MCSAF: AN EXTENSIBLE STATIC ANALYSIS FRAMEWORK FOR THE MATLAB LANGUAGE

by
Jesse Doherty

School of Computer Science
McGill University, Montréal

August 2011

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

Abstract

MATLAB® is a popular language for scientific and numerical programming. Despite its popularity, there are few active projects providing open tools for MATLAB related compiler research. This thesis provides the *Mc***LAB** Static Analysis Framework, *Mc***SAF**, the goal of which is to simplify the development of new compiler tools for MATLAB.

The *Mc***LAB** project was started in order to develop such tools in the hopes of attracting further research. The goal of the project is to provide an extensible compiler toolkit for MATLAB and scientific programming. It is intended to explore the compilation challenges unique to MATLAB and to explore new language features that could help scientific programmers be more productive. One piece of functionality that is particularly important for compiler research is the ability to perform static analysis. Without the information provided by static analyses, program transformations and optimizations, and automated programmer feedback would not be possible.

In order to make the development of static analyses simpler, this thesis contributes a framework for creating static analyses for the MATLAB language. This framework is intended to make writing analyses easier by providing core functionality and API for developing such analyses. It also aims to make analysis development easier by providing an intermediate representation called *Mc***LAST**, which provides simpler syntax and explicitly exposes some of MATLAB's semantics. In order to give analysis writers a head start, some example analyses are provided. These include simple analyses intended to demonstrate the use of the framework, and some more complicated analyses that provide basic semantic information about MATLAB programs.

In addition to the framework making development of analyses simpler, *Mc***SAF** is also designed to be extended to new language features. Not only can the framework be extended,

but existing analyses can also be extended. This allows work that was previously done for analyzing MATLAB code to be applied to future language extensions.

Résumé

MATLAB® est un langage de programmation science et numérique utilisé autant en industrie que dans le milieu académique. Malgré cette popularité, peu de project de rercherhe on été entreprise dans le but de produire une suite de compilation pour MATLAB. Cette thèse contribue le *Mc***LAB** Static Analysis Framework, *Mc***SAF**, qui a l'objectif de simplifier le développement des nouveaux outils de compilation pour MATLAB.

Le projet *Mc***LAB** fait suite à ce manque, dans l'espoir d'attiser les recherches sur ce sujet. L'objectif principale ce résume au développent d'une trousse de compilation extensible pour MATLAB et les langage de programmation pour science. Le projet est motivé par des défis de compilation unique à MATLAB, et par lexploration de nouvelles structures syntaxical améliorant lexpérience de programmation scientifique. L'une des fonctionnalité cher au domaine de la compilation est l'habilité a performer des analyses statique de programme. Sans ces informations que nous procures lanalyse statique, une grande partie des transformations et autres optimisations désiré lors du processus de compilation ne serait pas possible.

Pour rendre la développent des analyses statique plus simple, cette thèse contribue un cadre pour créer des analyses statique pour le langage MATLAB. L'objectif de ce cadre est de rendre la programmation des analyses plus simple en fournissant les fonctionnalités de base et une API pour développer de telles analyses. Un autre objectif est de rendre le développent des analyses plus simple en fournissant une représentation intermédiaire, *Mc***LAST**, qui fourni une syntaxe plus simple est qui expose les sémantiques de MATLAB. Pour aider les écrivains d'analyse, quelques exemples d'analyse sont fournis. En plus, quelques analyses utiles sont également fournis. Ces analyses fournissent des informations de base reliée à les sémantiques de MATLAB. Ils ont des application partout dans le projet.

L’objectif final du cadre est d’être extensible. Le framework doit fonctionner avec des nouvelles structure de langue. Sa veut dire que les programmeur peuvent créer des nouvelles analyses pour ces extension, et que les analyses qui existaient pour le langage de base, peut être adapter aux nouvelles structure.

Acknowledgements

I would like to thank my supervisor, Laurie Hendren, whose high standards and demand for clear descriptions have helped shape my work and my writing.

I would also like to thank the entire *Mc***LAB** team. In particular I would like to acknowledge contributions by the following members: M.Sc. student Soroush Radpour, who has taken responsibility for continued development on the Kind Analysis; M.Sc. graduate Toheed Aslam for being the first to use the Kind Analysis, and for helping to inspire its creation; and finally, M.Sc. student Anton Dubrau for being the first major user of *Mc***SAF**, and for putting up with my constant delays.

Of course I would also like to thank my friends and family, whose support and constant prodding provided the motivation to complete this thesis as soon as possible.

Finally, I would like to thank my loving wife Jessica Ganten. She has put up with my constant preoccupation with MATLAB and compilers, and with the occasional minor depression that resulted from learning some new disturbing "feature" of MATLAB.

This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Table of Contents

Appendices

List of Figures

List of Tables

List of Listings

Chapter 1
Introduction

MATLAB is a popular programming language among scientists and engineers. It provides high-level matrix operations that are useful to scientists. Its dynamic type system can also make code more natural to write, allowing programmers to avoid declaration statements and reuse variables when appropriate. MATLAB also includes a large collection of libraries and toolboxes that add useful features. All of these features give MATLAB a low initial learning curve and good productivity. This is reflected in MATLAB's very large and increasing user base. 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.[1]

Despite this initial ease of adoption, once programs become more complex or performance becomes a concern, MATLAB ceases to be easy to use. For instance the dynamic semantics can make programs difficult to understand and can impact performance. To overcome performance issues a programmer must write their code in such a way as to take advantage of MATLAB's core matrix operations. This can lead to code that is even more difficult to understand.

Some of the responsibility for performance can be taken on by an optimizing Ahead-of-Time (AOT) or Just-in-Time (JIT) compiler. This can let a programmer focus on expressing their ideas in a natural and clear way. Static tools can also be used to help a programmer

[1] From www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf.

better understand their programs. These tools could provide information about the programs they are writing, such as where possible performance or ambiguity problems may occur. There can also be tools to aid in code maintenance by providing automated refactoring features. New language features could be added to give programmers new tools and abstractions to use, making it easier to express some ideas.

In order for a compiler to perform optimizations or to provide feedback to programmers, it must analyze the program source code. Making such analyses easier to write requires a simple, well-defined Intermediate Representation (IR) and analysis framework. This framework should also allow new language extensions to use and adapt existing analyses. In spite of MATLAB's popularity, and the apparent need for static analysis, there has been no publicly available framework for creating static analyses for the MATLAB programming language. This thesis provides such a framework, the *Mc***LAB** Static Analysis Framework, or *Mc***SAF**. *Mc***SAF** was created as part of the *Mc***LAB** project to satisfy these requirements. The goal of the framework is to make new analyses easy to write and easy to extend to new language features.

The *Mc***LAB** project was started to create and explore compiler tools for scientific computing. The goal is to improve performance and usefulness of scientific programming languages, with a focus on MATLAB. Further details concerning the *Mc***LAB** project are presented in *Section 2.1*.

A diagram providing an overview of the project and main contributions of this thesis is given in *Figure 1.1*. The main contributions of this thesis are outlined in the grey area of the diagram. The basic structure of the *Mc***LAB** system is as follows. MATLAB source code is taken in by the front-end, which produces an abstract syntax tree (AST) representing the input. We refer to this AST as *Mc***AST**. Prior to the contributions made by this thesis, *Mc***AST** was fed directly into one of three back-ends. These back-ends either produce MATLAB source, FORTRAN source, or execute the program. The contributions of this thesis add extra steps between the front-end and the back-ends. *Mc***AST** is fed into a static analysis in order to determine basic information about the program. This information is then used to simplify the AST into a lower level representation of the original input, called *Mc***LAST**. *Mc***LAST** can then be further analyzed, producing new information. Finally, *Mc***LAST** and the analysis information are given to a back-end to produce the desired output.

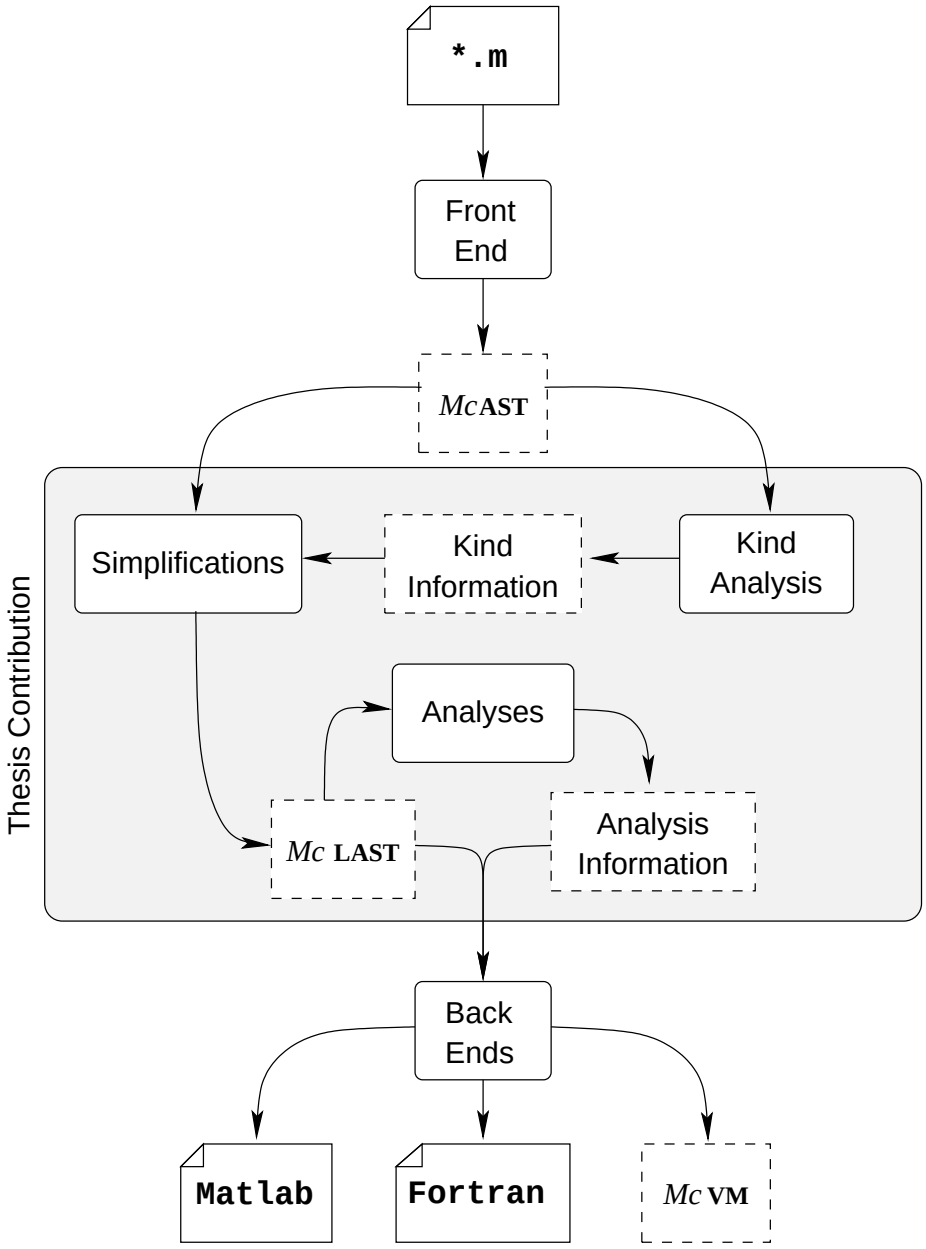


Figure 1.1 The *Mc*LAB system

This thesis focuses on the design and implementation of *McSAF*.

1.1 Contributions

The goal of this thesis is to provide a framework for creating static analyses for the *McLAB* project. To this end, this thesis makes three main contributions. First, we have done an exploration of MATLAB's semantics. This was a necessary first step because the analysis framework needs to accurately capture MATLAB's semantics. Since MATLAB has no formal specification, this involved interpreting documentation and experimenting with the current version of the official MATLAB environment.

Our second contribution is a well defined and simplified intermediate representation and procedure for producing it. The *McLAB* front-end produces an abstract syntax tree after parsing a program. *McAST* is too complex and causes the analysis writing process to be overly complicated. The simplified *McLAST* also makes some of MATLAB's semantics more explicit. This takes some of the burden of understanding MATLAB's semantics off of the analysis writer.

Our final contribution is the static analysis framework. The framework is designed to make analyses easy to write. It is also intended to accommodate language extensions. It does this by allowing existing analyses to be incorporated into new language extensions with minimal effort.

1.2 Outline

This thesis is split into 8 chapters (including this introductory chapter). *Chapter 2* gives background information necessary for this thesis. This includes a discussion of important tools used in the *McLAB* project. It also includes a discussion of the MATLAB language. In *Chapter 3* we introduce the IR. First we introduce the high-level AST, *McAST*, produced by the front-end. Next we describe *McLAST*, a simplified IR designed as a contribution of this thesis. *Chapter 4* continues the discussion of the IR by describing the transformations used to produce *McLAST* from *McAST*. *Chapter 5* discusses the final contribution of this thesis, the analysis framework and description of some example analyses. This discussion

is continued in *Chapter 6* with the description of the extensibility of the framework. Finally, *Chapter 7* discusses related work and *Chapter 8* presents our conclusions.

Chapter 2
Background

This chapter provides background information that is helpful in understanding the remainder of this thesis. We start with a brief description of *McLAB*, the project to which *McSAF* belongs. Next we give an overview of some parts of MATLAB that are not trivial to understand.

2.1 The *McLAB* Project

McLAB[1] is an extensible compiler framework for MATLAB. It consists of a front-end, a static back-end, and various code generation targets. These targets include pretty printing MATLAB source, FORTRAN source generated from inputted MATLAB code, and a virtual machine(VM) and just-in-time(JIT) compiler.

