McSAF: A Static Analysis Framework for MATLAB*

Jesse Doherty and Laurie Hendren

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

Abstract. Matlab is an extremely popular programming language used by scientists, engineers, researchers and students world-wide. Despite its popularity, it has received very little attention from compiler researchers. This paper introduces McSaf, an open-source static analysis framework which is intended to enable more compiler research for Matlab and extensions of Matlab. The framework is based on an intermediate representation (IR) called McLast, which has been designed to capture all the key features of Matlab, while at the same time being simple for program analysis. The paper describes both the IR and the procedure for creating the IR from the higher-level AST. The analysis framework itself provides visitor-based traversals including fixed-point-based traversals to support both forwards and backwards analyses. McSaf has been implemented as part of the McLab project, and the framework has already been used for a variety of analyses, both for Matlab and the AspectMatlab extension.

1 Introduction

Matlab is a popular dynamic ("scripting") programming language that has been in use since the late 1970s, and a commercial product of MathWorks since 1984, with millions of users in the scientific, engineering and research communities.[1] There are currently over 1200 books based on Matlab and its companion software, Simulink (`http://www.mathworks.com/support/books`).

Despite the popularity of the language, there exists relatively little compiler research for Matlab, and without an existing framework it is difficult for researchers to tackle such research. McSaf, the topic of this paper, is a compiler analysis framework that is intended to enable compiler research by providing both a convenient intermediate representation and an intraprocedural analysis framework which can be used both for Matlab and language extensions of Matlab. It has been developed as a key component of the McLab project [3].

* This work was supported, in part, by NSERC. A special thanks to Soroush Radpour for his help with the experiments.

[1] The most recent data from MathWorks shows that the number of users of Matlab was 1 million in 2004, with the number of users doubling every 1.5 to 2 years.(From `www.mathworks.com/company/newsletters/news_notes/-clevescorner/jan06.pdf`.)

This paper does not just report on a standard compiler engineering effort. Designing an intermediate representation (IR) for MATLAB that is suitable for program analysis was quite challenging for three reasons. First, the MATLAB language has grown somewhat organically and does not have a precise documented semantics. Thus, the IR needs to expose those semantics both correctly and in a manner that simplifies subsequent analyses. One example is the correct handling of the "`&`" and "`|`" operators, which are short-circuiting in some situations and not in others. Secondly, MATLAB supports several high-level features that are convenient for programmers, but difficult for compilers. For example, an assignment statement may assign to numerous variables at the same time, and the number of such variables may not be known statically. The IR must expose and simplify such features. Thirdly, the IR must correctly encode which identifiers refer to variables, and which refer to named functions. This is not trivial since the syntax of MATLAB is ambiguous, and there are no explicit declarations of variables. For example, the expression "`a(i)`" could mean four different things, depending on whether "`a`" is a variable or function, and whether "`i`" is a variable or a function. Consider two of the cases. If both "`a`" and "`i`" are variables, the expression "`a(i)`" means the i^{th} element of array "`a`". If both "`a`" and "`i`" are named functions, then "`a(i)`" is a call to function "`a`", where the argument is a call to the built-in function "`i`" (which returns the complex value i). Clearly a program analysis will apply very different analysis rules for an array access or a function call. Thus, these syntactic ambiguities should be resolved in the IR.

Given a suitable IR, another important challenge is to design an analysis framework that supports a wide variety of intraprocedural analyses. Our goal was to design a framework that is easy to use, thus enabling others to design new analyses. We also felt that it was important that our approach should support language extensions so that an analysis originally designed for MATLAB could be easily adapted to an extension of MATLAB.

In this paper we provide our solution to these challenges. Our contributions are as follows:

Design of MCLAST **IR:** The design and implementation of MCLAST, a lower-level AST-based intermediate representation that both exposes important MATLAB semantics and simplifies program analysis.

Generating MCLAST**:** The design and implementation of a collection of simplifying transformations that together provide a mechanism for generating the MCLAST IR from the higher-level AST. This simplification framework encodes the dependency structure between simplifications to enable the application of a subset of simplifications and also to allow for a structured way of introducing future simplifications. The simplifying transformations themselves support both standard and MATLAB-specific transformations.

Analysis Framework: The design and implementation of an extensible program analysis framework for MATLAB. The framework supports a variety of visitor-based traversal mechanisms including a depth-first style traversal suitable for flow-insensitive analyses and a structured-program-analysis traversal which supports both forward and backwards flow-sensitive analyses. The flow-sensitive traversal mechanism automatically supports conditional control flow, and iterative constructs, including support for "`break`" and "`continue`" statements.

Extensibility: Both the simplification and analysis frameworks have been designed to support extensions of MATLAB.

The remainder of this paper is structured as follows. In Section 2 we provide an overview of the whole McLAB project and show how the McSAF framework fits into the big picture. Section 3 introduces the McLAST IR and the simplifying transformations, while Section 4 describes the analysis framework. Related work is discussed in Section 5 and Section 6 concludes.

2 Overview

The McSAF system forms a key component of the McLAB project as illustrated in Figure 1. As a whole, McLAB is intended as a complete toolkit for MATLAB systems including an extensible front-end built using MetaLexer [5] and JastAdd [2,12], and several back-ends including source-to-source tools like the refactoring toolkit [18], static compilers such as the MATLAB-to-FORTRAN translator (McFOR) [16] and the MATLAB Tamer [10], and dynamic virtual machine/JIT systems (McVM) [6]. All parts of McLAB, with the exception of McVM, are implemented using Java and Java-based tools.

McSAF is the heart of the system, providing both the lower-level McLAST IR and the analysis toolkit/engine (as indicated by the grey boxes). The dotted boxes in Figure 1 denote existing implementations of flow analyses that use the McSAF toolkit. Note that McSAF can be used to build analyses both on the higher-level AST (for example, kind analysis), as well as on the lower-level McLAST. McSAF includes three different implementations of kind analysis [9], and a variety of standard dataflow analyses. Other components of the larger McLAB system also build analyses using McSAF. The refactoring toolkit implements a variety of specialized analyses to enable refactoring transformations [18], and the MATLAB Taming toolkit implements specialized simplifications and analyses to generate a call graph and type/class information [10]. McVM uses both the IR and some standard flow analysis information generated by McSAF.[2]

It is our intention that future users of McSAF would add to both the standard flow analyses and develop new specialized analyses for both our existing back-ends and for their own projects/tools.

3 Intermediate Representations and Simplifications

One of the key steps in designing McSAF was to design and implement an appropriate intermediate representation which exposed key semantics of MATLAB

[2] Specialized analyses in McVM are implemented in C++ and LLVM [15], and thus do not use McSAF.

