories

We categorize the MATLAB builtin functions according to many properties, such as mclass, arity, shape, semantics. To minimize the number of flow equations that need to be specified for analyses and properties, they may require different kinds of groupings for the builtins, based on the semantics of the analyses or property. Ideally, for every analysis there should be categories grouping builtins, so that the fewest possible flow equations have to be specified.

In general this is not possible, because we are using a tree to categorize builtins. Nevertheless we attempted to find as many useful categories as possible, which are partly inspired by potential needs for analysis, and partly by the similarities of existing builtin functions, and the categories we found.

Another motivation for the heavy use of categories is that our framework does not yet implement all MATLAB builtin functions, and we want to minimize the amount of work required to add a builtin. When adding builtins that fit in already existing categories, one can reuse the attributes and flow equations specified for these categories.

Effectively, we have made a survey of all the builtins, learning about their semantics, interfaces and mclass-behavior, and have retrofitted them with an object-hierarchy. This approach seems natural because we do generate object-oriented Java code for the builtins, which uses that same hierarchy.

In the following we list the categories we have used to group functions. We present every category along with their alternatives; the alternatives are mutually exclusive. We use naming conventions that attempt to follow MATLAB terminology, but some may only be valid for the builtin framework.

pure, impure

> Pure functions have no side effects, change no state, internal or otherwise, and always return the same result when called with the same arguments.

matrix, cell, struct, object, versatile

> Matrix functions operate on MATLAB values that are numerical, `logical` or `char`.

55

all arguments, operands and results should have these mclasses. For example, numerical functions are matrix functions.

Cell functions operate on cell arrays, struct functions operate on structures, object functions operate on objects.

Versatile functions operate on multiple kinds of the above categories. Some may operate on any MATLAB value. For example, query functions like `numel` only depend on the shape of the argument - since every MATLAB value has a shape, the function works on all arguments.

anyMatrix, numeric, float

These categories are sub-categories of the matrix category.

A function belonging to the anyMatrix category operates on numerical, `logical` or `char` arrays. Numeric Functions operate on numbers. They may also accept `char` or `logical` values, but these values will be coerced to `double`, so the actual operation and the result will be numerical.

Float functions only operate on floats, i.e. `single` or `double` values. Some of the functions in this category may also accept different arguments and coerce them to `double`.

proper/improper

Proper functions have strict arity, and the arguments are operands. As can be seen in *Figure 6.5*, a lot of numeric functions are proper. Almost all operators are proper functions (an exception is the colon operator).

Improper functions may operate on a variable number of operands, or allow optional parameters. Some may accept (optional) parameters specifying options for the computation to be performed - these option parameters are not operands and may be of a type that functions within its category do not accept as operands.

For example, the float function `eps` (machine epsilon) is improper: it allows zero arguments or one floating point argument, but it also supports the `char` values `'single'` and `'double'` as a sole argument. The function will always return a float value.

unary, binary

> A unary function requires exactly one argument, a binary function requires exactly two.

elemental, array

> The elemental category refers to element-wise functions, i.e. functions which operate on every element in an array independently. The result will have the same shape as the inputs. The array functions operate on the whole array at once. For example matrix multiplication belongs to the array category.
>
> The notion of elemental and array functions corresponds to MATLAB's notion of array vs matrix operators, introduced in Sec. **??**. Note the different terminology to avoid re-using the term 'matrix'.

dimensionSensitive   Dimension-sensitive functions are of the form $f(M, [dim])$, i.e. they take some array as the first argument, and allow a second optional argument `dim`. This argument specifies the dimension along which to operate. By default the dimension will be the first non-singleton dimension.

dimensionCollapsing

> A dimension-collapsing function is a dimension-sensitive function which will collapse every value along the operated dimension into one value, and return a new matrix with a corresponding shape. For example the `sum` function sums all values along the dimension it operates, turning them into single values. Other examples are the functions `prod`, `mean`, `mode`, `min` and `max`.

query

> A query is a function that given some arguments, will return a scalar or a vector containing information about the argument(s). The computation summarizes the information contained in the arguments in some fashion.

toLogical, toDouble

> These categories refer to the mclass of the result of the computation. We use these

as sub-categories of query. functions in the toDouble category will always return a `double` result, functions in the toLogical category will return `logical` results.

Besides the above general categories, we use ad hoc ones that attempt to group builtin functions according to their semantics, i.e. functions performing similar computation should be grouped together. For example in *Figure 6.5*, there are categories like 'trigonometric function' or 'factorization function'.

Within the tree-structure, categories are combined, creating more and more refined categories. For example, going down the tree one can reach the combination of categories termed `ElementalBinaryToLogicalMatrixQuery`. Functions in this combined category refer to query functions operating on matrices only, which take exactly two arguments, operate element-wise and will return values of mclass logical. The proliferation of these long names may explain some of our naming conventions, which are largely motivated by the desire for brevity, to keep combined categories manageable.

An example of a complete path along the builtin tree, showing further and further refinement of categories, is shown in *Figure 6.5*. It also shows alternative categories along the path.

6.4 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 attributes.

In the builtin specification language, an attribute is just a name, with a set of arguments that follow it. In the specification language the attributes are defined on the same line as the builtin itself. Starting with the third value, every value specifies an attribute. Internally we call attributes to builtins 'tags'.

A specific attribute 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.

Figure 6.5 An example showing all ancestors of a group of builtins, and all siblings for all these ancestors. This shows the refinement of categories from the top category of 'builtin' going to a specific builtin, and what the alternative categories are along the way.

It is possible to add new kinds of attributes to the builtin specification language. One merely has to provide a function [2] with a specific function interface that provides information about the specified builtin and the argument string for the attribute. The function has to return Java code that will be inserted in the generated Builtin class. The function may also update a list of interfaces that the generated builtin class implements. The name of that function is the name of the attribute as used in the builtin specification language. The argument to the attribute is an arbitrary string. It may, however, not contain a semicolon, because it is used to match the end of the attribute.