The *McLAB* project was created to explore compiler techniques and new language features in the domain of scientific computing. MATLAB is a very popular language among scientists and engineers, but due to it's closed source and proprietary nature it is difficult for the compiler community to explore and experiment with it. *McLAB* provides an open source framework that allows such work to be done.

One of *McLAB*'s goals is to explore new language extensions. To this end, the *McLAB* project has been created with extensibility in mind. It uses tools and designs that allow for

[1] http://www.sable.mcgill.ca/mclab/

new features to be added to the core language. As part of the project, the first language extension was created. This extension is called AspectMATLAB [TAH10]. As the name suggests, it adds aspect-oriented programming to the MATLAB language.

2.2 The MATLAB language

The MATLAB programming language is a high-level dynamic language for numerical computation. MATLAB's goal is to provide programmers with easy access to powerful numerical procedures. It does this by having a rich library of high-level procedures and allowing a flexible programming style. The flexible programming style is supported by dynamic semantics such as dynamic typing.

MATLAB is a closed sourced, proprietary programming environment. The language is defined by the current reference implementation[2] and official documentation[Mata]. There is no publicly available formal specification. In addition, the language has evolved fairly organically. Over time, new features have been added and language semantics have been tweaked. This has led to an eclectic mix of language features and confusing semantics.

We will list and describe some of the MATLAB's features that are relevant to the content of this thesis. For more about the MATLAB language, see the official site[Matb].

Ambiguous Syntax

MATLAB's syntax does not differentiate between function calls and array accesses. For example, the expression `x(i,j)` could be either a call to a function named `x`, with arguments `i` and `j`, or it could be an access to index `(i,j)` of array `x`. This cannot be distinguished syntactically.

Dynamic Name Binding

An identifier use can refer to either a function call or variable access. Which kind of use cannot be determined syntactically. Furthermore, it cannot be distinguished purely statically, it actually becomes a run-time property. This issue leads to the need for Kind Analysis described as *Section 5.2.5*.

[2] For the purposes of this project, we are using MATLAB version 7.12.0.635 (R2011a)

Dynamic Types

Types and matrix shapes are dynamic. Matrices can even be accessed with any number of dimensional indices, independent of the number of dimensions the matrix was created with. Matrices will also grow automatically when assigning to an index that is out of bounds. Matrices are by definition a homogeneous data structure. They can only contain one type of data. MATLAB also has heterogeneous data structures such as cell arrays and structures. The types contained by these data structures are of course determined at run-time, and can change through the life of the data structure.

For Loops

All `for` loops in MATLAB are for-each loops. The loop domain is the columns of the result of the loop domain expression, treated as a 2 dimensional matrix. More detail is given in *Section 3.2.2*.

End Expression

The `end` expression represents the last index of a dimension of an array indexed with a certain number of dimensions. We say the `end` binds to the array it is being used to index. The complexity arises because the `end` expression does not have to appear directly as the index expression, it can appear as a sub-expression. For example in the following expression:

```
A(2,f(end))
```

If we assume `A` is an array with value `[1,2,3;4,5,6;7,8,9]` and `f` is a function that computes $f(x) = x - 1$, then the `end` will evaluate to `9` and `A` will be indexed with `(2,8)`. Recall that for an array `A` with N dimensions, indexing `A` with $n < N$ dimensions will cause the n^{th} dimension to be interpreted as having a size equal to the product of the sizes of dimensions n through N. The `end`'s value depends on what array it is bound to, which dimension it is being used to index and how many dimensions are being used. This expression has implications for determining the kind of identifiers, and as such receives special treatment in the Kind Analysis. There is partial documentation available for this expression in MATLAB's official

documentation[3].

Comma-Separated Lists Comma-Separated lists(CSLs) are primarily a syntactic element in MATLAB. As their name suggests, they are lists of expressions separated by commas. They are used as input and return parameter lists. For example the statement `[a,b]=foo(n,x,y+4,m);` uses two CSLs, one for the arguments to `foo` and one to specify where to assign the two return values from calling `foo(...)`. However, the evaluation of some expressions can result in what we call CSL expansion. This means that an individual expression in a CSL, when evaluated, will expand at run-time to be multiple entries in that list. This results in the exact number of input or return parameters being known only at run-time. For example, the following code is equivalent to the previous assignment statement.

```
1 c1 = {x,y+4};
2 c2 = cell(1,2);
3 [c2{:}]=foo(n,c1{:},m);
4 [a,b]=c2{:};
```

Note, on line 3, the use of `c2` in the return parameters and `c1` in the input parameters. Notice that `c1` only covers two of the input parameters. The results of `c1{:}`, which are the values of `x` and `y+4`, will be incorporated into the input parameter CSL. The exact number of input and output parameters cannot be determined simply by inspecting this line. For more information regarding CSLs, see MATLAB's documentation[4] on the subject.

[3] http://www.mathworks.com/help/techdoc/ref/end.html
[4] http://www.mathworks.com/help/techdoc/matlab_prog/br2js35-1.html

Chapter 3
Intermediate Representations

Intermediate representations (IRs) are an important components of a compiler. They are used to represent the program at various stages of compilation. Compilers have multiple levels of IRs, each suited to different tasks. For instance, a high-level IR is useful in the front-end of a compiler. Such an IR will closely match the original structure of the input program, which makes it simpler to generate directly from source code. However, a high-level IR can be cumbersome to work with when performing analyses or transformations. A lower-level IR would be more appropriate for these tasks. A lower-level IR will contain simpler expressions and statements and will explicitly expose important semantics. This will simplify the task of writing analyses or transformations by reducing the number of cases that need to be handled.

The *McLAB* project uses tree-based IRs, taking the form of Abstract Syntax Trees(ASTs). A tree-based IR was chosen over a graph-based one to facilitate extensibility and maintain high-level structural information throughout the compilation process. Such structural information includes loops and `if` statements. These structures can be represented in a graph-based IR by reducing them to program jumps. That approach can be useful if the source language contains `goto` statements which allow the creation of arbitrary control flow graphs. Since MATLAB does not have `goto` statements such a reduction was not needed and so a structural, tree-based IR was chosen.

The *McLAB* project has two IRs. The first is a high-level AST called *McAST*. The second a low-level AST called *McLAST*, the design of which is a contribution of this

thesis. *McAST* is produced by the front-end. A simplification procedure is then applied to it, to produce *McLAST*. This process is depicted in *Figure 3.1*.

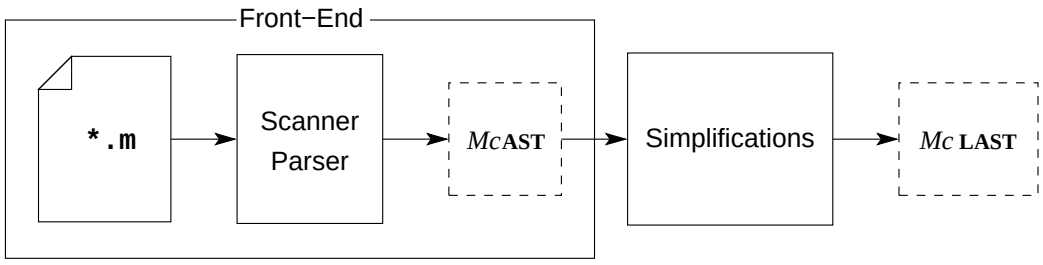


Figure 3.1 Front-end and *McLAST* generation

In this chapter we describe the structure of *McAST* and the structure and design of *McLAST*. The description of *McAST* is given first in *Section 3.2*. This is to give a good base for understanding the simplified IR. It includes a high-level description of the *McAST* structure and discussion of *McAST*'s abstract grammar. A description of *McLAST* is then given in *Section 3.3*. This description includes a grammatical presentation and a discussion of design choices. *Chapter 4* describes the simplification procedure used to generate *McLAST*.

3.1 Formalisms

In this chapter we rely on two formalisms to describe the IRs. In this section, we will describe these formalisms.

3.1.1 JastAdd Abstract Grammars

The first formalism used is the JastAdd [EH07b] abstract grammar specification format. This format is used to define the AST structure of the IRs. The JastAdd system uses this specification to generate Java classes representing the different AST node types. These classes will export a well defined API for construction and traversal determined by their specification. This means that to understand the AST class structure and API, it is sufficient to understand JastAdd specifications. The abstract grammars are specified in files with

extension `.ast`. JastAdd is a Java-based tool, and as such, these specifications incorporate some Java syntax and types.

As we said, these specifications are used to define AST nodes. There are two types of nodes that can be defined: abstract nodes and concrete nodes. An abstract node results in an abstract class, which can't be instantiated. A simple node of this type can be specified as follows:

```
abstract Program;
```

Here we are defining the Program node for *McAST*. More information on this node is available in *Section 3.2.3*.

A simple non-abstract node is defined by omitting the `abstract` keyword.

A node can also be defined to be a subtype of another node. This subtype relationship is directly mapped to JAVA's subclass relationship. It is specified in the following way:

```
EmptyProgram : Program;
```

This example specifies the concrete node `EmptyProgram`, which is a type of `Program`, so we specify it as a subtype. Note that an abstract node could also be a subtype of another node.

A node can also be specified to have children. There are four types of children possible: single child, optional child, list child, and typed token child. All children, except the typed token child, will be AST node types. The following example from the JastAdd documentation, slightly modified, has one of each child.

```
E: A ::= A [B] C* <D:String>;
```

This specifies a concrete node called `E`, which is a subtype of node type `A`. `E` has four children, one of type `A`, an optional child of type `B`, a list child containing nodes of type `C`, and a typed token child named `D` of type `String`. The typed tokens allow AST nodes to contain children that use Java types, rather than simply AST node types. Note that abstract nodes can also be specified with children. Also note that since the node subtype relationship matches the Java subclass relationship, when a node is a subtype of another node that has children, the subtype node will inherit these children.

Children can also be specified with names. This allows JastAdd to generate a more

meaningful API for the nodes. An example of a node with named children is the assignment statement, which defines a left- and right-hand side.

```
AssignStmt : Stmt ::= LHS:Expr RHS:Expr;
```

3.1.2 Grammar Specifications

In *Section 3.3* we describe the restrictions that *McLAST* has over *McAST*. Some of these restrictions are in fact only logical restrictions. By this we mean, the AST specification for *McLAST* is more permissive than the actual specification for *McLAST*. We describe these added restrictions by relying on a more standard grammar specification. The given grammar does contain some syntactic elements, but should not be interpreted as a grammar for parsing. These specifications are used to define restrictions on the actual AST definition. For example, the AST definition for **`if`** statements in *McLAST* is the same as the definition for *McAST*.

```
IfStmt : Stmt ::= IfBlock* [ElseBlock];
IfBlock ::= Condition:Expr Stmt*;
ElseBlock ::= Stmt*;
```

This definition allows multiple **`elseif`** blocks and an optional **`else`**. It also allows arbitrary expressions in the **`if`** condition.

McLAST defines a more restrictive **`if`** statement, but doesn't define new AST nodes to enforce it. These restrictions are specified in the following grammar.

$$
\begin{aligned}
IfStmt \quad &:= \quad if\,(CondExp)\,\texttt{Stmt}^{*}\,end \\
&\quad\;\; | \quad if\,(CondExp)\,\texttt{Stmt}^{*}\,else\,\texttt{Stmt}^{*}\,end
\end{aligned}
$$

This specification restricts an **`if`** statement to not allow **`elseif`**, and to restrict the condition expression, which is further specified in *Section 3.3.3*.

3.2 *McAST*

The front-end process produces *McAST* from given source code. *McAST* is the high-level IR being used by the *McLAB* project. It is the starting point for the simplified IR *McLAST*. The simplification procedure mentioned in *Figure 3.1* operates on *McAST*. There are also some basic static analyses that operate on it, in particular, the Kind Analysis described in *Section 5.2.5*. Describing the structure will give a good base for understanding *McLAST*'s structure and the simplification procedure. It will also give some context for understanding the complexities that *McLAST* exposes.

McAST will be described in a bottom-up fashion. The presentation is split into three levels, expressions, statements, and program structure.

3.2.1 Expressions

Expressions are the basic building blocks of programs. In *McAST* there are several different types of expressions. The top-level definition for expressions is given in *Figure 3.2*. It defines four types of expressions; literals, LValues, unary operations and binary operations. There are several other types of expressions that are described later in this section.

```
abstract Expr;
abstract LiteralExpr : Expr;
abstract LValueExpr : Expr;
abstract UnaryExpr : Expr ::= Operand:Expr;
abstract BinaryExpr : Expr ::= LHS:Expr RHS:Expr;
...
```

Figure 3.2 *McAST* top level expression definition

Literal Expressions

The literal expressions are the simplest expressions available. They represent numerical literal values and string literal values. For example `42` appearing in source code would be represented by an instance of `IntLiteralExpr`, which is one possible literal expression.

LValue expressions

LValue expressions are expressions that can appear on the left-hand side of assignments. Since they are expressions, they can also appear in any other location an expression can appear. A section of specification for these expressions is given in *Figure 3.3*.

```
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*;
```

Figure 3.3 *McAST* LValue expression definition

The most basic LValue expression is the name expression, which simply consists of a name.

The parameterized, cell index and dot expressions all have similar structures. They each contain a target that can be any expression. This allows for complex expressions such as `foo(2).bar(3)`. This expression is a parameterized expression with a single argument, `3`, and a complex target, `foo(2).bar`. The target is itself a dot expression, accessing the field `bar` and having a target that is a simple parameterized expression. The expression as a whole is interpreted as: access the third value in the array stored in the field `bar` of the structure stored in the second entry in the array `foo`.