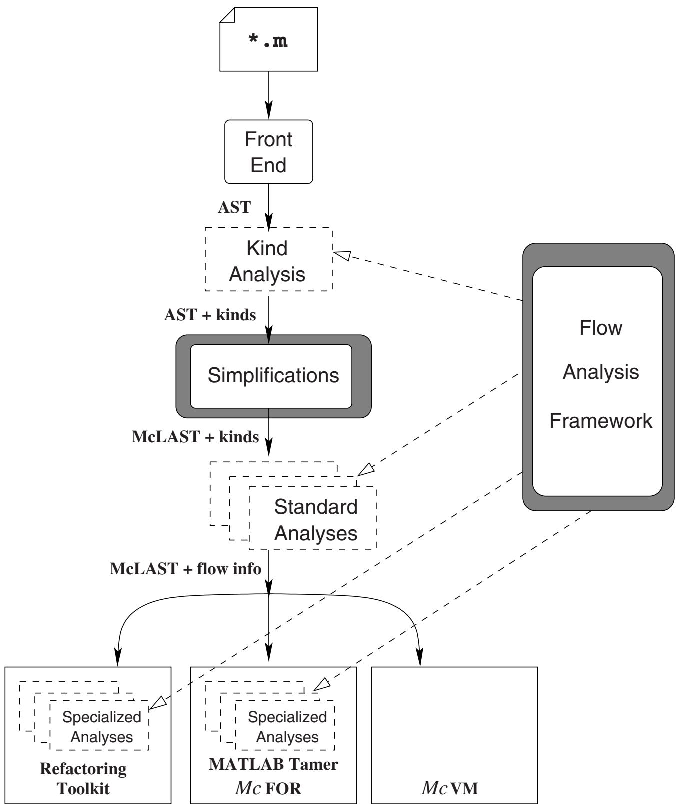


Fig. 1. Overview of McLab and McSaf. The grey boxes indicate the components presented in this paper: *Simplifications* are discussed in Section 3 and the *Flow Analysis Framework* is presented in Section 4. The dotted boxes correspond to existing analyses implemented using McSaf.

and provided a good basis for analyses. Although we hope the end result is quite clean and accomplishes these goal, the process of determining the correct IR was not at all straightforward. In this section we introduce the existing high-level AST and then discuss the simplifications we developed that result in the lower-level IR, McLast.

3.1 High-Level AST

The McLab front-end already has a well-defined AST specification and a Java implementation of the AST that is generated based on a JastAdd specification. Figure 2 gives an extract of the AST specification.

For those not familiar with JastAdd, we give a quick introduction. Each declaration defines a node type, which may be *abstract* or *concrete*. An abstract

```
// Top−level structure
abstract Program;
CompilationUnits ::= Program*;
Script : Program ::= HelpComment* Stmt*;
FunctionList : Program ::= Function*;
Function ::= OutputParam:Name* <Name:String> InputParam:Name*
   HelpComment* Stmt* NestedFunction:Function*;

// Statements
abstract Stmt;
WhileStmt : Stmt ::= Expr Stmt*;
ForStmt : Stmt ::= AssignStmt Stmt*;
BreakStmt : Stmt;
ContinueStmt : Stmt;
ReturnStmt : Stmt;
IfStmt : Stmt ::= IfBlock* [ElseBlock];
IfBlock ::= Condition:Expr Stmt*;
ElseBlock ::= Stmt*;
SwitchStmt : Stmt ::= Expr SwitchCaseBlock* [DefaultCaseBlock];
SwitchCaseBlock ::= Expr Stmt*;
DefaultCaseBlock ::= Stmt*;
AssignStmt : Stmt ::= LHS:Expr RHS:Expr;
ExprStmt : Stmt ::= Expr;

// Expressions
abstract Expr;
abstract LiteralExpr : Expr;
abstract LValueExpr : Expr;
abstract UnaryExpr : Expr ::= Operand:Expr;
abstract BinaryExpr : Expr ::= LHS:Expr RHS:Expr;
NameExpr : LValueExpr ::= Name;
ParameterizedExpr : LValueExpr ::= Target:Expr Arg:Expr*;
CellIndexExpr : LValueExpr ::= Target:Expr Arg:Expr*;
DotExpr : LValueExpr ::= Target:Expr Field:Name;
MatrixExpr : LValueExpr ::= Row*;
Name ::= <ID : String>;
Row ::= Element:Expr*;
 ...
RangeExpr : Expr ::= Lower:Expr [Incr:Expr] Upper:Expr;
ColonExpr : Expr;
EndExpr : Expr;
CellArrayExpr : Expr ::= Row*;
FunctionHandleExpr : Expr ::= Name;
LambdaExpr : Expr ::= InputParam:Name* Body:Expr;
```

Fig. 2. AST Definition

declaration, such as in line 2, will result in an abstract Java class being generated. Declarations can declare subtypes, for example on lines 4 and 5, Script and FunctionList are declared to be subtypes of Program. Each declaration may list children, for example, lines 6-7 declare that a Function has six children. Children may be explicitly named or not. For example, the first child of Function has the name OutputParam, whereas the fourth and fifth children are not named. A child may be a list of a specific type (indicated by $*$), a singleton, or optional (indicated by [...]).

The AST definition follows the natural abstract syntax of MATLAB. A MATLAB program consists of a collection of compilation units. Each compilation unit can contain either a script or a list of functions. Scripts contain only comments and statements. Functions have output parameters, input parameters, and possibly nested functions. The function body consists of comments and statements. Statements can be simple expression or assignment statements, or control-flow statements. Since MATLAB supports many high-level array operations, there are quite a few types of expressions.

3.2 Simplification Process

Although it is possible to define program analyses over the high-level AST, such an analysis must be able to handle arbitrarily complicated expressions and must correctly handle the semantics of MATLAB constructs. Thus, we have defined a lower-level, simpler AST called McLAST. Figure 3 shows the overall simplification process. The front-end delivers the high-level AST (also called McAST) and we wish to create a semantically equivalent lower-level representation.

One of the first surprises for us was that we could not create a lower-level AST before we resolved the meaning of all identifiers. For example, it is not possible to correctly simplify an expression of the form `a(f(end))` before one knows whether `f` is a function or a variable. Thus, before simplification, *kind analysis* [9] must be applied to determine the meaning of identifiers (AST nodes of type Name). The *kind analysis* has itself been implemented using the McSAF framework presented in this paper, and it computes, for each Name node, one of the following kinds: VAR, the Name refers to a *variable*; FN, the Name refers to a *named function*; or ID, the Name could be either a VAR or FN at run-time.

Another challenge is that the simplifying transformations depend on each other, and we wanted to be able to: (1) support applying only some transformations; and (2) allow for new transformations to be added in a consistent and simple fashion. Thus, each simplification forms part of a dependency structure which is enforced by the framework.

In the following sections, Sec. 3.3 to Sec. 3.6 outline the highlights of the individual simplfications, and then in Sec. 3.7 we describe our approach to handling dependencies between simplifications. We also provide some empirical measurements on the frequency of simplifications in Sec. 3.8. More detailed descriptions can be found in the first author's thesis [8] and in the implementation [3].