6.5   The Class and MatlabClass attribute

In order to build a complete callgraph, we need to know of what mclass a variable may be during runtime, due to the overloading lookup semantics introduced in Sec. **??**. To have complete knowledge of all possible mclasses for all variables at all times, we need to know how they behave with respect to mclasses. We opted to define all this information as attributes to builtins, defined in the builtin specification along with builtins themselves.

We defined an attribute called `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 object.

The mclass flow equation object itself is defined in the builtin specification as an argument to the `Class` attribute, using a small domain specific language that allows matching argument mclasses. It returns result mclasses based on matches. We have decided to build this little domain specific language because of the complexity of some builtins, and our desire to define mclass flow equations in a compact way.

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 added the `MatlabClass` attribute. It allows us to specify the exact MATLAB semantics - and thus provides an exact definition and documentation of MATLAB class semantics. Refer to *Figure 6.6* for an example usage of both a `Class` attribute and a `MatlabClass` attribute

---

[2]attribute functions are defined in processTags.py in the builtin framework

60

```
...

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

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

...
```

Figure 6.6 Excerpt of the builtin specification with the Class and MatlabClass attributes added
in. The Class attribute for `unaryNumericFunction` defines the mclass flow
equations for unary functions taking numeric arguments, and applies for all builtins
in the group. It specifies that given a numeric argument, the result will have the
same mclass (`numeric>0`). For `char` and `double` the result will be a double.
Note the `MatlabClass` attribute defined for `conj` and `sign`. These functions
have exact MATLAB semantics that differ from the default used by the builtin frame-
work: they disallow `logical` arguments (but not `char` arguments), using them will
result in an error.

showing slightly different behavior.

A detailed description of the domain-specific language used to represent mclass flow
equations is presented in *Appendix* **??**.

6.6 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. We have used
our analysis of the builtins to organize builtin functions into a tree structure, making it
easier to work with builtin functions.

This builtin framework is extensible both by allowing the quick addition of more builtin

functions; and by allowing to specify more information and behavior for builtin functions. This can be done either adding new properties to the framework itself; or by implementing visitor classes.

The compact representation of builtins also allows changing the organization of builtins. This means that the whole framework may evolve as our understanding of builtin functions and our requirements for analyses evolve.[3]

---

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

Chapter 7
Analyses

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 **abstract value analysis** (section 7.1), 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 7.3). 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 7.4). 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 9.

7.1   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).

7.1.1   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:

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

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:

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

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.

7.1.2 Flow Sets:

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

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

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*.

7.1.3   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 join-

ing 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.

7.1.4 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.

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

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.

7.2 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

- $var_t \ = \ var_s$: $f'_p = f_p \setminus \{(var_t, f_p(var_t))\} \cup \{(var, f_p(var_s))\}$

- $var \ = \ l$, where l is a literal with mclass c_l and value representation v_l:

$$f'_p = f_p \setminus \{(var, f_p(var))\} \cup \{(var, \{(c_l, v_l)\})\}$$

- $[t_1, t_2, \ldots, t_m] \ = \ func(a_1, a_2, \ldots, a_n)$, 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 behavior of builtin functions.

7.3 Structures, Cell Arrays and Function Handles

We implemented a value abstraction for structs, cell arrays and function handles (internally called `AggrValue`). 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 one of the integer mclasses). 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 the abstraction of composite values to implement a concrete value analysis.

7.3.1 `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 op-

eration 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.

7.3.2 `function_handle`:

As explained in section **??**, 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 4.3) 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.

7.4 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.

Chapter 8
Code generation

There are several categories of related work. First, we have the immediate work upon which we are building. The *Mc**LAB*** project already provided the front-end and the McSAF [Doh11] analysis framework, which provided an important basis for the Tamer. Then there is McFOR, a previous attempt to build a static compiler targeting FORTRAN95, that is part of the *Mc**LAB*** project. There are also other compilers for MATLAB, both static ones and dynamic ones. There is also related work on statically analyzing and compiling other dynamic languages, with some similar problems we have faced, and some similar approaches. Some of this work is presented in section Sec. 10.4.

8.1 McFOR

We learned a lot from *Mc**LAB***'s previous McFOR project[Li09] which was a first prototype MATLAB to FORTRAN95 compiler. McFOR supported a smaller subset of the language, and simply ignored unsupported features - leading to possibly undefined behavior. McFOR did also not have a comprehensive approach to the builtin functions, did not support the MATLAB function lookup semantics, 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 particular it showed that the numerical and matrix features of FORTRAN95 are a good match for compiled MATLAB,

and that the static nature of the language, together with powerful FORTRAN95 compilers provide the potential for high performance.

We have developed the Tamer with targeting FORTRAN95 in mind. In order to provide some extra flexibility for other potential backends we have restricted MATLAB less than may be necessary for a MATLAB to FORTRAN compiler, i.e. it may have to restrict the MATLAB language further. For example, FORTRAN95 has very limited polymorphism support, meaning that any polymorphic code can not be easily translated to compact and readable FORTRAN. MCFOR does observe these limitations, but does have an interesting way to deal with one polymorphic case: If an if-statement results in incompatible types for a variable along both branches, the code following that if-statement gets copied into both branches, so that there won't be a confluence of incompatible types. For example,

```
if (...)
  x = 3
else
  x = 'Hi'
end
foo(x)
```

may be converted to

```
if (...)
  x = 3
  foo(x)
else
  x = 'Hi'
  foo(x)
end
```

This transformation is not possible in general for confluence points around loop statements, and does also not work if values with ambiguous types are returned from a function.

Despite being a full compiler with many interesting ideas, MCFOR is a prototype, with limited feature set and limited extensibility. For this thesis we have gone back to the ba-

sics 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.

8.2 Other Static MATLAB compilers

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 [RP99] 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 in-lining all of the the code. MaJIC[AP02], a MATLAB Just-In-Time compiler, is patterned after FALCON. It uses similar type inference techniques to FALCON, but are simplified to fit the JIT context. MAGICA [JB03, JB01] 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 [Joi03]. 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 *MATLABCoder* recently released by MathWorks[Mat]. 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.