The parameterized and cell index expressions also contain lists of arbitrary expressions for their arguments. This allows for even more complex expressions such as `foo(FOOBAR()).bar(FOOBAR())`. This expression has a similar structure to the previous example, but instead of simple literal values as arguments, the two parameterized expressions have another parameterized expressions as an argument.

The matrix expression is truly an LValue when it is a single row containing non-matrix expression LValues, such as `[a,b]`. It can also be used to represent matrix definitions of the form

```
    [1,2,3; 4,5,6]
```

where rows are delimited by semicolons and elements of rows by commas. The reason this is considered an LValue expression is discussed further in *Section 3.2.2* when describing assignment statements.

Recall that the cell index and dot expressions also have a special implicit property. When used in a Comma-Separated List (CSL), they can undergo CSL expansion, as described in *Section 2.2*.

Unary and Binary expressions

Unary and binary expressions represent various operations. A portion of the specification for these expressions is given in *Figure 3.4*. The unary and binary expressions are each defined by an abstract node defining the left- and right-hand sides. Each type of unary or binary expression is then specified by a concrete node that extends `UnaryExpr` or `BinaryExpr`.

```
abstract UnaryExpr : Expr ::= Operand:Expr;
UMinusExpr : UnaryExpr;
UPlusExpr : UnaryExpr
...
abstract BinaryExpr : Expr ::= LHS:Expr RHS:Expr;
PlusExpr : BinaryExpr;
MinusExpr : BinaryExpr;
...
```

Figure 3.4 *McAST* unary and binary specification portion

Obviously the unary and binary expressions will contain arbitrary expressions as their operands. This can lead to similar complexities to ones seen with some LValue expressions.

Remaining Expressions

The specification for the remaining expressions is given in *Figure 3.5*.

The range expression represents the colon notation for defining range vectors. An example of these expressions is `1:2:10`, which evaluates to a vector contain all odd numbers from 1 to 10. The expression contains a lower and upper bound and an optional increment. These can all be arbitrary expressions.

```
RangeExpr : Expr ::= Lower:Expr [Incr:Expr] Upper:Expr;
ColonExpr : Expr;
EndExpr : Expr;
CellArrayExpr : Expr ::= Row*;
FunctionHandleExpr : Expr ::= Name;
LambdaExpr : Expr ::= InputParam:Name* Body:Expr;
```

Figure 3.5 *McAST* miscellaneous expressions

The colon expression is used only as an argument to a parameterized or cell index expression and only for indexing. The expression represents a range from 1 to the size of the dimension it is indexing.

The **end** expression has a similar use to the colon expression. It represents the last index of the dimension of the array in which it is used. As explained in *Section 2.2*, it is more complex than the colon expression because it can be used as a sub-expression in the index expression. To summarize the complexity, an **end** expression binds to a particular array access in a non obvious way. Furthermore it has implications in the Kind Analysis described in *Section 5.2.5*.

The function handle and lambda expressions are used to create function handles of named and anonymous functions. The lambda expression has a list of input parameter names and a single body expression.

3.2.2 Statements

Statements introduce various declarations, control flow and uses of expressions, including assignments. The simplest statement that involves expressions is the expression statement. The specification rules for this expression are given in *Figure 3.6*.

```
abstract Stmt;

ExprStmt : Stmt ::= Expr;
```

Figure 3.6 *McAST* expression statement

There are two types of declaration statements: global and persistent. The specification

for these is given in *Figure 3.7*. The global and persistent statements declare the names in the statement as either global or persistent variables, respectively.

```
GlobalStmt : Stmt ::= Name*;
PersistentStmt : Stmt ::= Name*;
```

Figure 3.7 *McAST* declaration statement

The assignment statement is defined in the specification section in *Figure 3.8*. It simply contains left-hand and right-hand side expressions. There should only ever be LValue expressions in the left-hand side expression. This is enforced by a weeding procedure. The semantics here are fairly straight forward: evaluate the right-hand side to get a value, evaluate the left-hand side to get a location and store the value in the location. A complexity arises from the fact that matrix expressions are LValue expressions. In this case, the matrix is expected to have only one row and only contain LValue expressions. The weeding procedure mentioned previously will enforce this structure. When a matrix expression appears on the left-hand side of an assignment expression, the assignment is interpreted as a multi-assignment statement. This means the expression on the right-hand side is expected to return multiple values, and those values are stored in the locations resulting from the LValue expressions in the matrix expression.

```
AssignStmt : Stmt ::= LHS:Expr RHS:Expr;
```

Figure 3.8 *McAST* assignment statement

The final types of statements are control flow statements. The specification for these statements is give in *Figure 3.9*.

The `break`, `continue` and `return` statements are simple control flow statements that operate in loops or functions. The **`while`** statement consists of a conditional expression and a list of statements representing the body of the loop. The conditional expression can be arbitrarily complex.

A **`for`** statement contains an assignment statement. This assignment statement is assumed to only contain a name expression on the left-hand side. We will refer to this name

```
BreakStmt : Stmt;
ContinueStmt : Stmt;
ReturnStmt : Stmt;
WhileStmt : Stmt ::= Expr Stmt*;
ForStmt : Stmt ::= AssignStmt Stmt*;
IfStmt : Stmt ::= IfBlock* [ElseBlock];
TryStmt : Stmt ::= TryStmt:Stmt* CatchStmt:Stmt*;
SwitchStmt : Stmt ::= Expr SwitchCaseBlock* [DefaultCaseBlock];

IfBlock ::= Condition:Expr Stmt*;
ElseBlock ::= Stmt*;

SwitchCaseBlock ::= Expr Stmt*;
DefaultCaseBlock ::= Stmt*;
```

Figure 3.9 *McAST* control statements

expression as the loop variable. The right-hand side can contain arbitrary expressions. The way it is interpreted is as follows. The value of the right-hand side is interpreted as an array and this array is treated as two dimensional. The loop will iterate over this two dimensional array, assigning each column to the loop variable for each iteration. These semantics are not obvious, but only becomes a problem because arbitrary expressions are allowed in the right-hand side of this assignment statement. *McLAST* forces the semantics to be made explicit in the code by limiting the right-hand side to be a range expression.

The **`if`** statement consists of a list of if-blocks and an optional else-block. Each if-block after the first is considered an **`elseif`**, with the else-block being the final **`else`**. The following is an example **`if`** statement with two if-blocks, and an else-block.

```
if E1
  body1();
elseif E2
  body2();
else
  body3();
end
```

The primary **`if`** is the first if-block, the **`elseif`** is the second if-block. This structure means that analyses and transformations have to deal with control flow with an arbitrary number

of branches. It would be much simpler if `if` statements only caused binary branching. The example could be rewritten to only use binary branching and would look like the following.

```
if E1
  body1();
else
  if E2
    body2();
  else
    body3();
  end
end
```

The `try` statement consists of the body of the try followed by the body of the catch.

A switch statement contains the expression being switched on, a list of case blocks, and a default block. The case blocks each have an expression to match against and a body of statements.

3.2.3 Program Structure

In order to define a program there must be some top level structure to the AST. This structure represents the different types of MATLAB files and gives a way of combining multiple files into one tree. The specification for this structure is shown in *Figure 3.10*. We however do not discuss classes in this thesis.

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

Figure 3.10 *McAST* program structure

The compilation units represent a collection of programs. A program represents a MATLAB file. A program can either be a script or a function file. A script simply contains a

list of statements. A function file is represented by a function list. The function list simply consists of a list of functions.

The definition of a function is slightly complex. The important parts of the definition are that it has a list of names representing the output parameters, a string for the name of the function, a list of names for the input parameters, a body of statements and a list of nested functions.

3.2.4 Overview

McAST has a number of sources of potential complexity. This includes arbitrarily complex expressions, the use of `end` expressions, `for` loops with complicated semantics, cumbersome `if` statement structure. Another important issue to note is there is no explicit array indexing or function call expressions. This is due to the ambiguity between function calls and array indexing. This issue cannot be completely solved statically and is discussed later is *Section 5.2.5*.

McLAST is intended to avoid some of the complexities present in the full *McAST*.

3.3 *McLAST*

The design of *McLAST* incorporates a number of simplifications that were found useful in the JIT and Fortran code generation. It also includes simplifications that enforce and expose MATLAB semantics and simplify analysis writing.

In this section we describe the design of *McLAST*. The description is separated into five sections, each dealing with different portions of the specification. Two of these sections deal with expressions and are grouped together. The other three deal with statements and are separate.

These sections also include grammar definitions and JastAdd specifications for those portions of *McLAST*. This is done to make the restrictions and additions more explicit and to give a reference for what can be expected from an instance of *McLAST* that respects those restrictions.

In addition to defining *McLAST*, there needs to be a way of generating it. In order to do

that, a collection of simplifications were implemented. These simplifications are described in *Chapter 4*.

The goal of this section is to express what portions of *McLAST* are more restricted, and to justify the need for these restrictions

3.3.1 Expressions

Most programming languages allow arbitrarily complex expressions. This allows programmers to write more concise code. However, such expressions in a low-level IR can be difficult for compiler and analysis writers to reason about. The portions of *McLAST* that deal with expressions will be discussed in two parts. First we discuss expressions that compute memory locations; we call such expressions LValue expressions. These expressions appear on the left-hand side of assignment statements and evaluate to the location that is being assigned to. Next we discuss expressions that compute actual values which we call RValue expressions.

LValue expressions

LValue expressions compute the memory locations that are assigned to by assignment statements. The complexity of these expressions comes from the chaining of indexing and field accesses, and the use of RValue expressions to compute index values. Such an expression is illustrated in the following example.

```
A(a+b,a).e(foo()) = value;
```

In this assignment statement the left-hand side is a complex LValue expression. The expression computes as follows:

1. the array `A` is indexed by the result of `a+b` and `a`
2. the indexing results in a structure with a field `e`, which is acceessed
3. this field contains an array, which is indexed by the value resulting from the expression `foo()`

It is the indexed location in the final array that is assigned to.

There are two sources of complexity in these expressions. The first is the inclusion of arbitrary RValue expressions and the second is the chaining of indexing and field accessing. Fortunately it is simple to restrict these expressions so that they can only include RValue expressions that are either literal values, or variables.

The issue of chaining array indexing and field accesses is more troublesome. Because MATLAB has no semantics for storing references to memory locations it is not possible to break apart such chains. One could add new language features to the IR to make it possible, but we decided this would add too much complexity. As a result chained indexing and field accesses are allowed in *McLAST*.

These restriction can be summarized as the following:

- LValues can contain only variable names, indexing, or field accesses
- the computation of indexes is restricted to literal values or simple variable access

These restriction are expressed in *Figure 3.11*. *Indexing* is essentially a restriction of the parameterized expression described in *Section 3.2.1*.

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

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

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

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

Figure 3.11 *McLAST* LValue grammar

These name expressions will also be required to be variable names, rather than function names. A name is considered a variable name if the Kind Analysis described in *Section 5.2.5* can determine it is a variable name.

RValue expressions

RValues expressions can be arbitrarily complex expressions in MATLAB.

To make matters more complicated, although MATLAB defines a simple precedence based left-to-right evaluation order[1] for expressions, there is a bug in the implementation that breaks this order[2]. We implement a correct left-to-right evaluation order.

This evaluation order allows *Mc*LAST to have a simple definition of allowed RValue expressions. In *Mc*LAST, an RValue can consist of at most one complex operation. These operations can consist of a function call, operator use, indexing, field access, or range expressions. Note however that *Mc*LAST does not allow scalar short-circuiting boolean operators (`&&`,`||`) at all. The reason for this is that such operators contain implicit control flow, which adds tremendous complexity. Further discussion of short-circuiting boolean operators and how they are removed are presented in *Section 4.10.1*.

Another complexity that arises from RValues is comma-separated list (CSL) expansion. This was described in *Section 3.2.1*. Expressions that can undergo CSL expansion are not necessarily obvious. To simplify this situation, we introduce the *CSLExpr*. The *CSLExpr* is essentially a name expression, but specifies that it might undergo CSL expansion. This new expression is included in the RValue description.

Finally, as was described in *Section 2.2*, **end** expressions cause some complexity. To simplify this complexity, **end** expressions are not allowed in *Mc*LAST. To replace them we introduce a new expression called the *EndCallExpr*. This expression is an explicit **end** expression that captures the bound expression being indexed, the number of dimensions being used, and what index the **end** appeared in. A simple example of a non explicit end expression is the following:

```
A(1,end,2)
```

[1] http://www.mathworks.com/help/techdoc/matlab_prog/f0-40063.html

[2] The current implementation (7.12.0.635 (R2011a)) contains a bug when evaluating expressions containing function calls with global side effects. Mathworks has accepted a bug report for this issue.

Where the array `A` is indexed with 3 dimensions where the second dimension is indexed with an **end** expression. In *McLAST* this would be represented as the following:

```
t = EndCall(A, 3, 2);
A(1, t, 2)
```

Note the use of the temporary `t` to avoid complex expressions.