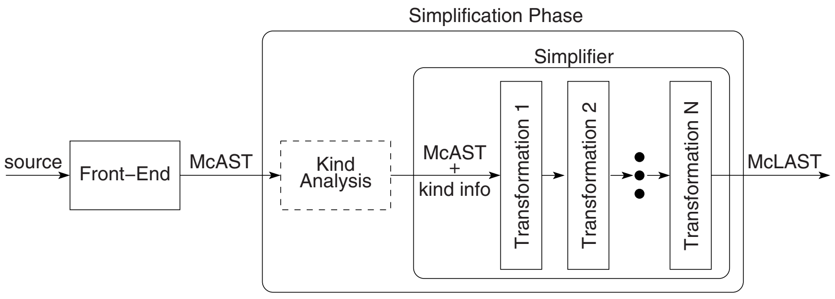


Fig. 3. Simplifications

3.3 Exposing Implicit Control Flow

One important group of simplifications is to expose implicit control flow. Explicit and simple control flow makes the subsequent implementation of flow-sensitive analyses much more straightforward.

The case of MATLAB short-circuit operators illustrates the difficulty in capturing the semantics of MATLAB, which often includes special and somewhat irregular rules that seem to have emerged over time. MATLAB supports implicit control flow via the scalar logical operators `&&` and `||`, which are always short-circuit operators in the usual sense. The logical operator simplification transforms all occurrences of these operators to equivalent explicit control flow with conditional statements, ensuring that code copying is minimized. For example, the original expression in Figure 4(a) would be converted to the conditional statement in Figure 4(b).

```
t = E1 && E2;
```

(a) original

```
t = E1;
if (E1)
  t = E2;
else
  t = false;
end
```

(b) first try

```
t = E1;
CheckScalarStmt(t);
if (E1)
  t = E2;
  CheckScalarStmt(t);
else
  t = false;
end
```

(c) correct

Fig. 4. Example of simplifying short-circuit `&&`

In implementing the short-circuit transformations we were careful not to duplicate code and we were also quite careful to capture the correct MATLAB semantics of short-circuits. In particular, the MATLAB semantics for the scalar logical operators dictate that the result of any operand that is evaluated must be a scalar logical, and if it is not a scalar, a run-time error is raised. However, the same run-time check is not made for conditional expressions for `if` and `while` statements. These conditional statements are considered true when the result is non-empty and contains all nonzero elements (logical or real numeric), and false otherwise. Thus, to maintain the correct semantics the simplification introduces new `CheckScalarStmt` nodes, as shown in Figure 4(c).

Another non-obvious twist is that MATLAB also has implicit short-circuits for the element-wise operators `&` and `|`, but **only** when they appear as the top-level operators in the condition of an `if` or `while` construct. Thus, the simplification for element-wise logical operators must ensure that the short-circuit simplification is applied in the correct contexts. In particular, such simplifications must be done before another transformation moves the expression out of the condition.

3.4 Simplifying Control Constructs

In addition to exposing implicit control flow, simplifications are also applied to `if` and `for` statements. The `if` simplification simply restructures `elseif` constructs to equivalent nested `if-else` constructs. This ensures that subsequent flow analyses only have to deal with two control-flow branches.

The `for` simplification is somewhat more MATLAB-specific. Many MATLAB `for` loops are written in the style of Figure 5(a), where `i` takes on values from 1 to `n` in steps of `k`. This style of loop, which we call a *range for loop*, is ideal for subsequent analysis and loop transformations. However, the general form of a MATLAB `for` loop is shown in Figure 5(b). The general semantics is that the expression `E` is evaluated to an array `a`, and `a` is treated as a two-dimensional array. The loop iterates over `a`, assigning to `i` the i'th column of `a`. If `a` is empty, then the loop body does not execute, and the final value of `i` is the empty array. Figure 5(c) shows the simplification that converts a general array to an equivalent range `for` loop.

```
for i = 1:k:n
    <BODY>
  end
```

(a) range for loop

```
for i = E
    <BODY>
  end
```

(b) original general loop

```
t1=E;
[t2,t3] = size(t1);
i = [];
for t4 = 1:t3
  i = t1(:,t4);
  <BODY>
end
```

(c) simplified loop

Fig. 5. Example of simplifying `for` loops

3.5 Simplifying Single Statements and Expressions

A key part of the simplification process is simplifying single assignment statements and expressions. The key idea is that each statement and expression is simplified as much as possible, thus reducing the complexity for subsequent analyses.

The first simplification merely divides an assignment so that it has a complex expression only on the right-hand-side (rhs) or only on the left-hand-side (lhs). Thus, statements of the form `E1=E2;` are transformed to `t1=E2; E1=t1;`. Subsequent simplifications then simplify either the rhs (*RValue*) or the lhs (*LValue*).

Each *RValue* is simplified so that it contains at most one complex operation (function call, operand, index, field access or range expression). For example, assuming that `x`, `a` and `i` have kind VAR, the *RValue* expression in `lhs=a(f(g(i),sin(x))).b` would be simplified to `t1=g(i); t2=sin(x); t3=f(t1,t2); t4=a(t3); lhs=t4.b`.

Simplifying complex expressions that are *LValue*'s (i.e. expressions on the lhs of assignment statements) is more difficult because the expression now denotes an address and not a value and MATLAB has no natural way of expressing addresses. Thus, the simplification of *LValue*s simplifies internal expressions as much as possible, but does allow for a chain of indexing and field expressions. The grammar in Figure 6 gives the rules for *LValue*, with a further restriction that NameExpr must refer to names with kind VAR. If we take the same expression as before but use it as an *LValue*, as in the assignment statement `a(f(g(i),sin(x))).b = rhs`, the simplification would be: `t1=g(i); t2=sin(x); t3=(t1,t2); a(t3).b = rhs`. Note that the final statement in the simplification uses the chain of references `a(t3).b`, which cannot be further simplified.

$$\begin{aligned} LValue &:= \texttt{NameExpr} \\ &\mid Indexing \\ &\mid Access \end{aligned}$$

$$\begin{aligned} Indexing &:= \texttt{NameExpr}(NameOrVal^*) \\ &\mid Access(NameOrVal^*) \end{aligned}$$

$$Access := LValue.\texttt{Name}$$

$$\begin{aligned} NameOrVal &:= \texttt{NameExpr} \\ &\mid \texttt{LiteralExpr} \end{aligned}$$

Fig. 6. LValue grammar

There is one additional MATLAB-specific expression that must be simplified correctly, which is the `end` expression. The `end` expression binds to the tightest enclosing array or cell array, and it returns the last index of the dimension in which it appeared. For example, the expression `a(foo()).b(i,end,k)` where `a` has kind VAR and `foo` has kind FN, `end` refers to the last index of the second dimension of the 3-dimensional view of the array resulting from the evaluation of `a(foo()).b`. The `end` simplification replaces the `end` expression with an explicit call to the `endfn` function which has three arguments: the array, the dimension in which the `end` expression was found, and the total number of dimensions. For the example above, the simplification is quite straightforward and would be `t1=foo(); t2=a(t1); t3=t2.b; t4=endfn(t3,2,3); t3(i,t4,k)`. However, more complex situations can arise, especially when `end` is used inside an *LValue*. Consider the example, `a(foo()).b(i,bar(end),k)=rhs`, assuming `bar` has kind Fn. This will be simplified to `t1=foo(); t2=endfn(?,2,3); t3=bar(t2); a(t1).b(i,t3,k)=rhs` where the `?` must be filled in with correct simplified *LValue*, which in this case is `a(t1).b`.