8.3 Other Matlab-**like systems**

There are other projects providing open source implementations of Matlab-like languages, such as Octave[Oct] and Scilab[INR09]. 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. In particular Octave may present opportunities to improve the usefulness of our static compiler framework without requiring an actual Matlab installation. Octave, being an interpreter system, may not provide very high performance, but it does include a large library similar to Matlab's library. Enabling our framework to support Octave's specific Matlab flavor may help bring together Octave's completeness with the potential performance gains of a static compilation framework.

8.4 Static Approaches to other Dynamic Languages

Other dynamic languages have had very successful efforts in defining static subsets in order to provide static analysis.

8.4.1 Python

Reduced Python (RPython)[AACM07] provided inspiration for our approach at dealing with a dynamic language in a static way. Rather than attempting to support dynamic features that are not amenable to static compilation, for example by providing interpreter-like features as a fallback, RPython restricts ("reduces") the set of allowable features such that programs are statically typable. At the same time, it attempts to stay as expressive as possible.

RPython was originally developed for PyPy, a Python interpreter written in Python, but has evolved into be a general purpose language. It was not developed to compile programs completely statically, but rather with the goal to speed up execution times in virtual machines like VM or CLI, which are themselves developed for static languages (Java and C#, respectively).

Besides disallowing dynamic features, RPython disallows a basic feature of many dynamic programing languages: at a confluence point, a variable may not be defined with two incompatible types. This notion that a variable should have one specific type at every programming point is something that we expect for static backends of our framework as well, in particular for FORTRAN, even if the Tamer Framework itself actually supports union types. Both RPython, as well as our own research have indicated that this restriction is not a serious limitation in practice.

RPython restricts Python's container types. In particular, it forces that dictionaries (hash-tables) and arrays are homogeneous, i.e. all elements have the same type. Tuples are allowed to be inhomogeneous. For the Tamer, we represent the two builtin container types (structs, cells) in both possible ways: as a tuple or as a collection, which correspond to inhomogeneous and homogeneous representations, respectively.

RPython does not directly support generic functions, i.e. if a function is used multiple times with incompatible arguments, the program gets rejected. The Tamer uses a context-sensitive interprocedural analysis that creates copies of functions when they are called with incompatible arguments.

8.4.2 Ruby

DiamondbackRuby (DRuby) is a static type inference toolkit for Ruby [FAFH09], mostly with the goal to gain the advantage of static languages to report potential errors ahead of time. Ruby, like MATLAB, is a dynamic, interpreted language, but is used more in web development. Some of the approaches of DRuby are similar to the Tamer framework.

Similar to MATLAB, the core library of Ruby is written in native code (i.e. in C), rather than Ruby itself - which may also have different behaviors depending on the incoming argument types. Thus DRuby has to provide type information for builtin functions. In order to that, DRuby includes a type annotation language, which can also be used to specify types for functions with difficult behavior. Note that at this point, the focus of our builtin framework is to organize the large number of builtins, but our work may lead to a proper type annotation language as well.

DRuby also provides a type inference, but it is based on a constraint-based analysis.

DRuby constrains the set of supported language features to enable the static analysis, but allows some of them by inserting runtime checks to still be able to support them. These are included in such a way as to help users identify where exactly the error occurred.

Using the results of the static analyses provided by the MATLAB Tamer to provide information about potential runtime errors is one of the possible goals of continued research.

Chapter 9
Evaluation

In order to exercise the framework, we applied it to the set of benchmarks we have previously used for evaluating McVM/McJIT[LH11], a dynamic system. The benchmarks and results are given in *Table 9.1*. About half of the benchmarks come from the FALCON project[RP99] and are purely array-based computations. The other half of the benchmarks were collected by the *Mc*LAB 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. For functions like `load`, which can return arbitrary values, we may have to provide alternative, more "tame" versions in order to produce a tamed program. 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.

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

Name	Description	Source	#Fn	#BFn	Features	Wild	Mclass
adpt	*Adaptive quadrature*	Numerical Methods	1	17		no	unique
beul	*Backward Euler*	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 9.1 Results of Running Value Analysis

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

Chapter 10
Related work

There are several categories of related work. First, we have the immediate work upon which we are building. The *Mc***LAB** project already provided the front-end and the MCSAF [Doh11] analysis framework, which provided an important basis for the Tamer. Then there is MCFOR, a previous attempt to build a static compiler targeting FORTRAN95, that is part of the *Mc***LAB** project. There are also other compilers for MATLAB, both static ones and dynamic ones. There is also related work on statically analyzing and compiling other dynamic languages, with some similar problems we have faced, and some similar approaches. Some of this work is presented in section Sec. 10.4.

10.1 MCFOR

We learned a lot from *Mc***LAB**'s previous MCFOR project[Li09] which was a first prototype MATLAB to FORTRAN95 compiler. MCFOR supported a smaller subset of the language, and simply ignored unsupported features - leading to possibly undefined behavior. MCFOR did also not have a comprehensive approach to the builtin functions, did not support the MATLAB function lookup semantics, 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 particular it showed that the numerical and matrix features of FORTRAN95 are a good match for compiled MATLAB,

and that the static nature of the language, together with powerful FORTRAN95 compilers provide the potential for high performance.

We have developed the Tamer with targeting FORTRAN95 in mind. In order to provide some extra flexibility for other potential backends we have restricted MATLAB less than may be necessary for a MATLAB to FORTRAN compiler, i.e. it may have to restrict the MATLAB language further. For example, FORTRAN95 has very limited polymorphism support, meaning that any polymorphic code can not be easily translated to compact and readable FORTRAN. MCFOR does observe these limitations, but does have an interesting way to deal with one polymorphic case: If an if-statement results in incompatible types for a variable along both branches, the code following that if-statement gets copied into both branches, so that there won't be a confluence of incompatible types. For example,

```
if (...)
  x = 3
else
  x = 'Hi'
end
foo(x)
```

may be converted to

```
if (...)
  x = 3
  foo(x)
else
  x = 'Hi'
  foo(x)
end
```

This transformation is not possible in general for confluence points around loop statements, and does also not work if values with ambiguous types are returned from a function.