The restricted RValue definition is shown in the grammar rule in *Figure 3.12*, and the extra node specification is given in *Figure 3.13*. In this rule *Name* represents names, *NameOrVal* represents names or values, and **OP** represents unary or binary operators other than short-circuiting boolean operators.

$$
\begin{aligned}
\textit{RValue} \;:=\; & \textit{NameOrVal} \\
\mid\; & \texttt{NameExpr}(\textit{NameOrVal}^{*}) \\
\mid\; & \texttt{NameExpr.Name} \\
\mid\; & \texttt{NameExpr.Name}(\textit{NameOrVal}^{*}) \\
\mid\; & \textit{NameOrVal}\ \textbf{OP}\ \textit{NameOrVal} \\
\mid\; & \textbf{OP}\ \textit{NameOrVal} \\
\mid\; & \textit{NameOrVal} : \textit{NameOrVal} : \textit{NameOrVal} \\
\mid\; & \textit{NameOrVal} : \textit{NameOrVal} \\
\mid\; & \texttt{CSLExpr} \\
\mid\; & \texttt{EndCallExpr}
\end{aligned}
$$

Figure 3.12 *McLAST* RValue grammar

```
EndCallExpr: Expr ::= Array:Expr <NumDim : int> <WhatDim : int>;
CSLExpr : NameExpr;
```

Figure 3.13 New *McLAST* RValue nodes

3.3.2 Multi-Assign Statements

One convenient feature of the MATLAB language is that it allows functions to have multiple return parameters. We will call assignment statements that accept multiple return values

multi-assign statements. This feature can be helpful for programmers but, like complex expressions, multi-assign statements can cause difficulty and complexity for compiler developers.

The left-hand side of assignments can have arbitrary expressions designating storage locations, or LValues. In the case of multi-assign statements the complexity is increased by having multiple, possibly related, arbitrary expressions appearing in one statement. CSL expanding expressions can also appear in multi-assign statements. An example of such a statement follows to illustrate the point.

```
[a,b(a),a,c.e(a)] = somefunction();
```

In this example we have four LValue expressions, two of which are simply the variable `a`, and the other two depending on the two different values that `a` gets assigned. Writing analyses to deal with general multi-assign statements could become very cumbersome. The solution is to restrict the types of LValues allowed in a multi-assign statement.

The restrictions we decided on are intended to make analyses easier to write. To that end we made multi-assign statements as simple as possible. In *Mc*LAST, a multi-assign statement can only contain simple variable names on the left-hand side. This is further restricted by allowing a name to appear at most once on the left-hand side. This means that an analysis need not be concerned with the complexities of multi-assign statements, and can treat it as a set of very simple assignments.

A special case for multi-assign statements has to do with dealing with CSL expressions. In order to assign to a CSL expression, it must be the only expression on the left-hand side of a multi-assign statement. In order to assign to a CSL expanding expression, it must also be the only expression on the left-hand side of a multi-assign statement and must have a CSL expression on the right-hand side.

These restrictions are represented in a grammar rule for assignment statements presented in *Figure 3.14*.

Note that the names must also be unique, a property not represented in grammar.

$$\begin{aligned} \textit{MultiAssignStmt} &:= [(\texttt{NameExpr}|\texttt{CSLExpr})^{+}] = \textit{RValue} \\ &\mid [\textit{LValue}] = \texttt{CSLExpr} \end{aligned}$$

Figure 3.14 *McLAST* multi-assign grammar

3.3.3 Conditional Expressions

The boolean expressions in conditional statements such as **`if`** and **`while`** statements are another source of complexity. The *McLAB* JIT, *McJIT*, will simplify conditionals by pulling any expression more complex than a simple name look-up out into a temporary. It was decided that such an aggressive simplification was not desirable in the front end. Instead we allow a single relational operator to be present to perform comparisons between variables.

This allows for a simple definition of conditional expressions, which is expressed in the *CondExp* grammar rule in *Figure 3.15*. This figure also includes the *WhileStmt* definition.

$$\textit{WhileStmt} := \textit{while}(\textit{CondExp})\texttt{Stmt}^{*}\textit{end}$$

$$\begin{aligned} \textit{CondExp} &:= \textit{NameOrVal} \\ &\mid \sim \textit{NameOrVal} \\ &\mid \textit{NameOrVal}\ \textbf{RelOp}\ \textit{NameOrVal} \end{aligned}$$

Figure 3.15 *McLAST* condition expression grammar

3.3.4 For Loops

A **`for`** loop in MATLAB is in fact a for-each loop. This means that the loop variable will iterate over the elements of some fixed sequence of elements. The complexity of a **`for`** loop comes from how the sequence can be defined.

A `for` loop is made up of two components, the loop variable assignment statement and the loop body. The left-hand side of the assignment statement is the loop variable and the right-hand side is an expression that defines the sequence to be iterated over. The expression can be any arbitrary RValue expression resulting in some type of array. The sequence that is iterated over is defined to be the columns of the right-hand side array value. What this means is that if the array is a row vector then each element in the vector is iterated over. If the array is a column vector, then only the single column is iterated over. If the array is a two dimensional matrix then each column in the matrix is iterated over. If the array is an array of greater than two dimensions, then it is treated as a two dimensional matrix in a way similar to what is described in *Section 2.2*.

These semantics are not obvious, and the arbitrary expressions can be difficult to analyze. What is desired is to have all `for` loops be in a simple form. We call this simple form, range `for` loops. Range `for` loops are `for` loops whose sequence expression can only be a range expression consisting of literals or variable uses. Such `for` loops are illustrated in the following example.

```
for i = 1:2:x
   BODY
end
```

Where `x` is a variable.

These simple range `for` loops can be defined by the grammar rules in *Figure 3.16*.

$$\textit{ForStmt} \;:=\; \textit{for} \quad \texttt{NameExpr} = \textit{SimpleRangeExpr} \quad \texttt{Stmt}^{*} \quad \textit{end}$$

$$\begin{aligned} \textit{SimpleRangeExpr} \;&:=\; \textit{NameOrVal} : \textit{NameOrVal} \\ &\mid\; \textit{NameOrVal} : \textit{NameOrVal} : \textit{NameOrVal} \end{aligned}$$

Figure 3.16 *McLAST* `for` loop grammar

A `for` loop that does not fit this pattern can be rewritten to do so. For example, the following contains a `for` loop that is not a range `for` loop.

```
A = [1,2,3; 4,5,6];
for i = A
  display(i);
end
```

This code loops through the columns of `A` and prints out the following:

```
i =
   1
   4

i =
   2
   5

i =
   3
   6
```

The **`for`** loop can be rewritten to be a simple range **`for`** loop. You would get the following code.

```
A = [1,2,3; 4,5,6];
for j = 1:3
  i = A(:,j);
  display(i);
end
```

This code has the same output, would be valid *McLAST* and exposes what the values of `i` will be.

3.3.5 If Statements

The **`if`** statement defined by *McAST* in *Section 3.2.2* has extra complexity due to inclusion of **`elseif`** in the actual structure. Fortunately an **`elseif`** is not a needed structure and can be simplified away. To this end, *McLAST* does not allow **`elseif`**, and instead has a simpler definitions for if statements. This is definition is given in *Figure 3.17*

$$
\begin{array}{rcl}
IfStmt & := & if(CondExp)\texttt{Stmt}^{*}end \\
 & | & if(CondExp)\texttt{Stmt}^{*}else\ \texttt{Stmt}^{*}end
\end{array}
$$

Figure 3.17 *McLAST* if statement grammar

3.3.6 Assignment Statements

Even with simplified expressions, assignment statements with arbitrary simplified expression on both the left and right side can be complicated. *McLAST* restricts assignment statements to simplify them a little more. An assignment statements can either be a multi-assign statement, or it has a variable on the left or a variable or literal on the right. This definition is given in *Figure 3.18*.

$$
\begin{array}{rcl}
AssignmentStmt & := & MultiAssignStmt \\
 & | & \texttt{Name} = RValue \\
 & | & LValue = NameOrVal
\end{array}
$$

Figure 3.18 *McLAST* assignment statement grammar

3.3.7 Check Scalar Statement

There are some operations represented in *McAST* and *McLAST* that have implied run-time checks. One check occurs in *McAST* but not *McLAST*. This check is to see if the operands of a scalar short-circuiting operator is in fact scalar. Since *McLAST* does not allow these operators and instead relies on explicit control flow, the implied check is not present. To allow us to remedy this situation, we introduce a new statement, the `CheckScalarStmt`. The definition for this statement is given in *Figure 3.19*.

```
CheckScalarStmt: Stmt ::= NameExpr;
```

Figure 3.19 *Mc*LAST assignment statement grammar

3.3.8 Validator

Since *Mc*LAST is specified by an AST definition and a collection of logical restrictions placed on the AST, it is possible to construct an AST that does not conform to *Mc*LAST's specification. The simplifications described in *Chapter 4* are designed to guarantee that the result does conform. However, if arbitrary transformations are performed on an AST that has been simplified, it may no longer conform. There should be a way to ensure that a given AST instance conforms to the *Mc*LAST specification.

To solve this problem, we have provided a validator. This validator traverses a given AST instance and ensures that it conforms to the grammar specification provided in this chapter. The validator only takes syntactic properties into account. It does not validate that a given name refers to a variable and not a function.

Chapter 4
Simplifications

In *Chapter 3* we explained and justified a collection of restrictions for the *McLAB* IR. These restrictions represent the definition of *McLAST*. The goal of *McLAST* is to provide a representation of a program that is simple for *McLAB* developers to work with. In order for this representation to be useful, there must be a way of generating the IR. The original structure of the front-end simply produced *McAST* when compiling. In order to generate *McLAST*, we have added a new phase to the front-end. This is illustrated in *Figure 4.1*. This new phase performs the simplifying transformations needed to produce *McLAST* from *McAST*. In this chapter we describe the organization and execution of the simplifications followed by a description of each transformation.

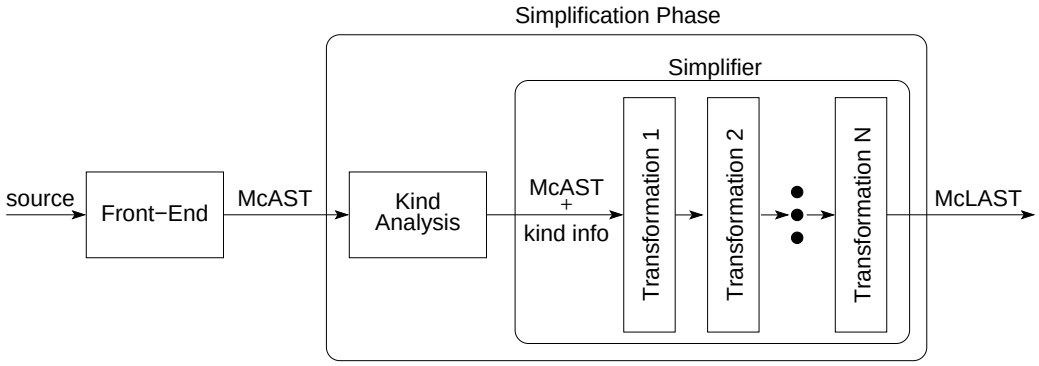


Figure 4.1 Front-end with IR generation

4.1 Organization and Execution

In order to make the transformation process more modular and simpler to implement we split it into several separate simplifications. Each of these simplifications relate to some aspect of the IR definition. Splitting the transformation process also has the benefit of allowing subsets of the process to be applied, rather than always requiring the full transformation. For example, the simplification that enforces simple range `for` loops can be applied in isolation. Of course, for some simplifications, it would not be correct to say the simplification was applied unless certain other simplifications were also applied. For example, one might want to apply the simplification that enforces no complex conditional expressions in `if` and `while` statements. In order for this transformation to be considered fully applied, one would have to expand element-wise short-circuiting expressions. This expansion is performed by a separate transformation. To have the simple conditional transformation applied correctly, one should first apply the transformation that expands element-wise short-circuiting expressions. The short-circuiting expansion could have been incorporated into the conditional simplification, but this would have increased implementation complexity. There also may be a need to have these short-circuiting expressions expanded without requiring the full conditional simplification.

4.1.1 Dependencies

In order to make it simpler to ensure that a given simplification is fully applied, they have been organized into a dependency graph. This graph is restricted to being a directed acyclic graph (DAG). The framework was not created with support for cyclic dependencies in mind.

To enforce the dependencies, a class called `Simplifier` was implemented. In addition, each simplification is implemented as a class extending `AbstractSimplification`. 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 will then perform a depth first traversal of the dependency DAG producing a list of sim-

plifications, 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.

The dependency DAG is shown in *Figure 4.2*. The rest of this chapter is spent describing each of these simplifications. To make the description simpler we present the simplifications in a dependency satisfying order. The diagram labels each simplification with the section where it is described.

If a new simplification is added in the future then two steps need to be followed to ensure that it works with all other simplifications. First all simplifications that the new one depends on must be listed in the `getDependencies` method of the new dependency. Second, each simplification that would now depend on the new simplification needs to add the new dependency to what is returned by their `getDependencies`.

4.2 Simple Assignment

One of the simplest transformation is to ensure single-assignment statements are simplified. This means that such assignments will not have a complex expression on both the left- and right-hand side. The simple assignment simplification ensures that all single-assignments have at most, either a complex expression on the left, or one on the right, but not both. Multi-assignment statements are dealt with in another transformation, presented in *Section 4.4*.

This simplification is very straight forward. It simply finds all non-multi assignment statements and checks if both sides are complex. If both sides are indeed complex, then it converts the statement into two assignments by introducing a new temporary variable. So for example, if we have the code:

```
E1 = E2;
```

Where E_1 and E_2 are both complex expressions, the simplification will produce the follow-

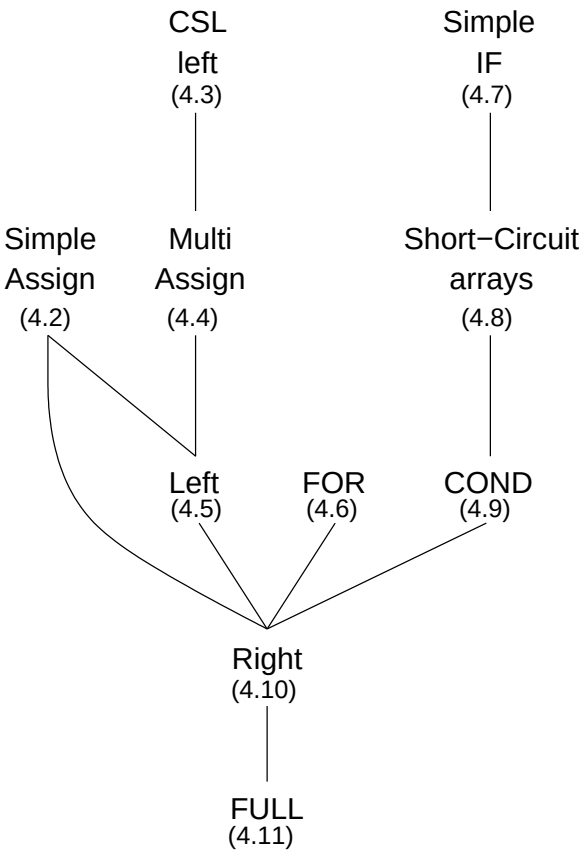


Figure 4.2 Dependency tree for simplifications

ing code:

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

where `t` is a fresh temporary variable.