3.6 Dealing with Multiple Assignments

In the previous simplifications we have assumed that assignments have a simple lhs. However, Matlab supports assignments to multiple variables on the lhs. Dealing with multiple values on the lhs is not usually difficult, but with Matlab there is an important complication. That is, that Matlab allows an unknown number of lhs variables. The simplifications handle the simple and more complicated cases as follows.

In the case where the number of lhs variables is known, the simplification ensures that only simple variables, without repetition occur on the lhs. For example, `[a,b.c,a]=lhs` would be simplified to `[a,t1,t2]=lhs; b.c=t1; a=t2`. This simplification simplifies subsequent interprocedural analyses since the number and names of the return parameters are explicit.

The case where the number of variables on the lhs is not known is harder, and it was quite difficult to decide how to handle this case. In the end we decided to introduce a new `CSL` IR node. A typical example is the statement `[a,b{:},c]=rhs`. In Matlab this specifies that the first return value is bound to `a`, the last return value is bound to `c`, and the middle values are bound to the cell array `b`. The simplification for these cases introduces an explicit `CSL` node for each item in the return list than can possibly correspond to a list of values (known in Matlab as a *Comma Separated List* (CSL)). For each such CSL, the simplification introduces an explicit `CSL` node associated with a new temporary name. For example, `[a,b{:},c]=rhs` would be simplified to `[a,CSL[t1],c] = rhs; [b{:}] = CSL[t1]`.

3.7 Simplification Dependencies

The complete simplification procedure is implemented as a collection of simplifying transformations. Some simplifications depend upon others having already been applied. The dependencies between the existing simplifications is given in Figure 7. The simplification called `FULL` represents the case where all simplifications should be applied. However, framework users might want to only apply some simplifications, for example loop transformations may just want to apply the `FOR` simplifications.

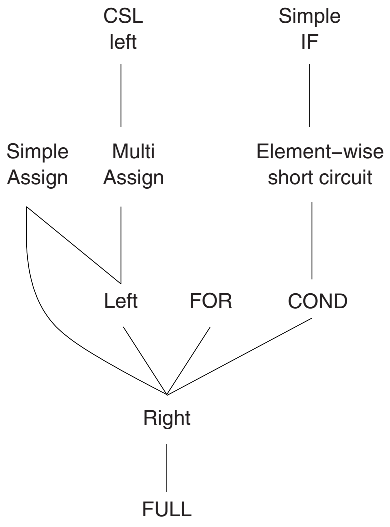


Fig. 7. Dependencies for Simplifications

To enforce the dependencies, the framework provides the Simplifier class. In addition, each simplification is implemented as a class extending Abstract-Simplification. The AbstractSimplification class requires that each simplification have a method called getDependencies that returns a set of dependencies. In order to use the simplifier, an instance must be constructed with a given set of simplifications to perform. The simplifier then performs a depth first traversal of the dependency DAG producing a list of simplifications, avoiding duplication. Executing the simplifications in the order of the list will ensure that all dependencies will be met. To make it simpler to perform any given simplification and its dependencies, each simplification has a getStartSet static method. This method returns a singleton set containing the simplification itself. Clearly framework users can add new simplifications and state their dependencies quite easily. In fact, the Matlab Taming project has recently built upon the McSaf framework by introducing new simplifications which were inserted quite easily using this approach.

3.8 Simplification Frequencies

In order to examine the relative frequency applications for each simplification type, we instrumented our simplification framework to count the number of times each simplification caused a statement to be rewritten and the number of times the simplification extracted an expression to a temporary variable. We applied the instrumented simplifier to a large collection of Matlab functions and scripts.[3] The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing. We analyzed 3057 projects composed of 11698 functions and 2349 scripts. The projects vary in size between 283 files in one project to a single file. A summary of the data collected is given in Figure 8, ordered by increasing frequency.

Simplification	# rewrites	(%total)	# temps extracted
FOR	329	(0.1%)	329
Multi-Assign	354	(0.1%)	651
Element-wise short-circuit	1791	(0.5%)	4068
Simple IF	4518	(1.3%)	0
Left	6649	(1.9%)	7969
Simple-Assign	24478	(7.1%)	24478
Right	306397	(88.9%)	325780

Fig. 8. Frequencies for Simplifications

These results show that the FOR and Simple IF transformations are applied relatively infrequently. The benchmarks contained 12189 `for` statements and only 329 required the FOR simplifications. Similarly only 4518 `elseif` statements needed to be simplified, out of a total of 31758 `if` statements in the benchmarks. We were somewhat surprised that there were almost 1800 occurrences of the element-wise short-circuit simplification. The use of element-wise short-circuiting is discouraged by MathWorks, but it appears that existing code does use this feature. This may be a potential refactoring opportunity. It was also interesting to note that there were relatively few applications of the Multi-Assign simplification, especially as compared to the Simple-Assign. As expected, by far the most frequent simplification was the Right simplification which simplifies expressions by extracting sub-expressions to a temporary.

4 Analysis Framework

A key part of the McSaf toolkit is an analysis framework that supports a variety of pre-defined and extensible traversal mechanisms; built-in and extensible support for representing a variety of flow data types; and support for depth-first and structure-based analyses. The toolkit has been designed to work both for the higher-level AST, as well as for the lower-level McLast IR.

[3] Benchmarks were obtained from individual contributors plus projects from `http://www.mathworks.com/matlabcentral/fileexchange`, `http://people.sc.fsu.edu/~jburkardt/m_src/m_src.html`, `http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/` and `http://www.mathtools.net/MATLAB/`. This is the same set of benchmarks that are used in [9].

4.1 Traversal Mechanism

The traversal mechanism is central to the framework - in fact it is used both to drive the simplifications presented in the previous section and the analyses presented later in this section. The framework accommodates different traversals by implementing a variant of the visitor pattern. The IR consists of instances of different types of AST nodes. The types form a class hierarchy, which is induced by the JastAdd specification, an excerpt of the hierarchy is depicted in Figure 9.

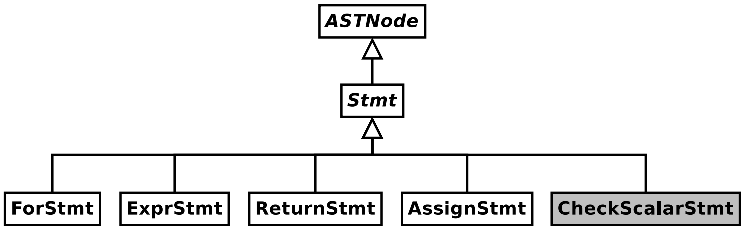


Fig. 9. Excerpt of AST class hierarchy. The grey class, CheckScalarStmt is an AST node that is part of McLast and not McAst. All white classes in this diagram are part of both McLast and McAst.