Despite being a full compiler with many interesting ideas, MCFOR is a prototype, with limited feature set and limited extensibility. For this thesis we have gone back to the ba-

sics 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.

10.2 Other Static MATLAB compilers

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 [RP99] 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 in-lining all of the the code. MaJIC[AP02], a MATLAB Just-In-Time compiler, is patterned after FALCON. It uses similar type inference techniques to FALCON, but are simplified to fit the JIT context. MAGICA [JB03, JB01] 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 [Joi03]. 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 *MATLABCoder* recently released by MathWorks[Mat]. 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.

10.3 Other MATLAB-like systems

There are other projects providing open source implementations of MATLAB-like languages, such as Octave[Oct] and Scilab[INR09]. 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. In particular Octave may present opportunities to improve the usefulness of our static compiler framework without requiring an actual MATLAB installation. Octave, being an interpreter system, may not provide very high performance, but it does include a large library similar to MATLAB's library. Enabling our framework to support Octave's specific MATLAB flavor may help bring together Octave's completeness with the potential performance gains of a static compilation framework.

10.4 Static Approaches to other Dynamic Languages

Other dynamic languages have had very successful efforts in defining static subsets in order to provide static analysis.

10.4.1 Python

Reduced Python (RPython)[AACM07] provided inspiration for our approach at dealing with a dynamic language in a static way. Rather than attempting to support dynamic features that are not amenable to static compilation, for example by providing interpreter-like features as a fallback, RPython restricts ("reduces") the set of allowable features such that programs are statically typable. At the same time, it attempts to stay as expressive as possible.

RPython was originally developed for PyPy, a Python interpreter written in Python, but has evolved into be a general purpose language. It was not developed to compile programs completely statically, but rather with the goal to speed up execution times in virtual machines like VM or CLI, which are themselves developed for static languages (Java and C#, respectively).

Besides disallowing dynamic features, RPython disallows a basic feature of many dynamic programing languages: at a confluence point, a variable may not be defined with two incompatible types. This notion that a variable should have one specific type at every programming point is something that we expect for static backends of our framework as well, in particular for FORTRAN, even if the Tamer Framework itself actually supports union types. Both RPython, as well as our own research have indicated that this restriction is not a serious limitation in practice.

RPython restricts Python's container types. In particular, it forces that dictionaries (hash-tables) and arrays are homogeneous, i.e. all elements have the same type. Tuples are allowed to be inhomogeneous. For the Tamer, we represent the two builtin container types (structs, cells) in both possible ways: as a tuple or as a collection, which correspond to inhomogeneous and homogeneous representations, respectively.

RPython does not directly support generic functions, i.e. if a function is used multiple times with incompatible arguments, the program gets rejected. The Tamer uses a context-sensitive interprocedural analysis that creates copies of functions when they are called with incompatible arguments.

10.4.2 Ruby

DiamondbackRuby (DRuby) is a static type inference toolkit for Ruby [FAFH09], mostly with the goal to gain the advantage of static languages to report potential errors ahead of time. Ruby, like MATLAB, is a dynamic, interpreted language, but is used more in web development. Some of the approaches of DRuby are similar to the Tamer framework.

Similar to MATLAB, the core library of Ruby is written in native code (i.e. in C), rather than Ruby itself - which may also have different behaviors depending on the incoming argument types. Thus DRuby has to provide type information for builtin functions. In order to that, DRuby includes a type annotation language, which can also be used to specify types for functions with difficult behavior. Note that at this point, the focus of our builtin framework is to organize the large number of builtins, but our work may lead to a proper type annotation language as well.

DRuby also provides a type inference, but it is based on a constraint-based analysis.

DRuby constrains the set of supported language features to enable the static analysis, but allows some of them by inserting runtime checks to still be able to support them. These are included in such a way as to help users identify where exactly the error occurred.

Using the results of the static analyses provided by the MATLAB Tamer to provide information about potential runtime errors is one of the possible goals of continued research.

Chapter 11
Conclusions and Future Work

This thesis 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, and should help motivate some of the restrictions we impose on the language. 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 behavior. We defined a Tame IR and added functionality to MCSAF to produce the IR and to extend the analysis framework to handle the new IR nodes introduced. We provided an interprocedural analysis framework that allows creation of full-program analyses of MATLAB programs. Finally, we developed an extensible value estimation analysis that we use to provide a call-graph constructor for MATLAB programs, using the proper lookup semantics, starting with some entry point.

11.1 Future Work

Our initial experiments with the framework are very encouraging and there are several possible projects to continue the development of static compilers for MATLAB as part of

the *Mc***LAB** project. We also hope that others will also use Tamer the framework for a variety of static MATLAB tools.

In the following we will present some ideas for the continued development of the static portion of the *Mc***LAB** framework.

The major goal of the Tamer Framework is to provide a starting point for compiler backends targeting static programming languages. In particular, we have developed our toolkit with compilation targeting FORTRAN95 in mind. In order to actually be able to compile, the abstract value representations need to be further refined, and the value analysis extended. In particular, shape information for arrays is needed, which may be dealt with in a similar way as the mclass information. Further refinement of the value representations can improve the supported feature set and performance. For example, having exact knowledge whether numerical values may be real, complex or imaginary allows using complex data types only when necessary, rather than using complex numbers by default for all values. Advanced analyses could be used to the relationships of array-shapes and values of variables, enabling the removal of run-time array bounds checks. This may provide significant performance benefits.