4.3 CSL Left Expansion

This simplification is intended to fulfill the CSL restriction described in *Section 3.3.2* for the left-hand side of assignments. In particular it affects multi-assignment statements. This is because CSL expansion on the left-hand side of assignments can only occur in multi-assignment statements.

The procedure for this simplification is fairly simple. It is described in the pseudocode in *Figure 4.3*.

```
function CSL_Left_simplification(program p)
for each multi-assign stmt s in p
  l = new empty list of statements
  for each expression e in LHS of s in left-right order
   if e is a possibly expanding expression
     replace e with fresh CSL temp t
     add [e]=t; to l
   end
  end
  add all statements in l immediately after s
end
```

Where a possibly expanding expression is defined to be either a cell indexing expression or a structure access.

Figure 4.3 CSL left expansion pseudocode

This transformation is done indiscriminately. With more shape information it would be possible to either avoid such transformations, or reverse them when it is known that no expansion would occur. With such information it could also be possible to replace CSL variables with a fixed number of normal variables. However an analysis to obtain such information is beyond the scope of this thesis.

Figure 4.4 presents some example code before and after simplification. Notice the CSL expression `CSL[t0]`. This syntax represents a CSL variable occurrence. The variable in question has name `t0`.

```
[a,b{:},c] = foo();
```

becomes

```
[a,CSL[t0],c] = foo();
[b{:}] = CSL[t0];
```

Figure 4.4 CSL expansion

This assignment statement will be simplified in the following way. We start by creating a new empty list `l` that will contain new assignment statements. Each expression in the left-hand side of the assignment will be examined. First, a will be seen. This expression can't possibly undergo CSL expansion, so we move to the next expression. Now `b{:}` will be examined. This expression can undergo CSL expansion, so we replace it with

the CSL temporary `CSL[t0]`. We create a new assignment statement to get the value out of `CSL[t0]` and put it in `b{:}`. This assignment will be `[b{:}] = CSL[t0];`. We put this new statement at the end of our list `l`. Now we move to the next expression on the left-hand side of the original assignment, which is `c`. This expression can't undergo CSL expansion, so we skip it. There are no other expressions to examine so now take all the new assignment statements in `l` and insert them immediately after the modified multi-return statement, which gives the result in the example.

4.4 Multi-Assignment Simplification

The multi-assignment simplification enforces the constraints on the left-hand side of multi-assignments described in *Section 3.3.2*. These constraints require that there be only simple variables without repetition on the left-hand side of multi-assignment statements.

This simplification requires that the CSL left expansion simplification, described in *Section 4.3*, be performed. This is because this simplification is written to extract offending left-hand side expressions into simple temporary variables. If an expression can undergo CSL expansion then it cannot be replaced by a non CSL variable. The CSL left expansion simplification ensures that there are no expressions on the left-hand side that can undergo CSL expansion.

Given that we can assume that the CSL expanding expressions have already been removed, the implementation of this simplification is straightforward. The basic procedure is to traverse the left-hand side expressions in left to right order and replace each non simple name expression or repeated names with a temporary variable. Then we assign each temporary to the appropriate removed expression in the same order. The only minor complexity comes from enforcing non-duplicate variables without replacing every variable. Pseudocode for this transformation is shown in *Figure 4.5*.

Once this transformation is done, we can ensure that all multi-assignment statements will be simple to work with. The order of evaluation for each expression will also be made explicit and the actual assignments in the multi-assignment can be interpreted in any order.

To demonstrate this simplification, we give a simple example in *Figure 4.6*. In this example, there are two expressions on the left-hand side which should be removed. The

```
for each multi-assignment statement s
  l = new empty list of statements
  N = new empty set of names
  for each expression e in left-hand side of s from left-right
    if e is name
      if e ∈ N
        replace e with fresh temporary t
        add e=t; to l
      else
        add e to N
      end
    else
      replace e with fresh temporary t
      add e=t; to l
    end
  end
  add statements in l after s
end
```

Figure 4.5 Multi-Assignment simplification pseudocode

first is `b.c` because it isn't a simple `NameExpr`. The second is third expression, `a`. This expression is a simple `NameExpr`, but, since a already appeared in the left-hand side of this assignment, it needs to be extracted. These two expressions are replaced by temporaries `t0` and `t1`, respectively, and two new assignment statements are inserted.

```
[a,b.c,a] = foo();
```

becomes

```
[a,t0,t1] = foo();
b.c = t0;
a = t1;
```

Figure 4.6 Multi-assignment simplification example

4.5 Left-Hand Side simplification

The *McLAST* specification requires more constraints on the left-hand side of assignments than what is enforced by the CSL left and multi-assignment simplifications. In particular, it requires that no expression on the left-hand side can contain arguments that are not either variable names or literals. For instance this would apply to the arguments of an

indexing expression on the left-hand side of an assignment. If we have a simple assignment such as `A(3+4)= 3`, the index value `3+4` is not a simple variable use, so it will be extracted into a temporary. The multi-assignment simplification is a dependency for the left-hand side simplification, and so will already enforce this for multi-assignment statements. So all this transformation needs to simplify is simple assignments. This is a major reason why this dependency exists. The dependency could not be removed because the multi-assignment simplification introduces new simple assignments. If the left-hand side simplification is not guaranteed to be run after the multi-assignment simplification, then there may be expressions on the left-hand sides that don't get simplified. Alternatively, the left-hand side simplification could be made more complex and actually deal with multi-assignment statements. This was deemed undesirable due to that added complexity and because the dependency seems to be a natural one.

The left-hand side simplification could be implemented in a simple way by replacing every argument or subexpression in the left-hand side with temps. This would satisfy *Mc***LAST**'s constraints for the left-hand side of assignments. Unfortunately such a simple transformation would create excessive temporary variables. For instance, if we have the statement `A(i)= b`. The `i` on the left hand side may be a variable, in which case we wouldn't want to extract it. However, the `i` could also be referring to a function, such as the built-in function `i` that returns the unit imaginary number. Such a simple transformation would also have the consequence of causing the simplification to never reach a fixed-point if repeatedly applied. In such a situation it would extract all the temporary variables put in place by the previous application, replacing them with even more temporaries. To avoid these problems the simplification needs to be implemented with more care. To do this we require information about what names refer to variables. Fortunately there is a static analysis that computes such information. This analysis is called the Kind Analysis [DHR11] and is used to estimate the kind of a given identifier. In particular the analysis can estimate if an identifier is a variable or a function. There are situations where it cannot determine this information for some identifiers. When it can't determine the exact kind of an identifier it is simply extracted. The Kind Analysis has the nice property that the way temporary variables are added ensures that the analysis will always recognize them as variables. In addition, each temporary variable is tagged as a temporary in the IR. This is so new temporary

variables can be recognized in the IR without re-analyzing the program. More information about the analysis can be seen in *Section 5.2.5*.

The procedure for performing this simplification would be straightforward except for the presence of **end** expressions. As described in *Section 2.2*, the **end** expression binds to the tightest enclosing array or cell array indexing. So if an **end** is bound to a given indexing expression *e*, and we extract all sub-expressions of *e*, then it would be difficult to find what expression the **end** binds to after the transformation. It would also not be simple to have a separate transformation that removes the **end** expressions, or makes the binding explicit. This is because of side effects. For an expression such as `a(foo()).b(end,42)` where `a` is a variable and `foo` is a function, it would be tempting to write `a(foo()).b(end(a(foo()).b,2,1),42)` where `end(a(foo()).b,2,1)` represents the explicit binding. Performing such a transformation would then cause `foo()` to be evaluated more than once. This is wasteful and incorrect due to side effects.

To have a simplification that does a correct transformation would require duplication of functionality implemented in the left and right-hand side simplifications. Instead, each of these simplifications handle the **end** in their own implementation. The functionality for removing **end** is similar in both left and right, so it was factored out and used by both.

The strategy for performing this simplification is to extract all complex parameters into temporaries. If they are CSL expanding parameters, they are extracted into CSL temporaries. At the same time, the expression being extracted is searched for **end** expressions that bind to the target of the parameters. It is replaced with an explicit **end** and we record that the explicit **end** is associated with the bound target. Once all indices are removed and the expression being indexed is completely simplified, all explicit **end** expressions are made to bind to the simplified target. New assignments are added for the extracted expressions before the statement being simplified. The procedure is demonstrated through an example.

We will start off with an assignment statement that has a fairly complex left-hand side, containing an **end** expression.

```
1 a(foo(),3).b(4,bar(end),v) = 42; %Where v is a variable
```

We will be simplifying the expression on the left-hand side of this assignment. This expression is a `ParameterizedExpr` with the target being `a(foo(),3).b`. Further more,

since this `ParameterizedExpr` is the left-hand side of an assignment, it must be indexing the array given by its target. Since it is a `ParameterizedExpr` its arguments may contain an **end** expression that binds to the array being indexed. For the sake of the argument, we will assume that `bar` is a function, so the `ParameterizedExpr` does indeed contain an **end** expression that binds to it. The binding of this **end** must made explicit, we do this by replacing it with an explicit end. This results in the following partially transformed code.

```
a(foo(),3).b(4,bar(end(?,2,3))) = 42;
```

Notice that the explicit end contains a `?` instead of an expression giving the array it is bound to. This is because we do not yet have the fully simplified expression that gives the desired array. For now, we will have to keep track of the fact that this **end** will bind to the simplified version of `a(foo(),3).b`. Now that we have made all the **end** expressions explicit, we continue by extracting complex sub-expressions from the partially simplified expression.

This statement has two expressions that need to be removed, `foo()` and `bar(end(?,2,3))`. The target of the `ParameterizedExpr` being simplified needs to be computed first, which means `foo()` needs to be computed first. To reflect this, `foo()` needs to be extracted first. It is replaced by a new temporary and a new assignment statement is created for it. This results in the following code.

```
t0 = foo();
a(t0,3).b(4,bar(end(?,2,3)),v) = 42;
```

Next, we will extract `bar(end(?,2,3))` from the indices of `a(t0,3).b`. This results in the following code.

```
t0 = foo();
t1 = bar(end(?,2,3));
a(t0,3).b(4,t1,v) = 42;
```

This leaves the left-hand side of the original assignment statement fully simplified. However, we are still left with the partially transformed explicit **end** expression. At this point, we have the fully simplified expression that the **end** binds to, and we can complete

the transformation. This results in the following fully transformed code.

```
t1 = foo();
t2 = bar(4,end(a(t1,3).b, 2, 3), v);
a(t1,3).b(4,t2,v) = 42;
```

4.6 For Loop Simplification

A `for` loop in MATLAB can be somewhat complex. In particular, what a `for` loop is actually looping over may not be clear. A `for` loop with the form

```
for i = E
  BODY
end
```

will first evaluate the expression `E`. It will then treat the resulting array as two dimensional. The loop will loop through the columns, assigning each column to the loop variable `i`. For this reason, *McLAST* requires that there only be simple range `for` loops. That is to say, `for` loops with a start, stop, and optional step number. The loop variable will loop over the numbers between start and stop, rather than arbitrary values. This restriction still allows us to perform all the same computations, but forces the loop semantics to be explicit.

The goal of the `for` loop simplification is to transform arbitrary `for` loops into simple range `for` loops. At the same time we want to expose the loop semantics of the original loop.

The procedure for this simplification is straightforward. To perform the transformation we find each `for` statement that isn't already a range `for` loop, and modify the loop variable assignment, adding necessary temporaries and assignments to expose the semantics. We will demonstrate the procedure with an example. Each step of the example is justified to demonstrate validity of the transformation.

We start with a simple generic for loop.

```
for i = E
  BODY
end
```

We assume that the expression `E` is not a range expression. We don't wish to evaluate `E` multiple times. To this end, we first extract it into a variable. This results in the following equivalent code.

```
t1=E;
for i = t1
  BODY
end
```

Next, we need to add computations for the limits of the range loop. As we mentioned previously, MATLAB will treat the array being looped over as two dimensional and loop over each column. The transformed range loop will instead loop over the number of columns. The number of columns is simple to compute. For this, we will take advantage of the built-in `size` function. This function is used to return the sizes of the dimensions of a given array. It can be called in a way that expects two return values, the second return value will be the desired number of columns. We add these computation to the **`for`** loop.

```
t1=E;
[t2,t3] = size(t1);
for i = t1
  BODY
end
```

`t2` will contain the size of the first dimension, a value that isn't actually needed. `t3` will be the desired number of columns.

Obviously this is still the equivalent loop, since we have only added a single new side effect-less statement.

We can now modify the loop to be a range loop. As we said, the loop will range over the number of columns. This range will be `1:t3`. We will also need to ensure that the loop variable contains the appropriate value, which will be the appropriate column from the original array.