To facilitate traversal, there is a Java interface, NodeCaseHandler, that consists of methods of the form void caseStmt(Stmt node). There is one such method for every AST class. The framework also provides a simple abstract implementation called AbstractNodeCaseHandler. This implementation provides default behaviour for each node case. This default behaviour is that for each AST class, the node case for that class simply forwards to the node case of its parent class. The forwarding is done by calling the case for the parent class with the input from the case for the given class. We demonstrate this with the following excerpt from AbstractNodeCaseHandler for the AssignStmt and Stmt classes. Notice that case AssignStmt(...) is forwarding up to the case belonging to its parent class, Stmt.

```
public void caseAssignStmt( AssignStmt node ) { caseStmt( node ); }
public void caseStmt( Stmt node ) { caseASTNode( node ); }
```

Figure 9 shows that the AssignStmt node type extends the Stmt node type. This means that the default behaviour for case AssignStmt(...) is to forward to case - Stmt(...), which is done by passing the argument from case AssignStmt(...) to case Stmt(...). The definition for the case AssignStmt(...) method demonstrates the forwarding behaviour. This method takes in an instance of AssignStmt and calls case Stmt(...) with that instance. Note that ASTNode is the root type of the AST class hierarchy. The Stmt class is a top level node type, which directly extends ASTNode, so the case Stmt(...) will forward to case ASTNode(...). The AbstractNodeCaseHandler leaves the case ASTNode(...) method unimplemented.

Each AST class implements a method called analyze that takes a NodeCaseHandler as an argument. These methods will call the appropriate node case of the given handler, passing itself to the handler. For example, here is the code implementing the analyze method in the AssignStmt class.

```
public void analyze( NodeCaseHandler handler ) { handler.caseAssignStmt( this ); }
```

In order to create a particular traversal, a programmer needs to create a specialized NodeCaseHandler. The different types of analyses are implemented in this manner, but a programmer can also directly create a specialized traversal. To demonstrate this process we present a simple traversal, called StmtCounter, that counts the number of statements in a given AST. Code for this traversal is given is Figure 10.

```
public class StmtCounter extends AbstractNodeCaseHandler {
  private int count = 0;
  private StmtCounter() { super(); }

  public static int countStmts( ASTNode tree ) { tree.analyze( new StmtCounter() ); }

  public void caseASTNode( ASTNode node )
    { for( int i = 0; i<node.getNumChild(); i++ )
          node.getChild(i).analyze( this );
    }

  public void caseStmt( Stmt node ) { count++; caseASTNode( node ); }
}
```

Fig. 10. Example traversal counting statements

To use this class, a programmer simply needs to call the static method countStmts. This method creates a new instance of the traversal and starts the analysis off.

This traversal will visit all nodes in the tree in depth-first order, and count each statement node. There are two important details to note from this example. The first thing is the case ASTNode(...) implementation. In this example, this method does the actual traversal over the tree, looping through and visiting each of a node's children. Since StmtCounter extends AbstractNodeCaseHandler all cases that are not overridden will forward up until they reach this case. This means that the default behaviour for AST nodes will be to simply traverse through their children. This is a common pattern when implementing traversals. The flow-insensitive traversal is implemented similar to this, and the flow-sensitive traversals have a similar case ASTNode(...) with other behaviour implemented for control structures like loops and conditionals.

The second thing to notice is the case Stmt(...) method. Besides case ASTNode (...), this is the only case implemented by StmtCounter. Again, since StmtCounter extends AbstractNodeCaseHandler, all node types that are descendants of Stmt will forward up to this case. So this case will capture all statements, which gives a good place to perform the count. Note that this implementation of case Stmt-(...) forwards to case ASTNode(...). This is because there are some statements, such as if statements, that can contain other statements. We wish to visit all of the statements contained in other statements, so we need to visit the children of a given statement. To do this, we simply forward to case ASTNode(...).

The StmtCounter example does have some inefficiencies. It will visit all children of a given node, even children that cannot be, or contain, statements. For example, the children of an expression cannot be a statement, nor can it contain statements. This shortcoming can be overcome by providing specialized implementations of appropriate cases. To avoid visiting unnecessary expression children one could add the following method to the class. This method will prevent all children of any expression from being visited.

```
...
public void caseExpr( Expr node ) { return; }
...
```

The example can be refined further, but the original version is concise and correct, and demonstrates how simple it can be to create new traversals. The traversal mechanism is also used by the simplifications presented in Sec. 3. There is a specialized traversal created for all simplifications. This traversal implements the rewrite mechanisms as well as the AST traversal. Each simplification extends this simplification traversal, implementing it's own behaviour for the appropriate node cases.

4.2 Representations for Flow-Data

An analysis is written to produce information about the program being analyzed. McSaf's analysis classes are generic in the type of information produced. An analysis of type StructuralAnalysis <D> is an analysis that produces information of type D. To make the framework as general as possible, the information can be of any type. However, the type of information often falls into certain categories. One example is an analysis that produces, for every program point, a set of variables that must be defined at that program point. Alternatively, for every program point, the analysis could have produced a map from variable names to their types. To make implementing analyses that produce such information easier, the McSaf framework defines interfaces and implementations for basic flow-data. A class hierarchy for flow-data structures provided by the framework is given in Figure 11.

The FlowData<D> interface is the base type for all predefined flow-data representations. This type represents a collection of data of type D. The interface is primarily intended to tag a class as representing flow-data. As such, it defines no methods. In addition to this basic interface, the framework also provides two more specific interfaces, one for sets and the other for maps. For each of these,

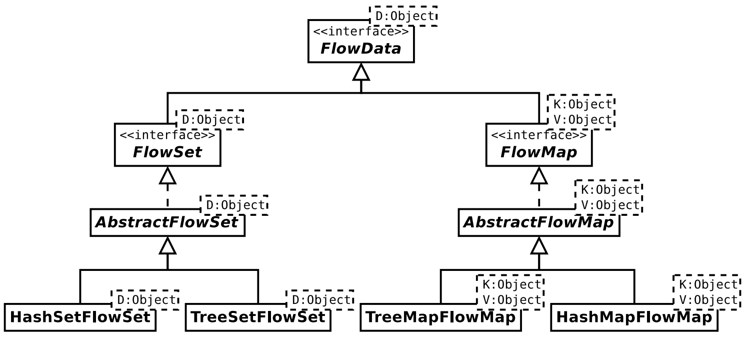


Fig. 11. flow-data class hierarchy

an abstract implementation is provided to make creating new implementations simpler. In addition, each of these interfaces also have concrete implementations.

4.3 Depth-First Analysis

The simplest form of analysis supported by the framework is the depth-first analysis. This type of analysis is intended to traverse the tree structure of the AST, visiting each node in a depth-first order. The depth-first analysis type can be used to implement flow-insensitive analyses.

This type of analysis is implemented by extending the AbstractDepthFirstAnalysis<A> class. The AbstractDepthFirstAnalysis implements the Analysis interface and extends AbstractNodeCaseHandler. This relationship is depicted in Figure 12.