Further work may focus around expanding the set of supported MATLAB features. Interesting may be the extension of the Tamer framework to fully support MATLAB user-defined classes, including the "old" semantics, the "new" semantics since version 7.6, and possibly handle-classes. The MATLAB Tamer already supports the notions of mclasses, and the overloading semantics necessary to implement class semantics are already supported. Note that in order to support handle-classes, it is not sufficient to extend the value representations - the machinery of the analysis also has to be extended to capture changes of arguments that use the reference semantics of handle-classes. This is also true if the Tame MATLAB language subset was extended to support global and persistent variables.

The Tamer framework could work together with *Mc***LAB**'s refactoring tools in two ways. For one it would be possible to use the refactorings as code transformations in a pre-processing step, to be able to reduce/refactor some unsupported dynamic feature of MATLAB. For example, the refactoring toolkit allows transforming MATLAB scripts into MATLAB functions. Another way the refactoring tools could work together with the Tamer framework is in an interactive fashion. A user wishing to compile a program may find that

the Tamer rejects it; the refactoring toolkit could then step in and suggest to refactor the program in certain ways to make it possible to compile.

Future work may advance the static compilation framework and the notion of bridging the gap between dynamic languages and static analyses and compilation. The builtin framework with its approach to allow the explicit and compact definition of flow information for functions may lead to a general type annotation language for MATLAB types, which could be used both to type builtin functions, or to type user and library functions with complex behavior. Static information provided by full-program analyses using these frameworks could be used to find potential runtime errors, and aid programmers build better and more correct programs.

Appendix A

XML structure for builtin framework

In the following we provide a list of MATLAB builtin functions. The corresponding numbers show how many callsites there are for the function within the large set of MATLAB programs that can be analyzed by the MCBENCH framework.

When selecting the initial set of builtin functions for the builtin framework, we use most of the below functions, excluding dynamic and GUI functions. We added functions corresponding to MATLAB operators, as well as some functions that are very closely related to functions in the list.

1	recycle	1	dmperm	2	ge
1	schur	1	fileattrib	2	uint64
1	gt	1	delaunay	2	atanh
1	mislocked	1	isequalwithequalnans	2	ishghandle
1	acotd	1	javaMethod	2	cotd
1	more	1	functions	2	isdeployed
1	acot	1	structfun	2	le
1	uminus	2	subsasgn	2	prefdir
1	uitoolbar	2	rehash	3	isstrprop
1	dbclear	2	ne	3	dragrect
1	isjava	2	linsolve	3	uitoggletool
1	ordschur	2	regexptranslate	3	ferror
1	munlock	2	memory	3	javaArray
1	superiorto	2	uitable	3	javaObject
1	unicode2native	2	matlabpath	3	sec
1	methods	2	exit	3	hgconvertunits

Table A.1 List of builtins and their frequency of occurrence (continued on the following pages)

3	pack	11	rmdir	27	rmappdata
3	sortrowsc	11	libisloaded	27	cumprod
3	lt	11	isletter	27	struct2cell
3	echo	11	cast	28	isfloat
4	validatestring	11	unloadlibrary	29	nnz
4	tand	11	evalc	29	bitget
4	dbstop	12	waitfor	29	uint32
4	int64	12	power	29	ftell
4	csc	12	isvarname	29	cosh
4	hardcopy	12	loglog	30	surface
4	uipushtool	12	pow2	31	waitforbuttonpress
4	asinh	13	convhull	31	fgets
4	asind	13	mexext	31	realsqrt
5	native2unicode	13	speye	33	rectangle
5	erf	13	vertcat	33	arrayfun
5	who	13	int16	34	tanh
5	hggroup	13	getenv	34	bitshift
5	rbbox	14	func2str	34	semilogy
5	bitor	14	acosd	34	nargoutchk
5	erfinv	14	lasterror	35	asin
5	lu	14	movefile	35	mkdir
5	colstyle	15	isspace	35	int32
5	im2frame	15	isinteger	36	textscan
6	frame2im	15	randi	36	svd
6	ifftn	15	horzcat	36	sinh
6	hgtransform	16	gammainc	36	lasterr
6	diary	16	regexpi	36	computer
6	subsref	17	accumarray	38	version
6	erfcinv	17	dbstack	40	cosd
6	rcond	17	hypot	41	xor
6	home	17	isappdata	43	sind
7	uicontextmenu	17	or	44	beep
7	cot	18	whos	45	triu
7	reset	19	unix	45	gammaln
7	eq	19	tril	47	copyobj
7	sech	19	inputname	49	fill3
7	builtin	19	copyfile	49	islogical
8	type	19	qr	51	semilogx
8	setstr	20	cell2struct	51	histc
8	validateattributes	20	isobject	55	uint16
8	what	23	ancestor	55	gamma
8	atand	23	handle	55	system
9	issorted	24	issparse	56	uipanel
9	acosh	24	bitset	60	dos
9	int8	25	and	60	transpose
9	light	25	lastwarn	61	complex
9	betainc	26	bitand	62	format
10	keyboard	26	nonzeros	66	acos
11	rethrow	26	matlabroot	67	erfc
11	fftn	26	chol	68	calllib
11	feof	27	typecast	69	strncmpi