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

Now there is one extra detail that needs to be covered by this simplification. When the loop domain of a `for` loop is empty, the loop variable will actually be given the value `[]`, which is a 0×0 `double` array. In order to capture this, we add an assignment to `i` with the value `[]` immediately before the loop.

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

This is the final result of our transformation. The loop is now a simple range `for` loop, ranging over the number of columns in the original loop domain. The original loop variable is given the value of the appropriate column at the start of each iteration. This results in a loop that performs the same iterations as the original loop, except the loop values are now made more explicit.

4.7 Simple If Statements

The simple `if` statement transformation is a straightforward simplification. Because *McLAST* does not allow `elseif` statements, they need to be simplified into explicit nested `if` statements. The procedure is straightforward. For every `if` statement containing an `elseif`, we remove the first `elseif`, create a new `if` out of it and put any remaining `elseif` and the `else` from the original `if` into it. We then put this new `if` into a new `else` of the original `if`. Pseudocode for the procedure is given is *Figure 4.7*

```
for each if statement s in original program
  if s has elseifs
    simplify(s)
  end
end
```

```
function simplify( s ):
  ei = the first elseif in s
  ni = a new if statement
  set ni condition to ei condition
  set ni then block to ei then block
  set ni elseifs to s elseifs without ei
  set ni else block to s elseblock if one exists
  remove all elseifs from s
  remove else block from s
  add an else block to s containing only ni
  simplify( ni )
end
```

Figure 4.7 Simplifying `if` statement pseudocode

This simple transformation allows us to assume all `if` statements will only be `if else` statements. This assumption has greatly simplified implementation of other simplifications.

4.8 Array Short-Circuit simplification

In MATLAB there are two forms of short-circuit operations. The first is the standard short-circuit operations `&&` and `||`. These operators work with scalar values of type logical. These will be dealt with in a *Section 4.10*. The second type is the array logical operators `&` and `|`. In normal expressions, these are not short-circuiting. They only become short-circuiting when in the conditional of an `if` or `while` statement. They will also not be short-circuiting if they are nested inside an expression that isn't a `&` or `|` expression. Because these operators are only short-circuiting in special circumstances, we perform the simplification in this transformation, rather than in a general way with the `&&` and `||` operators.

The goal of this simplification is to expose the control flow introduced by these short-circuit operators. We also wish to maintain the correct semantics of these operators. The important semantics to consider are constraints on size and shape of operands. If both

operands are to be evaluated, then both operands must evaluate to an array of the same size and shape.

Another important consideration is that these operators are not recommended for use in short-circuiting situations. They are however acceptable and even desirable in non short-circuiting situations. Because of this recommendation, we have focused on performing an accurate, safe, and simple transformation at some cost to run-time performance.

The transformation has two steps. The first step extracts all array short-circuiting expressions for the conditions of `if` and `while` statements. It replaces them with a temporary variable, assumed to contain the result of the expression. The second step expands the short-circuit control flow and generates the statements to represent it. These statements are generated to ensure that the final result will be assigned to the desired temporary variable. The statements will be inserted before the `if` or `while` they were extracted from. The insertion process in particular is greatly simplified by the simplification of `if` statements. Pseudocode for the process is given in *Figure 4.8*.

```
1   for each if or while statement s
2    c = the condition expression of s
3    if c is a & or | expression
4     t = fresh temporary
5     replace c with t
6     l = new list of statements
7     if c is a & expression
8      simplify(c,&,l,t)
9     else
10     simplify(c,|,l,t)
11    end
12    add statements in l before s
13   end
14  end
```

The `simplify(...)` function is given in *Figure 4.9*.

Figure 4.8 Array short-circuiting simplification pseudocode

We will demonstrate this procedure with an example. The following code contains an

```
function simplify(exp, op, l, t)
 L = left operand of exp
 R = right operand of exp
 t1 = fresh temporary
 if L is a & or | expression
  simplify( L, operator of L, l, t1 )
 else
  add t1 = L; to l
 end
 l1 = new empty list of expressions
 t2 = fresh temporary
 if R is a & or | expression
  simplify( R, operator of R, l1, t2 )
 else
  add t2 = R to l1
 end
 add t = t1 op t2; to l1
 if op is |
  add the following if statement to l,
     replacing l1 with the appropriate value
  'if t1
    t=t1;
   else
    l1
   end'
 else
  add the following if statement to l,
     replacing l1 with the appropriate value
  'if t1
    l1
   else
    t=t1;
   end'
 end
end
```

Figure 4.9 Simplify function pseudocode for array short-circuiting simplification

array short-circuit expression.

```
if A & (B | C)
  foo();
end
```

The implied logic is that, first `A` is evaluated. If that array only contains `true` values, then the short-circuit expression `B | C` is evaluated. The evaluation of this expressions means that first `B` is evaluated. If this array only contains `true` values, then `C` does not need to be evaluated. Otherwise, we must evaluate `C`. If the array that `A` evaluated to didn't contain all `true` values, then we wouldn't have needed to evaluate the rest of the expression. Simplifying all this logic out, using the procedure described in this section, results in the following code.

```
t1 = A;
if t1
  t3 = B;
  if t3
    t2 = t3;
  else
    t4 = C;
    t2 = t3 | t4;
  end
  t0 = t1 & t2;
else
  t0 = t1;
end
if t0
  foo();
end
```

4.9 Conditional Simplification

Even after performing **`if`** statement and array short-circuit simplification, **`if`** and **`while`** statements can still contain complex conditional expressions. According to the IR definition, these conditional expressions should only be literals, variable uses or comparisons of

variable uses or literals.

The transformation to enforce this constraint is a simple one. It uses the Kind Analysis to distinguish names that are definitely variables in order to avoid excessive temporaries. Pseudocode for this transformation is given in *Figure 4.10*. The procedure is to check each conditional expression and extract it if it is not a simple expression.

```
for each if or while statement s
  l = new list of statements
  c = condition expression of s
  if c is not a variable name
    if c is a <, >, <=, >= expression
      if left operand L of c is not a variable
        t = fresh temporary
        replace L with t
        add t = L; to l
      end
      if right operand R of c is not a variable
        t = fresh temporary
        replace R with t
        add t = R; to l
      end
    else
      t = fresh temporary
      replace c with t
      add t = c to l
    end
    add statements in l before s
  end
end
```

Figure 4.10 Condition simplification pseudocode

A simple example of this simplification is given in *Figure 4.11*. In this example, the conditional expression `(foo()+2)>X` is simplified. Note that in the result of this example, the expression `foo()+2` is left unsimplified on the right-hand side of an assignment. This is because simplifying this expression is left to subsequent transformations.

```
if (foo()+2)>X
  bar();
end
```

becomes

```
t0 = foo()+2;
t1 = X;
if t1>t2
  bar();
end
```

Figure 4.11 Conditional simplification example

4.10 Right-Hand Side Simplification

As with left-hand side expressions, the *Mc*LAST specification requires that no expression contain any sub expression that is not either a literal or a variable use. Meeting this requirement is the goal of the right-hand side simplification. All simplifications discussed to this point are dependencies of the right-hand side simplification. Having these dependencies will greatly simplify the right-hand side simplification. This is because all these dependencies have the effect of having all complex expressions exposed as either expressions on the right-hand side of assignments, or as expressions statements. A `for` statement can also contain complex expressions in its loop variable assignment but, since the `for` loop simplification has run, only range expressions need to be considered here. All other expressions can assumed to be simplified already.

The basic procedure is to take right-hand side expressions and expression statements and remove complex sub expressions. These sub expressions are replaced by temporary variable uses and statements to perform the correct computation representing the expression and assigning the value to the temporary are created. The statements are then simplified further.

There are four types of complex expressions that must be removed and dealt with in different ways. The first are basic complex expressions. These expressions are simply uses of operators, and function calls, This also includes array and cell array construction such as `[1 2 3 4]`. Such expressions are simply moved into temporaries and assignments are created from the expression to the temporary.

The second type of complex expressions are CSL expanding expressions as parameters. As in *Section 4.3*, CSL expansion must be treated in a special way. CSL expansion can

occur if an appropriate expression is used as an argument or index. These expressions are be extracted into CSL temporaries.

The third type of expressions considered are `end` expressions. As was described in *Section 4.5*, an `end` expression binds to the tightest array indexing containing them. When simplifying a given expression, if an `end` can bind to it, then it must be searched for `end` expressions. The `end` expressions will need to be dealt with in the same way that they were in *Section 4.5*.

The final type of expressions are short-circuit expressions. In *Section 4.8* some short-circuit expressions were dealt with. These short-circuit expressions were the array short-circuit expressions using the `&` and `|` operators. These operators will only be short-circuiting when used inside the condition of an `if` or `while`. Since they have already been simplified, they do not need to be dealt with here. There are however still the scalar short-circuit operators `&&` and `||`. These operators will be short-circuiting anywhere, so they must be simplified. We describe the procedure for simplifying these expressions in more detail in *Section 4.10.1*. For now, it is sufficient to explain that simplifying these expressions will expose the control flow caused by the short-circuit behaviour. This will cause new `if` statements to be created.

The procedure is to simplify each needed statement, then recursively simplify each statement generated by simplifying that statement. This will generate a list of simplified statements, the last of which is the simplified original statement. This list of statements is then inserted in place of the original statement. Pseudocode for this procedure is given in *Figure 4.12*.

We give an example to demonstrate the recursive procedure for simplifying a right-hand side expression. This example is given in *Figure 4.13*

4.10.1 Short-Circuit Expression simplification

Scalar short-circuit expressions need to be expanded to make explicit the control flow they represent. These expressions can appear in any other expression, and can contain arbitrary sub-expressions. Because of this, simplifying short-circuit expressions must be done at the same time as other expressions are simplified. It would be preferable to have this

```
function rightSimplification(program)
  for each statement s in program
    l = new list of statements
    simplifyStmt( s, l )
    replace s with statements in l
  end
end
```

```
function simplifyStmt( s, l )
  if s is assignment
    e = right-hand side of s
    simplifyExpr( e, l )
  else if s is expression statement
    e = expression in s
    simplifyExpr( e, l )
  else if s is for statement
    e = right-hand side of assignment in s
    simplifyExpr( e, l )
  end
  append simplified s to l
end
```

For the sake of clarity, `simplifyExpr(...)` is describe through an example in *Figure 4.13*. This example illustrates one iteration of the recursive process, which results in new assignment statements. These new statements will be simplified further in subsequent iterations.

Figure 4.12 Right-hand side simplification pseudocode

transformation separate from the right-hand side simplification, but this would require a cyclic dependency between the short-circuit expressions simplification and the right-hand side simplification.

There are important issues to consider when expanding short-circuit expressions. The expansion should avoid causing duplicate code, the transformed code should not be significantly larger than the original code. Creation of new temporaries should be reduced. The association between the terms in a short-circuit expressions should be preserved as much as possible. And of course MATLAB semantics should be preserved.

These concerns sometimes compete with each other. For instance, the goal of maintaining the association between terms is to make it obvious what terms have already been evaluated and what their truth values are when evaluating a specific term. To this end, it would be preferable to transform `t = (A && B)|| (C && D)` into the code shown in *Fig-*

```
%assume A is a variable and foo and bar are functions
X = A(foo(1+2),bar(end));
```

The statement in this example is an assignment statement, which means that the right-hand side expression will be simplified by `simplifyExpr(...)`. The rest of this example demonstrates this procedure.

Note the use of the **`end`** expression as an argument to `bar(...)`. This **`end`** expression is bound to `A` and will need to be made explicit in this pass of the simplification. We will also be removing the expressions `foo(1+2)` and `bar(end)`, resulting in new assignment statements. This first iteration of the recursive procedure results in the following code:

```
t0 = foo(1+2);
t1 = bar(end(A,1,1));
X = A(t0,t1);
```

Note that this result contains two new assignment statements that require further simplification on their right-hand side. These are simplified further by calling the `simplifyStmt(...)` procedure, given in *Figure 4.12*, on each of them recursively.

Figure 4.13 Right-hand side simplification example

ure 4.14. Lines 5-9 and 12-16 are duplicates of each other. As the expression gets larger, more duplication will occur. This conflicts with reducing code expansion.

Avoiding code duplication is an important concern. The running time of code analysis will grow with respect to code size. Duplicating function calls can create difficulty for a context sensitive interprocedural analysis. Because of this, code duplication is avoided in this transformation. There is however, still an attempt to maintain the association between some terms in the expression.

The procedure for simplifying can be explained using some transformation patterns. This procedure would be a special case of the `simplifyExpr` function. Rather than performing the simplification described in *Figure 4.13*, `simplifyExpr` would detect that a short-circuiting expression is being simplified and instead use this procedure. These patterns are presented next.

Some notes need to be made concerning the simplification and notation used. First,

```
1   if A
2    if B
3     t = true;
4    else
5     if C
6      t = D;
7     else
8      t = false;
9     end
10   end
11  else
12   if C
13    t = D;
14   else
15    t = false;
16   end
17  end
```

Figure 4.14 Naive short-circuit expansion

due to the conditional simplification in *Section 4.9* and the procedure of the right-hand side simplification, we can assume that all short-circuit expressions are in statements of the form `t=EXP`. Where `t` is some temporary. In the patterns, the notation $[\texttt{t,EXP}]_{SC}$ and `[t,EXP]` are used. These represent the short-circuit and non short-circuit simplification, respectively, of the given expression with the resulting computed value stored in temporary `t`.