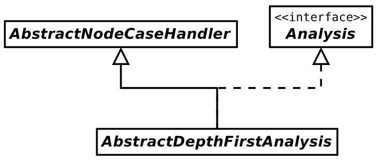


Fig. 12. Class hierarchy snipet for depth-first analysis

Since AbstractDepthFirstAnalysis extends AbstractNodeCaseHandler it inherits all the default traversal behaviour. It extends this behaviour with default implementations of the new case methods defined by the Analysis interface. The behaviour for these new cases is to forward to the case associated with the type of the argument that the case accepts. For instance caseLoopVar(AssignStmt loopVar) accepts an AssignStmt. So the default behaviour will be to forward to caseAssignStmt(...). The caseWhileCondition(...) and caseIfCondition(...) are exceptions to this. These cases are specialized versions of caseCondition(...) so they will both forward to caseCondition(...) by default.

The most important feature of AbstractDepthFirstAnalysis is that it implements a caseASTNode(...) method. The implementation of this method provides the basic traversal for this type of analysis. Figure 13 presents the source code for this method. The caseASTNode(...) method takes in the ASTNode being visited. For each child of that node, that child is analyzed. So to reiterate, since AbstractDepthFirstAnalysis extends AbstractNodeCaseHandler, and due to its implementation of caseASTNode(...), the default behaviour for every node is to simply analyze all children of that node.

```
public void caseASTNode(ASTNode node)
{ //visit each child node in forward order
  for( int i = 0; i<node.getNumChild(); i++ ){
     if( node.getChild(i) != null )
       node.getChild(i).analyze( callback );
     }
}
```

Fig. 13. Depth-first caseASTNode(...) source code

AbstractDepthFirstAnalysis also defines some new methods and fields for storing and accessing the data being produced by the analysis. It provides a map from AST nodes to the data being computed. This allows data to be associated with any desired node. In order to implement a new depth-first analysis, a programmer must create a concrete class that extends AbstractDepthFirstAnalysis. To create this class, a programmer must: (1) select an analysis data type; (2) implement an appropriate newInitialFlow method; and (3) implement an appropriate constructor. This will result in an analysis that will traverse the entire tree visiting each node in depth-first order. To get the analysis to perform a useful task, the programmer must override appropriate case methods. The analysis will usually build up flow-data, and can also associate flow-data with particular nodes in the tree.

To demonstrate the process of implementing a depth-first analysis, we present a simple example analysis, given in Figure 14. This analysis collects all names that are assigned to. It performs two tasks. First, for each assignment statement in the tree, it associates all names that are assigned to by that assignment statement to the assignment statement. Second, it collects in one set, all names that are assigned to in the entire AST. Names are stored as strings, so the flow-data has type HashSetFlowSet<String>. The analysis defines a field to store the full set of names, and a flag for indicating when the traversal is in the lhs of a statement. There are two accessor methods, one to get all the names, and the other to get the full set. The core of the analysis is defined by the three "case" methods which guide the traversal and collect the information.

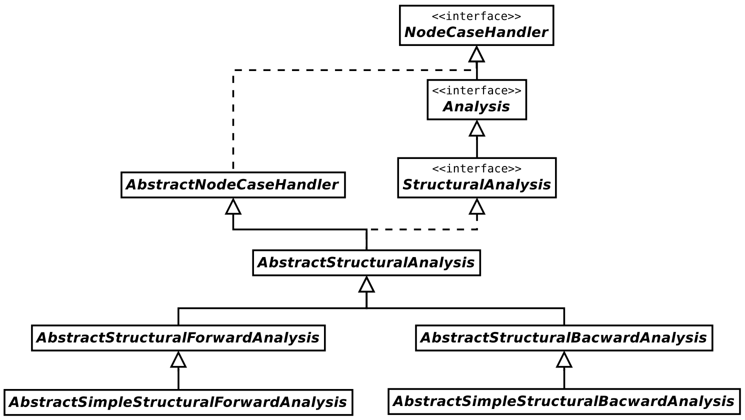
```
public class NameCollector extends
            AbstractDepthFirstAnalysis<HashSetFlowSet<String>>
{ private HashSetFlowSet<String> fullSet;
  private boolean inLHS = false;

  public NameCollector(ASTNode tree)
  { super(tree);
     fullSet = new HashSetFlowSet<String>();
  }

  public HashSetFlowSet<String> newInitialFlow()
  { return new HashSetFlowSet<String>(); }

  public Set<String> getAllNames() { return fullSet.getSet(); }

  public Set<String> getNames( AssignStmt node )
  { HashSetFlowSet<String> set = flowSets.get(node);
     if( set == null ) return null; else return set.getSet();
  }

  public void caseName( Name node )
  { if( inLHS ) currentSet.add( node.getID() ); }

  public void caseAssignStmt( AssignStmt node )
  { inLHS = true;
    currentSet = newInitialFlow(); // init set for this stmt
    analyze( node.getLHS() ); // analyze only the lhs
    flowSets.put(node,currentSet); // store names in node
     fullSet.addAll( currentSet ); // add to full set
    inLHS = false;
  }

  public void caseParameterizedExpr( ParameterizedExpr node )
  { analyze(node.getTarget()); } // only the target can be a defn
}
```

Fig. 14. NameCollector definition

4.4 Structure-Based Analysis

Structure-based flow analysis is the core part of the analysis framework. Structure-based flow analyses perform a complete forwards or backwards analysis over the AST or McLast representation, merging control flow for conditional and switch statements and computing fixed-points for loops, including the proper handling of `break` and `continue` statements. The computational part is driven via specialized traversal mechanisms, either forwards or backwards. The user of the framework only needs to focus on the analysis rules for basic statements and the implementation of the flow-data type (which is either a direct use of a flow-data type provided by the framework or a specialized user-provided type).

The organization of the structure-based analyses are illustrated in Figure 15. The abstract implementation, called AbstractStructuralAnalysis, provides constructors and implementations for most of the API methods and has extensions to support forwards and backwards analyses.



<<interface>>
NodeCaseHandler
<<interface>>
Analysis
<<interface>>
StructuralAnalysis
AbstractNodeCaseHandler
AbstractStructuralAnalysis
AbstractStructuralForwardAnalysis
AbstractStructuralBacwardAnalysis
AbstractSimpleStructuralForwardAnalysis
AbstractSimpleStructuralBacwardAnalysis

Fig. 15. Class hierarchy snipet for structural analysis

The AbstractStructuralAnalysis implementation is similar to the AbstractDepth FirstAnalysis implementation in that it also provides a protected method void analyze(ASTNode node) and is also intended to abstract away from the basic traversal mechanism. Unlike AbstractDepthFirstAnalysis, AbstractStructuralAnal ysis does not provide an implementation for case ASTNode(...). This is because structural analyses are split into two flavours, forwards and backwards, and each of these flavours requires its own implementation of case ASTNode(...). The forwards and backwards analyses are implemented in a similar way. For each, the framework provides a general abstract implementation and a simple abstract implementation.