70	strtrim	210	cumsum	590	real
74	det	213	ishandle	607	fread
76	atan	213	rem	623	toc
76	import	214	nargchk	633	warning
82	isstruct	222	fft	682	ceil
82	sscanf	228	sign	683	axes
85	isstr	228	cd	685	char
86	not	235	j	696	sort
90	full	240	str2func	703	clear
91	strncmp	245	input	707	eval
92	eig	256	prod	757	ischar
93	log2	260	isreal	766	mod
93	regexprep	265	clock	768	log
94	permute	272	randn	777	double
95	which	276	getappdata	812	true
96	class	281	uint8	863	reshape
97	tan	284	iscell	933	any
101	isinf	286	eye	949	false
103	bitxor	294	setappdata	983	gca
105	upper	303	uimenu	1101	numel
105	ifft	314	line	1109	floor
105	isvector	321	strrep	1214	exp
106	fieldnames	325	display	1235	figure
109	fill	335	load	1240	nargout
114	filter	337	diag	1265	round
117	cputime	344	log10	1332	uicontrol
119	fseek	346	drawnow	1471	isequal
124	assert	349	fix	1618	sprintf
125	image	349	findstr	1682	ones
131	bsxfun	352	lower	1760	sin
131	regexp	352	nan	1761	cos
131	conv2	354	isnumeric	1945	strcmp
137	evalin	356	eps	2117	find
137	assignin	356	pause	2237	plot
142	sparse	365	cell	2337	sqrt
146	dir	376	struct	2405	min
154	clc	378	isfield	2630	abs
157	logical	397	strcmpi	2797	sum
158	isscalar	406	fopen	3304	pi
164	cat	409	imag	3378	max
166	patch	410	all	3539	fprintf
170	isfinite	419	conj	3866	isempty
171	deblank	428	norm	4059	zeros
173	plot3	445	tic	4131	nargin
174	atan2	457	fclose	4555	single
176	cellfun	510	delete	4965	error
180	feval	524	isnan	6731	size
183	fscanf	547	mfilename	7031	disp
183	inv	555	isa	7768	length
193	save	559	inf	8379	get
199	strfind	563	text	11460	set
204	fwrite	564	exist	13880	rand
204	ndims	583	diff		

Appendix B

isComplex analysis Propagation Language

B.1 Introduction

In the following, we define the tiny language used to define how mclasses propagate through builtins. This is used in the builtin specification to specify mclass propagation, using the `Class` attribute. To specify the mclass propagation for a builtin, or an abstract builtin, it is specified as an attribute in the builtin specification, using the syntax

$$Class(< expr >)$$

or

$$Class(< expr_1 >, < expr_2 >, < expr_3 >, \ldots, < expr_n >)$$

where the expressions follow the syntax of the mclass propagation language. It is also possible to separate the cases using the || operator:

$$Class(< expr_1 > || < expr_2 > || < expr_3 > || \ldots || < expr_n >)$$

The language itself is somewhat similar to regular expressions in that it matches incoming mclasses. But rather than having a match as a result, the language allows to explicitly state what the result is.

B.2 Class Specification

B.2.1 Basics

Every expression is interpreted either in LHS (left-hand side) or in RHS (right-hand side) mode. In LHS mode it matches the mclasses of arguments to the expression, in RHS it emits the mclasses as output. For example

$$double->char$$

will attempt to match a *double* argument, and if one is found, it will emit a *char* result. For any expression *iff* all input arguments have been consumed by matching, it will result in an overall match, and the emitted results will be returned.

At any point there will be a partial match, which consists of the next input argument index to be read, and the result mclasses emitted so far. For example, after the above expression, if one attempts to match the input arguments $[double, double]$, the partial result will refer to the 2nd argument, and will have *char* as an output.

B.2.2 Language Features

In the following, we present the syntax and semantics of the language, showing the LHS (matching) and RHS (emitting) semantics for every feature. Note that for some expressions, the LHS and RHS semantics are the same, i.e. some expressions may ignore the current mode.

Operators

$expr_1->expr_2$

    LHS, RHS: will attempt to match $expr_1$ as a LHS, and if it is a match, will execute $expr_2$ as a RHS expression and emit the results.

$expr_1\ expr_2$

    LHS: will attempt to match $expr_1$ as a LHS, and if it is a match, will attempt to match

expr₂.

RHS: will emit the results of *expr₁* in RHS mode, then will emit the results of *expr₂* in RHS mode.

expr₁|*expr₂*

LHS: will attempt to match both *expr₁* and *expr₂* independently, then will return whichever successful match consumed the most arguments.

RHS: will emit the union of the emitted results of *expr₁* and *expr₂*, run as RHS expressions. Both *expr₁* and *expr₂* must have the same number of emitted result. If not, it will throw a runtime error.

expr?

expr? is equivalent to *none*|*expr*

LHS: will attempt to match the expression. If does not match, ? will still return a match successfully, but it does so by matching *none*.

RHS: This will likely cause an error, because the union of two match results must both result in the same number of emitted outputs.

expr∗

expr∗ is equivalent to *expr*? *expr*? *expr*?...

The operators ∗ and ? have the same, highest precedence, | has a lower precedence, putting no symbol (i.e. *expr₁* *expr₂*) has a lower precedence than that. -> has the lowest precedence.

Non-parametric Expressions

Builtin MClasses

The mclass propagation language supports the following builtin mclasses and groups of mclasses:

double	*logical*
float	*function_handle*
char	*int8*
uint8	*int16*
uint16	*int32*
uint32	*int64*
uint64	

LHS: will attempt to match the builtin mclass

RHS: will emit the builtin mclass

Groups of Builtin Mclasses

Certain mclasses are grouped together using union:

float	is the same as	*single*|*double*
uint	is the same as	*uint8*|*uint16*|*uint32*|*uint64*
sint	is the same as	*int8*|*int16*|*int32*|*int64*
int	is the same as	*uint*|*sint*
numeric	is the same as	*float*|*int*
matrix	is the same as	*numeric*|*char*|*logical*

Non-parametric Language Features

none

> LHS, RHS: matches without consuming inputs or emitting results

begin

> LHS, RHS: will match if the next argument is the first argument (no arguments have been matched)

end

> LHS, RHS: will match if all arguments have been matched

96

any

> LHS: will match the next argument, no matter what it is, if there is an argument left to match
>
> RHS: error

parent

> LHS, RHS: will substitute the expression that is defined for the abstract parent builtin. If the parent builtin does not define mclass propagation information, will substitute *none*.

error

> LHS, RHS: same as none, except that the result is flagged as erroneous. During matching *error* is ignored (partial matching will continue), but if a result is erroneous overall, it will result in not a match overall.

natlab

> LHS, RHS: Besides the `Class` attribute for builtins, one can define an alternative attribute `MatlabClass` which more closely resembles MATLAB semantics, including some of the irregularities of the language. When defining such a `MatlabClass` attribute, the keyword *natlab* will refer to the expression defined by the `Class` attribute. Note that one cannot define a `MatlabClass` without defining a `Class` attribute, so *natlab* should always be defined.

matlab

> equivalent to *natlab*, but this can be used inside `Class` attribute to refer to whatever is defined for the `MatlabClass` attribute.
>
> This does not verify whether the `MatlabClass` attribute has been defined; therefore, undefined behavior may result if the attribute is not defined.

scalar

> LHS: If there is another argument to consume, matches if it is scalar, or if its shape is unknown, without consuming the argument. This can be used to check if the next argument is scalar. This should only be used if the scalar requirement is directly related to mclass behavior. If shapes and types are independent, they should be specified independently.