We start the procedure off by simplifying $[\texttt{t,EXP}]_{SC}$, which will give us the simplified version of `t=EXP`. Patterns for performing these simplifications are given in *Figure 4.15*. These patterns match based on the expression in the $[\texttt{t,EXP}]_{SC}$ notation. They will either produce an **`if`** statement requiring further short-circuit simplification, a new expression requiring short-circuit simplification, or the same expression requiring non short-circuit simplification. It is important to note that all **`if`** statements produced by this simplification will have a very particular pattern. They will only contain simple assignments of boolean literals to a temporary variable. These **`if`** statements are further dealt with by patterns in *Figure 4.16*. When a pattern switches from short-circuit simplification to non short-circuit simplification, the transformation also inserts a `CheckScalarStmt` statement for the temporary holding the result of the expression. This new statement enforces that the

result computed by the transformed code must be a scalar value. Adding this statement is necessary because the `if` statements produced will not enforce this constraint. The exact behaviour of this statement is left undefined. In a virtual machine, this statement could be executed as a run-time check of the given variable. In a more static setting, such as *Mc***FOR**, a program may be statically rejected if it cannot guarantee that all such checks pass. In this case, the `CheckScalarStmt` has no run-time behaviour.

Since these patterns are recursive, we briefly discuss termination of the procedure. These patterns terminate when pattern 6 is reached. This occurs when an expression does not have a `&&`, `||`, or `~` as its main operator.

Consider the simplification of $\texttt{[t,EXP]}_{SC}$. Take the three-tuple $\langle \#\texttt{||}, \#\texttt{\&\&}, \#\texttt{\~{}} \rangle_{\texttt{EXP}}$ of the counts of `||`, `&&`, and `~` in the expression `EXP`. Each pattern will create new expressions that require simplification, e.g. $\texttt{[t,EXP}_2\texttt{]}_{SC}$. The three-tuple $\langle \#\texttt{||}, \#\texttt{\&\&}, \#\texttt{\~{}} \rangle_{\texttt{EXP}_2}$ corresponding to the new expression will be lexicographically less than the original three-tuple. This will cause the simplification to always reach pattern 6.

The patterns from *Figure 4.15* can generate `if` statements with conditional expressions that require simplifications. *Figure 4.16* presents patterns to simplify these `if` statements. These patterns only consider the conditional expression of the `if` statement. They will either produce a new `if` statement requiring further simplification, or pull out the expression for simplifying by the previous patterns.

An important note is that pattern 1 appears to introduce duplicated code. The `<ELSE PART>` appears twice in the result of the pattern. This is indeed a duplication of code but it is tightly controlled. These `if` statements are only produced by the previous patterns. The `<ELSE PART>` of those `if` statements will only contain a single assignment of a boolean literal to a temporary variable. These patterns for simplifying the `if` statements will not produce an `<ELSE PART>` more complex than what it is given, so it preserves this property. It guarantees that only a single simple assignment statement is duplicated. This is an acceptable duplication.

These patterns will terminate when pattern 3 is reached. This pattern will always be reached for the same reason that the previous patterns terminate. When pattern 3 is reached, simplification will continue using the previous patterns. Since those patterns terminate, the overall procedure will terminate.

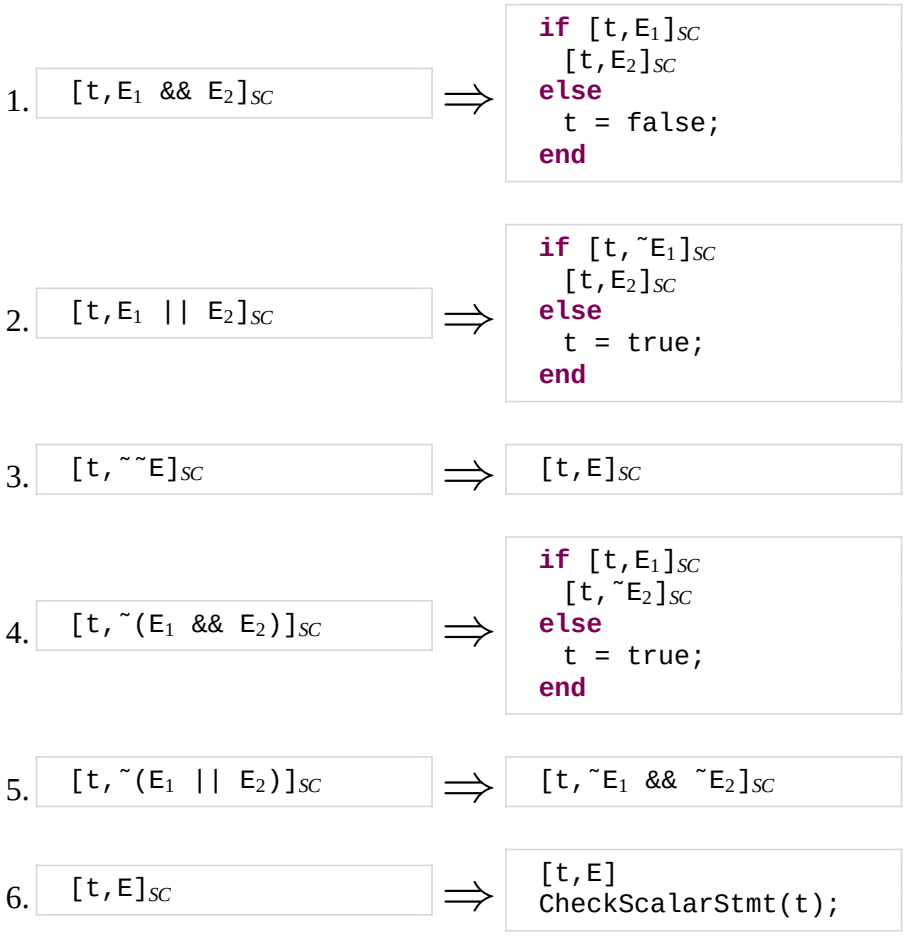


Figure 4.15 Short-circuit patterns for assignments

To demonstrate this procedure, we will use these patterns to simplify the expression `(A && B)&& C`. The process will start off as the following:

```
1  [t,(A && B) && C]SC
```

The main operator here is the an `&&`. This matches with pattern 1 from *Figure 4.15*.

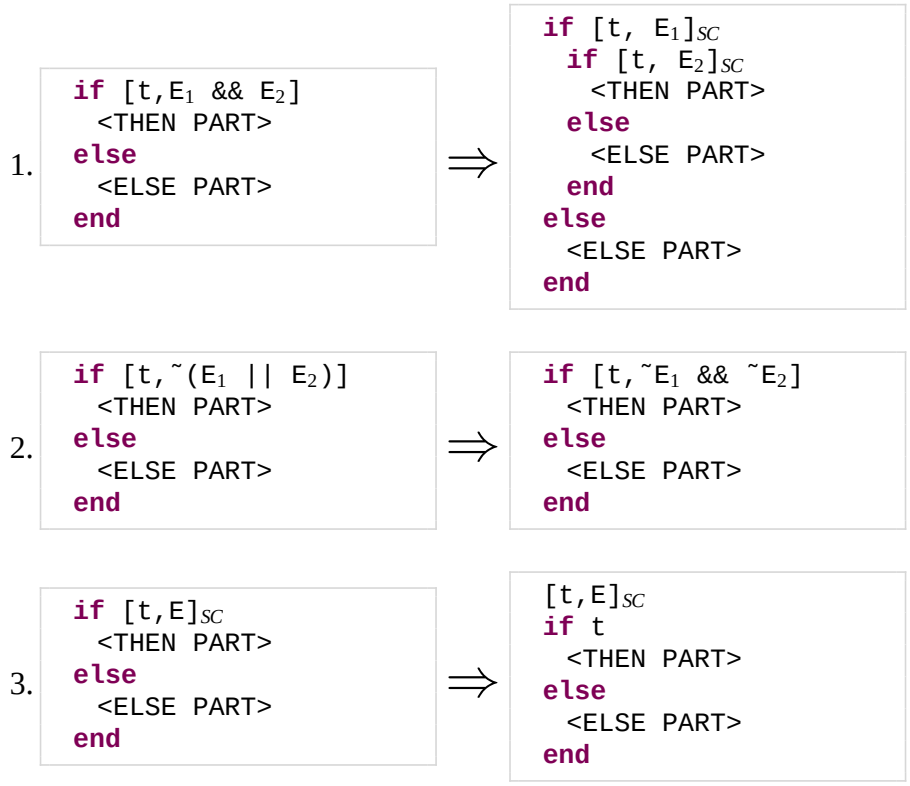


Figure 4.16 Short-circuit patterns for `if` statements

This results in the following:

```
1  if [t,A && B]SC
2    [t,C]SC
3  else
4    t = false;
5  end
```

Here we will be working with the patterns for `if` statements, given in *Figure 4.16*. Again, the main operator is a `&&`. This means we will be using pattern 1 in 4.16, resulting in the

following:

```
if [t,A]SC
  if [t,B]SC
    [t,C]SC
  else
    t = false;
  end
else
  t = false;
end
```

At this stage, all we have are simple name expressions left to simplify. Rather than go through all the steps needed to completely simplify, we will apply all the remaining patterns at once. The patterns that apply are pattern 3 from 4.16 and 6 from 4.15. We also perform the final step of simplifying, which is to simplify `[t,A]`, `[t,B]`, and `[t,C]` into `t=A;`, `t=B;`, and `t=C;`.

This gives us the following code.

```
t = A;
CheckScalarStmt(t);
if t
  t = B;
  CheckScalarStmt(t);
  if t
    t = C;
    CheckScalarStmt(t);
  else
    t = false;
  end
else
  t = false;
end
```

4.11 Full Simplification

The final simplification is the full simplification. This is actually just a dummy simplification in that it doesn't perform any transformation. The purpose for it, is to have one simplification that is guaranteed to execute all other simplifications. It does this by being a dependent of all simplifications that don't otherwise have dependents. In the current implementation the only simplification that falls into that category is the right-hand side simplification. If other such simplifications were added, then they would be listed in full simplification's dependencies. As it stands, the full simplification acts as a more appropriate starting point than the right-hand side simplification for fully simplifying the AST.

Chapter 5
Intraprocedural Analysis Framework

In order for a compiler to perform optimizations or provide feedback to programmers, it must have a way to automatically reason about the programs it is given. This is done by creating static analyses that are used to infer information about a given program. These analyses usually fit into one of a few categories. They can be flow-insensitive or flow-sensitive. If they are flow-sensitive then they can be forwards or backwards.There are also intraprocedural and interprocedural flavours of these. This thesis focuses on the intraprocedural flavour of these analyses.

Different analyses will often require the same functionality. For instance, all analyses require a way of visiting nodes in the IR. This is specialized for flow-insensitive and flow-sensitive and further specialized for forward and backward flow-sensitive analyses. This kind of basic functionality should not be rewritten for each analysis. Instead, as a contribution of this thesis, a framework for developing static analyses has been developed.

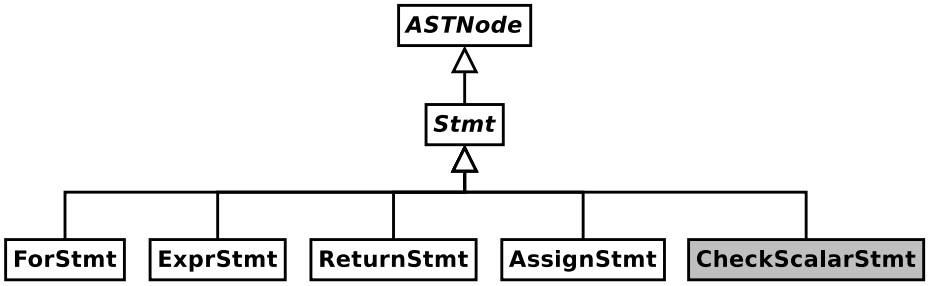
This static analysis framework is intended to make it simpler for programmers to develop new analyses. It does this by defining the different types of analyses and providing an implementation of basic procedures. Another goal is to make it extensible to new language extensions. This goal has two requirements, the framework itself should be usable to create new analyses for a language extension, and old analyses should be adaptable to new extensions.

In this chapter we will be presenting the design of the analysis framework. *Section 5.1* describes the basic traversal mechanism used by all analyses. This is followed by a

description of the different types of analyses in *Section 5.2*. In *Section 5.2.5*, some analysis created for *McLAB* using this analysis framework are described. Finally, in *Chapter 6* we describe how the framework and existing analyses can be extended to new language extensions.

5.1 Basic Traversal Mechanism

The analysis framework is designed to work with the IR; it can be used with both *McAST* and *McLAST*. The analyses created using the framework are going to need a way of traversing the IR. This traversal can take different forms; it can be a simple traversal of the tree structure, such as a depth-first traversal, or it can be a traversal where some nodes are visited repeatedly, for instance to compute a fixed-point for a flow-sensitive analysis.



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*.

Figure 5.1 Excerpt of AST class hierarchy

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, an excerpt of which is depicted in *Figure 5.1*. To facilitate traversal, we have created a Java interface called `NodeCaseHandler` that consists of methods of the form `void caseStmt( Stmt node )`. There is one such method for every AST class. We have also provided 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 an excerpt from `AbstractNodeCaseHandler` for the `AssignStmt` and `Stmt` classes. This excerpt is given in *Figure 5.2*. Notice on line 3, the `caseAssignStmt(...)` is forwarding up to the case belonging to its parent class, `AssignStmt`.

```
1 public void caseAssignStmt( AssignStmt node )
2 {
3   caseStmt( node );
4 }
5 public void caseStmt( Stmt node )
6 {
7   caseASTNode( node );
8 }
```

Figure 5.2 Excerpt of `AbstractNodeCaseHandler` demonstrating default behaviour

Figure 5.1 shows that the `AssignStmt` node type extends the `Stmt` node type. As described previously, this means the default behaviour for `caseAssignStmt(...)` is to forward to `caseStmt(...)`, which is done by passing the argument from `caseAssignStmt(...)` to `caseStmt(...)`. The definition for the `caseAssignStmt(...)` method demonstrates the forwarding behaviour. This method takes in an instance of `AssignStmt` and calls `caseStmt(...)` with that instance.