The general implementation provides an implementation for the basic API methods. It also provides an implementation for some traversal methods, including loops and conditionals. These implementations for the traversal methods are what makes analyses derived from these classes capable of computing flow-sensitive analyses. In the case of the loop cases, case ForStmt(...) and case WhileStmt(...), they provide the fixed-point computation procedure, and the traversal also correctly handles the control-flow due to `break` and `continue` statements.

The simple implementations go beyond this core functionality. They implement certain behaviour that would not be applicable to all analyses. Such behaviour includes how to deal with `continue` and `break` statements. These implementations represent the functionality needed to write analyses that do not need more complex behaviour. They were provided to make analyses simpler to write, requiring less duplication of code.

Handling Control Flow: Unlike analysis frameworks that operate on control flow graphs, our framework computes the flow information based on the structure of the AST. Figure 16 gives a high-level diagrammatic view of how the traverser handles the control flow. Figure 16(a) demonstrates that the condition is evaluated first, and then the flow information is sent to the `if` and `else` branches. The user of the toolkit is given the ability to specialize the flow sets for each branch, so an analysis can use the results of analyzing the condition to specialize the analysis for different branches. The $\bowtie$ operator demonstrates where the data flow merge operation is applied. Figure 16(b) gives the traversal for switches which demonstrates that for MATLAB the conditions can be arbitrary expressions that must be evaluated before the body of the case. Figure 16(c) and (d) illustrate the behaviour for `while` and `for` loops. In this case the framework takes care of merging the flow information from breaks and continues at the appropriate places, as well as computing a fixed point.

These general implementations represent the core functionality that is needed for these types of analyses. This functionality should be applicable to most analyses of this type, and most flow analysis developers should not have to override them. However, should a flow analyses developer have a special kind of fixed point that he/she would like to implement, this can be done by specifying a new specialized traversal.

Creating an Analysis Instance: To create a forwards analysis, a programmer must extend one of the forward classes from Figure 15. The flow-analysis computations are implemented in the case methods for various node types. The case - ASTNode(...) implements the basic traversal. It does this by looping through the children of a given node and using the provided analyze (ASTNode node) method. Recall that this method deals with basic traversal and also guarantees that the current InSet is set to the previous current OutSet. The case IfStmt(...) and case - SwitchStmt(...) implement the behaviour for non-looping branching code. The `if` statement behaviour provides what we call branching analysis. This means that, if the analysis writer wishes, they can provide a different out flow for when the if condition is true or false. This would be done in an implementation of case IfCondition (...). When a programmer provides true and false flow-data, case IfStmt(...) will ensure that each branch of the `if` will have the appropriate in flow-data.

Creating a backwards analysis is very similar, except that the backwards analysis does not support the option of different flow information for the different branches of an `if`.

4.5 Analyses for Language Extensions

One of the goals of the MCLAB project is to support modularly defined language extensions (as illustrated by the extension for ASPECTMATLAB [20]). Extensibility is also one of the goals for the design of MCSAF. The framework has been

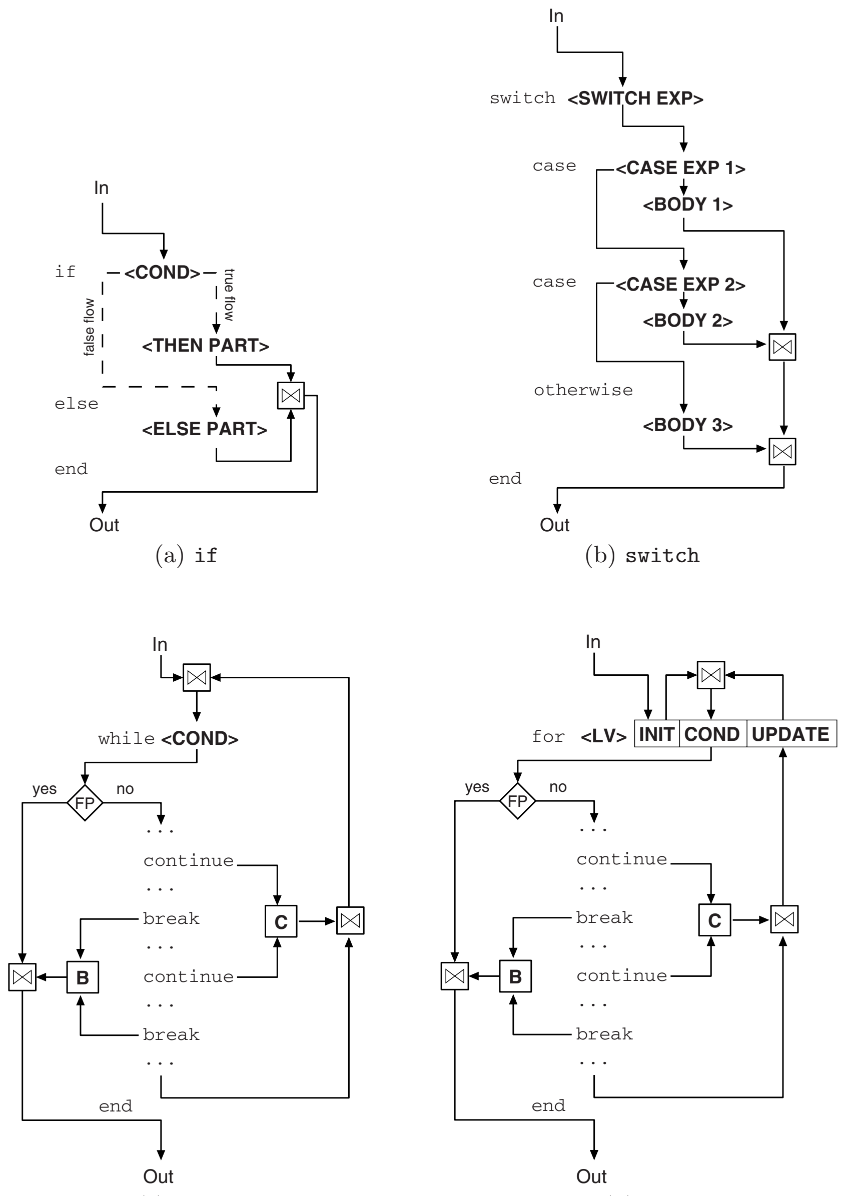


Fig. 16. Diagrammatic illustration of structural traversal rules

designed to support three kinds of extensions: (1) adding new syntactic nodes that are desugared before analysis; (2) adding nodes that need to be included in existing and new analyses; and (3) adding nodes that require new kinds of complex flow analysis - for example a new kind of loop. The framework comes with three example language extensions, one for each type. Although the second and third kind require slightly more glue code, all three kinds of extensions can be accomplished by defining new pieces and reusing the previous code.

5 Related Work

The McLab Static Analysis Framework is an extensible framework for creating static analyses for the Matlab language. This is the first open framework created for analyzing Matlab.