RHS: runtime error

Functions

coerce(*replaceExpr, expr*)

will take every single argument, and execute *replaceExpr* on it individually and independently. The replace expression must either not match, or a match and emit a single result. If it does, this result gets replaced as a the new argument. *coerce* will then take the new set of arguments, and execute *expr* with it, either as LHS or RHS depending on whether the *coerce* itself was executed in LHS or RHS mode, and return the result of that.

This allows operand coercion. For example, a function may convert all incoming *char* or *logical* arguments to *double*, which would be done using

$$coerce(char|logical\text{-}{>}double, expr)$$

typeString(*expr*)

LHS, RHS: if the next element is a *char*, *typeString* will consume it. If its actual runtime value is known, it will check whether the value of the string is the name of a mclass which is emitted by *expr* (running *expr* in RHS mode). If it is, *typeString* will emit that mclass.

If the *char* has another known value, *typeString* will return an error.

If the value is not known, will emit all results produced by *expr*. *expr* should produce one (union) result.

This can be used to match a last optional argument denoting a desired mclass for the return value. This used, for example, by the functions `ones` and `zeros`, which allow a last optional argument specifying that the result should have a numerical mclass other than the default *double*.

Number

< *number* >

LHS, RHS: Equivalent to the input argument with the same index as the given num-

98

ber. For example, 0 will match (LHS) or emit (RHS) the mclass of the first argument. Negative numbers will match from the back, so -1 is the mclass of the last argument.

B.3 Extra Notes on Semantics

B.3.1 RHS Can Have LHS Sub-expressions, and Vice Versa

An expression may emit results even if it is run as a LHS expression, and an expression run as RHS may match more elements. For example for

$$double\text{-}>(char\text{-}>logical\ int16)$$

the *char* expression will get matched, due to the second -> operation. Similarly,

$$(double\ char\text{-}>logical)\text{-}>int16$$

being an equivalent expression, will emit the *logical* because of the second ->.

B.3.2 Overall Evaluation of Class Attribute Expressions

Overall, expressions are evaluated as LHS expressions, so the builtin attribute

$$\texttt{Class(double)}$$

will attempt to match a incoming *double* argument, and have no returns. Multiple arguments to the `Class` attribute get transformed internally to their union, so

$$Class(expr_1, expr_2, .., expr_n)$$

is equivalent to

$$Class(expr_1|expr_2|..|expr_n)$$

This only applies for arguments to the `class` attribute , comma is not an operator equivalent to | in general.

B.3.3 Greedy Matching

While the language looks similar to regular expressions, it is in fact different: all matching is done greedily. So the expression

$$(double|none)\ (double)$$

When run on a single *double* input, will not match. This is because the union will greedily match the longest expression, which will consume the input argument.

B.4 Examples

Class(*double*|*single double*|*single->double*)
> will match two floats, and result in a *double*.

Class(*coerce*(*char*|*logical->double*, *numeric->*0))
> will convert any *char* and *logical* arguments to *double*, then will match any single *numeric* argument, and emit the type of that argument.

Class(*char char->char*, *numeric* 0|*double->*0, *double*|1 *numeric->*1)
> Either two *chars* will result in a *char*, or, if two arguments are *numeric*, they should either have the same mclass or at least one argument has to be a *double*, in which case it will return the mclass of the other argument.

Class(*none->double*)
> If there are no inputs, will result in a *double* (i.e. this models a double constant).

Class(*parent any*?)
> Matches whatever the parent builtin matches, but will allow for one extra argument with any mclass.

MatlabClass(*char*|*logical*1*->error*, *natlab*)
> This will define separate semantics for MATLAB, compared to natlab. The example

specifies that MATLAB will reject any input that is either two *chars* or two *logicals*, but use the original natlab definition other than that.

Class(*numeric* ∗ (*typeString*(*numeric*)|(*none->double*)))

Will match any number of *numeric* arguments. If the last argument is a *char*, will attempt to interpret it as a string denoting a numeric type. if it is, return that numeric type. if it is a string of unknown value, return all *numeric*. if it's another string, return an error. if the last argument is not a string, return a *double*. This can be used for function calls like `ones(3,3)` or `ones(2,2,4,4,'int8')`

B.4.1 Grammar

Below is the complete grammar of the class propagation language. The overall goal is to produce a node 'cases'.

```
%terminals NUMBER, LPAREN, RPAREN, OROR, OR, COMMA, MULT, QUESTION, ARROW, ID;
%terminals COERCE, TYPESTRING;


%left  RPAREN;
%left  MULT, QUESTION;
%left  OR;
%left  CHAIN;
%left  ARROW;
%left  COMMA;
%left  OROR;


cases
    = list
    ;
list
    = expr
    | expr COMMA list
    ;
expr
    = clause ARROW clause
    | expr   OROR   expr
    | clause
    ;
clause
    = clause QUESTION
    | clause MULT
    | clause clause @ CHAIN
    | clause OR clause
    | NUMBER
    | ID
    | LPAREN expr RPAREN
    | COERCE LPAREN expr COMMA expr RPAREN
    | TYPESTRING LPAREN expr RPAREN
    ;
```

Appendix C

$\mathrm{M}_{\mathrm{I}}\mathrm{X}10$ **IR Grammar**

In this appendix we present a grammar for the Tame IR. We have listed all Tame IR nodes, together with the parent class and the nodes they contain. An informal, but more detailed, discussion can be found in *Chapter* **??**. All Tame IR nodes either extend AST nodes, or other IR Nodes.