Soot [21] is one example of an optimization framework for Java which was developed by our research group for over a decade. McSaf emulates many of the positive features of Soot, including having a well-defined IR and a flexible and easy-to-use flow analysis framework. However, McLab solves very Matlab-specific problems in the IR design and uses a structured traversal-based approach (rather than a graph-based approach) to the flow analysis framework. So, in the end we found our experience with Soot helped us know the goals for the McSaf project, but developing a framework for Matlab required solving very different problems. Soot has enabled a lot of research for Java and we hope that McSaf will do the same for Matlab.

The JastAdd toolkit is designed for creating extensible compilers [2,12] and was also used in the development of McLab and McSaf. One feature of JastAdd that was not discussed in great detail in this paper is its attribute grammar system. JastAdd allows a developer to define attributes as part of the AST grammar specification. These attributes are effectively functions operating on the AST nodes. They can be used to propagate information through the tree and they can even be defined in a circular fashion. The JastAdd system provides a fixed-point computation for calculating the results of such circular attributes. JastAdd's attribute system provides a low-level means of performing flow analysis on an AST. It is up to the compiler writer to use these tools and to take the semantics of the language they are implementing into account, in order to create any meaningful analyses. It isn't a full dataflow analysis framework. However, some work [17] has been done to implement flow analysis for Java using the JastAdd extensible Java compiler [11].

In the past, there has also been some work towards open Matlab systems such as Octave [1]. These systems concentrate on front-ends and interpreters and so do not include analysis and optimization frameworks.

There has also been work on optimizing Matlab. The FALCON project [19,7] aimed to compile Matlab code into Fortran. Falcon focuses on type inference and code inlining to produce Fortran code. The Magica tool [14,13] focuses on type inference for matrix operations and functions. It not only infers the intrinsic type of matrices, such as `int32`, `double`, or `char`; but also matrix sizes and shapes. Magica is part of a larger Matlab compiler project, and is used for performing code optimizations. MaJIC incorporates a Just-In-Time(JIT) compiler component [4]. This allows it to achieve speedups similar to those produced by Falcon, without sacrificing the interactive nature of Matlab. These projects differ from McSaf in that their main goal was to improve the performance of Matlab programs. McSaf, on the other hand, was created with the goal of creating an open tool for researching compiler techniques in scientific programming. In fact the techniques used in these other projects could have been implemented using McSaf.

6 Conclusions and Future Work

In this paper we have presented the McSaf framework, an open source toolkit for developing analyses for Matlab. The goal of the toolkit is to enable compiler researchers to develop a wide variety of analyses which can target optimizations, source-to-source translators, and software engineering tools such as refactoring tools.

The toolkit includes a suite of simplifications which convert the high-level AST to a lower-level AST which is designed to expose important Matlab-specific semantics and to provide a simple IR which eases the burden of developing new analyses rules. Our simplifying transformations have been implemented along with a dependency structure so that it is easy for a user to enable only some simplifications or add new simpflications, while at the same time ensuring that all prerequisite simplifications have already been performed.

The heart of the toolkit is the support for different kinds of flow analysis driven by a collection of traversals, including a depth-first traversal which works well for flow-insensitive analyses and a family of structure-based traversals for forward and backward flow-sensitive analyses.

Developing this toolkit was not just an engineering exercise. At each step we had to understand the very Matlab-specific requirements and ensure that our approach captured the correct semantics in a way that made the IR and flow analysis framework easy to use. Indeed, we found the entire exercise much harder than we had anticipated, with the Matlab semantics often going against our expectations or enforcing some constraint that was not obvious.

Our goal was to make a toolkit that is easy to understand, and which is easy to extend. We don't want the compiler/analysis/tool developer to have to worry about all of the details of Matlab, but rather concentrate on using a well-structured object-oriented toolkit that provides an IR and analysis framework which has dealt with the messy details.

The McSaf toolkit has been implemented in Java as part of the McLab system and numerous analyses have already been implemented using it, including those mentioned in Figure 1. To date we have found that toolkit users find it quite easy to use and we look forward to feedback from users to help us improve it further.

Our intention is to continue using the toolkit in our own back-ends and tools, and we hope that other compiler researchers will also be able to use the toolkit as a simple and low overhead way to apply their research to the very popular Matlab language.

References

1. GNU Octave, `http://www.gnu.org/software/octave/index.html`
2. JastAdd, `http://jastadd.org/`
3. McLAB (2011), `http://www.sable.mcgill.ca/mclab/`
4. Almási, G., Padua, D.: MaJIC: compiling MATLAB for speed and responsiveness. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 294–303. ACM, New York (2002)
5. Casey, A., Hendren, L.: MetaLexer: A modular lexical specification language. In: AOSD, pp. 7–18 (2011)
6. Chevalier-Boisvert, M., Hendren, L., Verbrugge, C.: Optimizing MATLAB through Just-In-Time Specialization. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 46–65. Springer, Heidelberg (2010)
7. Derose, L., De Rose, L., Gallivan, K., Gallivan, K., Gallopoulos, E., Gallopoulos, E., Marsolf, B., Marsolf, B., Padua, D., Padua, D.: FALCON: A MATLAB Interactive Restructuring Compiler. In: Huang, C.-H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1995. LNCS, vol. 1033, pp. 269–288. Springer, Heidelberg (1996)
8. Doherty, J.: McSAF: An extensible static analysis framework for the MATLAB language. Master's thesis. McGill University (December 2011)
9. Doherty, J., Hendren, L., Radpour, S.: Kind analysis for MATLAB. In: OOPSLA (2011)
10. Dubrau, A.: Taming MATLAB. Master's thesis. McGill University (April 2012)
11. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, pp. 1–18. ACM, New York (2007)
12. Ekman, T., Hedin, G.: The JastAdd system – modular extensible compiler construction. Science of Computer Programming 69(1-3), 14–26 (2007)
13. Joisha, P.G.: A Type Inferenence System for MATLAB with Applications to Code Optimization. Ph.D. thesis, Northwestern University (2003)
14. Joisha, P.G., Banerjee, P.: The MAGICA Type Inference Engine for MATLAB. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 121–125. Springer, Heidelberg (2003)
15. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation. In: CGO, pp. 75–88 (2004)
16. Li, J.: McFor: A MATLAB to FORTRAN 95 compiler. Master's thesis. McGill University (August 2009)
17. Nilsson-Nyman, E., Hedin, G., Magnusson, E., Ekman, T.: Declarative intraprocedural flow analysis of Java source code. Electron. Notes Theor. Comput. Sci. 238, 155–171 (2009)
18. Radpour, S.: Understanding and Refactoring MATLAB. Master's thesis, McGill University (April 2012)
19. Rose, L.D., Padua, D.: Techniques for the translation of MATLAB programs into Fortran 90. ACM Trans. Program. Lang. Syst. 21(2), 286–323 (1999)
20. Toheed Aslam, A.D., Doherty, J., Hendren, L.: AspectMatlab: An aspect-oriented scientific programming language. In: AOSD, pp. 181–192 (March 2010)
21. Vallée-Rai, R., Gagnon, E.M., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)