Note that all Tame IR nodes are effectively AST subtrees, because they are subclasses of AST nodes. Users of the Tame IR should not modify IR Nodes, except the TIRStatementList. They should also only use the accessor methods provided by the Tame IR interfaces. The constructors of the Tame IR nodes enforce the constraints of the Tame IR.

All Tame IR nodes implement the interface `TIRNode`. Additionally, all the statement nodes of the Tame IR implement the interface `TIRStmt`.

C.1 Compound Structures

node	extends	contains
TIRFunction	Function	List\<Name> outputParams, String name, List\<Name> inputParams, List\<HelpComment> helpComments, TIRStmtList stmts, List\<TIRFunction> nestedFunctions
TIRStmtList	List\<Stmt>	List\<TIRStmt> statements
TIRIfStmt	IfStmt	Name ConditionVar, TIRStmtList IfStmts, TIRStmtList ElseStmts
TIRWhileStmt	WhileStmt	Name condition, TIRStmtList body
TIRForStmt	ForStmt	Name var, Name lower, (Name inc), Name upper, TIRStmtList stmts

C.2 Non-Assignment Statements

node	extends	contains
TIRReturnStmt	ReturnStmt	–
TIRBreakStmt	BreakStmt	–
TIRContinueStmt	ContinueStmt	–
TIRGlobalStmt	GlobalStmt	List\<Name> names
TIRPersistentStmt	PersistentStmt	List\<Name> names
TIRCommentStmt	EmptyStmt	–

C.3 Assignment Statements

node	extends	contains
TIRAbstractAssignStmt	AssignStmt	–
TIRAbstractAssignFromVarStmt	TIRAbstractAssignStmt	Name rhs
TIRArraySetStmt	TIRAbstractAssignFromVarStmt	Name arrayVar, TIRCommaSeparatedList indices, Name rhs
TIRCellArraySetStmt	TIRAbstractAssignFromVarStmt	Name arrayVar, TIRCommaSeparatedList indices, Name rhs
TIRDotSetStmt	TIRAbstractAssignFromVarStmt	Name dotVar, Name field, Name rhs
TIRAbstractAssignToListStmt	TIRAbstractAssignStmt	IRCommaSeparatedList targets
TIRArrayGetStmt	TIRAbstractAssignToListStmt	Name lhs, Name rhs, TIRCommaSeparatedList indices
TIRCellArrayGetStmt	TIRAbstractAssignToListStmt	Name cellVar, TIRCommaSeparatedList targets, TIRCommaSeparatedList indices
TIRDotGetStmt	TIRAbstractAssignToListStmt	TIRCommaSeparatedList lhs, Name dotVar, Name field
TIRCallStmt	TIRAbstractAssignToListStmt	Name function, TIRCommaSeparatedList targets, TIRCommaSeparatedList args
TIRAbstractAssignToVarStmt	TIRAbstractAssignStmt	Name lhs
TIRAssignLiteralStmt	TIRAbstractAssignToVarStmt	Name lhs, LiteralExpr rhs
TIRCopyStmt	TIRAbstractAssignToVarStmt	Name lhs, Name rhs
TIRAbstract-CreateFunctionHandleStmt	TIRAbstractAssignToVarStmt	Name lhs, Name function
TIRCreateFunctionReferenceStmt	TIRAbstract-CreateFunctionHandleStmt	Name lhs, Name function
TIRCreateLambdaStmt	TIRAbstract-CreateFunctionHandleStmt	Name lhs, Name function List<Name> lambdaParameters, List<Name> enclosedVariables

C.4   Other Tame IR Nodes

node	extends	contains
TIRCommaSeparatedList	List<Expr>	List<Expr> elements

Bibliography

[AACM07] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO languages. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, Montreal, Quebec, Canada, 2007, pages 53–64. ACM, New York, NY, USA.

[AP02] George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, Berlin, Germany, 2002, pages 294–303. ACM, New York, NY, USA.

[DHR11] Jesse Doherty, Laurie Hendren, and Soroush Radpour. Kind analysis for MATLAB. In *In Proceedings of OOPSLA 2011*, 2011, pages 99–118.

[Doh11] Jesse Doherty. McSAF: An Extensible Static Analysis Framework for the MATLAB Language. Master's thesis, McGill University, December 2011.

[FAFH09] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, Honolulu, Hawaii, 2009, SAC '09, pages 1859–1866. ACM, New York, NY, USA.

[INR09] INRIA. Scilab, 2009. http://www.scilab.org/platform/.

[JB01]   Pramod G. Joisha and Prithviraj Banerjee. Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In *APL '01: Proceedings of the 2001 conference on APL*, New Haven, Connecticut, 2001, pages 7–21. ACM, New York, NY, USA.

[JB03]   Pramod G. Joisha and Prithviraj Banerjee. Static array storage optimization in MATLAB. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, San Diego, California, USA, 2003, pages 258–268. ACM, New York, NY, USA.

[Joi03]   Pramod G. Joisha. a MATLAB-to-C translator, 2003. <http://www.ece.northwestern.edu/cpdc/pjoisha/mat2c/> .

[LH11]   Nurudeen Lameed and Laurie J. Hendren. Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In *Proceedings of the International Compiler Conference (CC11)*, 2011, pages 22–41.

[Li09]   Jun Li. McFor: A MATLAB to FORTRAN 95 Compiler. Master's thesis, McGill University, August 2009.

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

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

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

[Rad12]   Soroush Radpour. Understanding and Refactoring MATLAB. Master's thesis, McGill University, April 2012.

Bibliography

[RP99]   Luiz De Rose and David Padua. Techniques for the translation of MATLAB
              programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323,
              1999.