It should be noted 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 `caseStmt(...)` will forward to `caseASTNode(...)`. The `AbstractNodeCaseHandler` leaves the `caseASTNode(...)` 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 directly create a 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 5.3*.

```
1  public class StmtCounter extends AbstractNodeCaseHandler
2  {
3    private int count = 0;
4    private StmtCounter()
5    {
6      super();
7    }
8
9    public static int countStmts( ASTNode tree )
10   {
11     tree.analyze( new StmtCounter() );
12   }
13
14   public void caseASTNode( ASTNode node )
15   {
16     for( int i = 0; i<node.getNumChild(); i++ )
17       node.getChild(i).analyze( this );
18   }
19
20   public void caseStmt( Stmt node )
21   {
22     count++;
23     caseASTNode( node );
24   }
25 }
```

Figure 5.3 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 `caseASTNode(...)` 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 `caseASTNode(...)` with other behaviour implemented for control structures like loops and conditionals.

The second thing to notice is the `caseStmt(...)` method. Besides the `caseASTNode(...)`, 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. One should also notice that this implementation of `caseStmt(...)` forwards to `caseASTNode(...)`. 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 `caseASTNode(...)`.

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 or 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.

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

This method will prevent all children of any expression from being visited.

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. This mechanism is also used by the simplifications presented in *Chapter 4*. 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 its own

behaviour for the appropriate node cases.

5.2 Analysis Types

We have seen how the basic traversal mechanism is implemented and used. The basic API is defined by the `NodeCaseHandler` interface, and default behaviour is implemented by `AbstractNodeCaseHandler`. New traversals can be created by extending `AbstractNodeCaseHandler`, providing an implementation for `caseASTNode(...)` and overriding any needed methods. In the analysis framework, an analysis is effectively an implementation of a traversal. As was mentioned earlier, different analyses tend to require similar behaviour. The framework provides a standard API for analyses and implementations of basic procedures. These implementation details are split over the different types of analyses supported by the framework. The types of analyses currently supported are depth-first traversal, and forwards and backwards structural flow-sensitive analyses.

One element that is common to all analyses is that they produce some form of data. This data needs a way of being represented in analyses. The data will also be manipulated by common procedures implemented in the framework. This means the data should have a common interface, and some useful implementations.

This section continues by first describing, in *Section 5.2.1*, the definition and implementation of different analysis data representations. *Section 5.2.2* continues with the implementation details common to all types of analysis. In *Section 5.2.3* we describe those details unique to dept-first analyses. This includes a simple example analysis to demonstrate the process of creating a new depth-first analysis. *Section 5.2.4* describes details unique to structural analyses. A simple forward structural analysis is presented to demonstrate creating a new structural analysis.

5.2.1 Flow-Data Representation

An analysis is written to produce information about the program being analyzed. As we will see in later sections, the 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 framework defines interfaces and implementations for basic flow-data. A class hierarchy for flow-data structures provided by the framework is given in *Figure 5.4*.

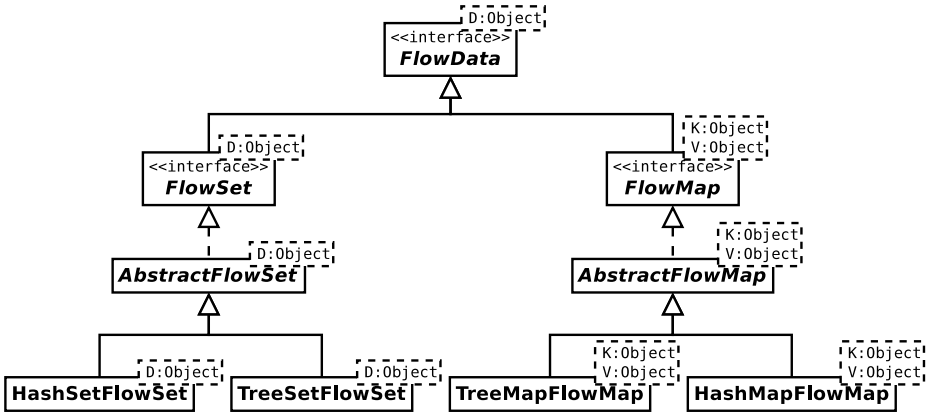


Figure 5.4 Flow-data class hierarchy

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

The set interface, `FlowSet<D>` defines a few additional methods related to manipulating data within the set. A list of these methods is provided in *Table 5.1*.

The abstract implementation of `FlowSet<D>` provides some inefficient implementations of some the methods defined by `FlowData` and `FlowSet`, but most methods are left abstract. This is because methods like `add` and `isEmpty` are too implementation specific to be imple-

`FlowSet<D>`	`copy()`	returns a copy of the data
`void`	`copy(FlowSet<? super D> dest)`	copies the flow-data into a destination flow-data
`void`	`clear()`	clears the collection of data
`boolean`	`isEmpty()`	checks if there is no data
`int`	`size()`	returns the size of the collection
`void`	`add(D obj)`	adds the given object
`boolean`	`remove(Object obj)`	removes the given object
`boolean`	`contains(Object obj)`	checks if the object is in the set
`Iterator<D>`	`iterator()`	returns an iterator over the elements of the set

Table 5.1 Methods in the `FlowSet<D>` interface

mented in the abstract version. `AbstractFlowSet` does however provide some other useful methods. As we will see in *Section 5.2.4*, when doing flow-sensitive analysis, the analysis needs to define a way of merging two flow-data. When dealing with sets, merging can often take the form of a union or intersection. The `AbstractFlowSet` provides implementations of union, intersection, and set difference to make implementing a merge simpler. A list of these methods can be seen in *Table 5.2*. Each of these operations has two versions. The first version operates on the given set and `other` and puts the result in the given set. The second version operates on the given set and `other` and puts the result in `dest`. This behaves as though `dest` is cleared before the result is put into it.

`void`	`union(FlowSet<? extends D> o)`
`void`	`union(FlowSet<? extends D> o,FlowSet<? super D> dest)`
`void`	`intersection(FlowSet<? extends D> o)`
`void`	`intersection(FlowSet<? extends D> o,FlowSet<? super D> dest)`
`void`	`difference(FlowSet<? extends D> o)`
`void`	`difference(FlowSet<? extends D> o,FlowSet<? super D> dest)`

Table 5.2 New methods in the `AbstractFlowSet<D>` interface

Finally there are two concrete implementations of `FlowSet`, `HashSetFlowSet<D>` and `TreeSetFlowSet<D>`. These implementations are backed by a `HashSet` and `TreeSet` re-

spectively. They implement their functionality mainly by relying on the functionality of the set that backs it.

The framework also provides map versions of flow-data. The map interface and its associated implementations are similar to the set ones. There are some differences though. `FlowMap<K,V>` represents flow-data that is a map from some type, `K`, of keys to some type, `V`, of values. In addition, a flow map should be considered a set with respect to its keys.

`FlowMap<K,V>` defines some additional methods. These methods are similar to the set methods, but are geared towards maps. A list of these methods is given in *Table 5.3*. One difference to note is the lack of an iterator method. There is however the `keySet` method that provides a set view of the map, containing the keys of the map. This set can be iterated over.

`FlowMap<D>`	`copy()`	returns a copy of the data
`void`	`copy(FlowMap<K,V> dest)`	copies the flow-data into a destination flow-data
`void`	`clear()`	clears the collection of data
`boolean`	`isEmpty()`	checks if there is no data
`int`	`size()`	returns the size of the collection
`void`	`put( K key, V value )`	associates the value to the key in the map
`boolean`	`remove( Object key )`	removes the entry for the given key
`boolean`	`removeKeys( Collection<?> keys)`	removes all keys from given collection
`boolean`	`containsKey( Object key )`	checks if the map contains an entry for the given key
`V`	`get( Object key )`	gets the value associated with the given key, if it exists
`Set<K>`	`keySet()`	gets a set view of the map's keys

Table 5.3 Methods in the `FlowMap<K,V>` interface

The abstract implementation of the flow map is very similar to the abstract implementation of the flow set. The biggest difference is in how the union and intersection operations

are implemented. Since the flow map is only considered a set in terms of its keys, there needs to be a way of performing these operations on two maps that share a key but differ in its value. To define this behaviour the framework relies on two interfaces, `Mergable<E>` and `Merger<E>`. A brief summary of these interfaces is provided is *Table 5.4*. The `Mergable<E>` interface is implemented by objects that can be merged with something of type `E`. They provide a merge method that returns the merged value. The `Merger<E>` interface represents an object that can merge two objects of type `E`.

The `Mergable<E>` Interface

`E`	`merge(E o)`	merges `o` and the given value

The `Merger<E>` Interface

`E`	`merge(E o1, E o2)`	merges `o1` and `o2`

Table 5.4 Merging interfaces

When performing a union or intersection operation, either a `Merger` must be available, or the values in the maps must implement `Mergable<V>`. A `Merger` can be available either by providing one to a constructor of a concrete map, or by specifying one to a variant of the operation's method. A list of the operation's methods is provided in *Table 5.5*

`void`	`union(FlowMap<K,V> other)`
`void`	`union(Merger<V> m, FlowMap<K,V> other)`
`void`	`union(FlowMap<K,V> other, FlowMap<K,V> dest)`
`void`	`union(Merger<V> m, FlowMap<K,V> other, FlowMap<K,V> dest)`
`void`	`intersection(FlowMap<K,V> other)`
`void`	`intersection(Merger<V> m, FlowMap<K,V> other)`
`void`	`intersection(FlowMap<K,V> other, FlowMap<K,V> dest)`
`void`	`intersection(Merger<V> m, FlowMap<K,V> other, FlowMap<K,V> dest)`
`void`	`difference(FlowMap<K,V> other)`
`void`	`difference(FlowMap<K,V> other, FlowMap<K,V> dest)`

Table 5.5 Operation methods in the `AbstractFlowMap<D>` interface

The framework also provides concrete implementations for flow map. These implementations are the `HashMapFlowMap<K,V>` and `TreeMapFlowMap<K,V>`. They are backed

by a `HashMap` and `TreeMap` respectively, and implement their functionality by relying on the set that backs them.

These data-flow representations are used in examples to describe the implementation of new analyses in *Section 5.2.3* and *Section 5.2.4*. More complex representations are used in the examples in *Section 5.2.5*.

5.2.2 Common Implementation

An analysis is implemented by visiting the nodes of an AST and performing some actions on particular node types. The basic traversal mechanism provides a basis for doing this. However, when performing an analysis, it would be useful to have more information about the node being visited. Information such as knowing when a given expression is the condition for a **`while`** or an **`if`**. This information is not available from the type of the node. We want our analyses to have this extra detail available. We also want our analyses to have a common API. To these ends, we define a new interface, `Analysis`, which provides more detailed node case methods and a common API. A summary of the added API methods and traversal methods is given in *Table 5.6*.

API Methods		
`void`	`analyze()`	performs the analysis
`ASTNode`	`getTree()`	gets the tree being analyzed
`boolean`	`isAnalyzed()`	checks if the analysis has been executed or not
`void`	`setCallBack(NodeCaseHandler h)`	sets the `NodeCaseHandler` used as a callback when visiting a node

Traversal Methods	
`void`	`caseCondition( Expr condExpr )`
`void`	`caseWhileCondition( Expr condExpr )`
`void`	`caseIfCondition( Expr condExpr )`
`void`	`caseLoopVar( AssignStmt loopVar )`
`void`	`caseSwitchExpr( Expr switchExpr )`

Table 5.6 Methods in the `Analysis` interface

One detail to note is the `setCallBack(NodeCaseHandler handler)` method. This

method sets the `NodeCaseHandler` to pass to an AST node's `analyze(...)` method. For the most part it is sufficient to ignore this functionality, or treat it as though the callback is always the analysis itself. This functionality is provided to allow more sophisticated behaviour. For instance, the structural analyses define helpers that implement the `NodeCaseHandler` interface. These helpers are set as the callback, and are used to maintain invariants pertaining to flow-data. Each case method in the helper performs some book keeping, then forwards to the same case of the analysis. Generally it's enough to know that those invariants are maintained, and not care about how, though in advanced cases, such features can be useful. It's also important to be aware of these details when extending the framework. For more information on extensibility, see *Chapter 6*.

5.2.3 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 5.5*.

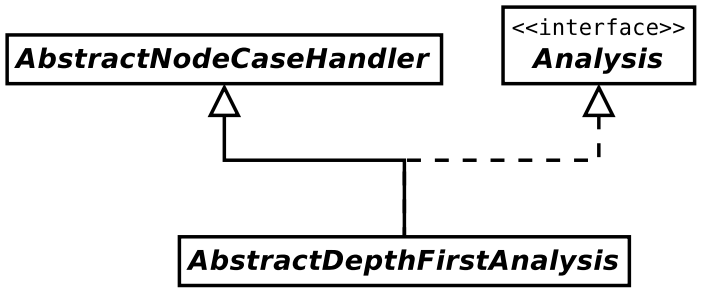


Figure 5.5 Class hierarchy snippet for depth-first analysis

It should be noted that `AbstractDepthFirstAnalysis<A>` is generic with the type variable `A`. This is important because it also provides a new method, `newInitialFlow()`. The `newInitialFlow()` method returns a value of type `A`, representing an initial flow

value. This type variable represents the type of data being computed by the analysis. It is an unbounded type variable, but in practise, it will usually be a subclass of `FlowData`. The reason it is kept unbounded is to increase flexibility for the framework.

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 5.6* 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.

```
1 public void caseASTNode(ASTNode node)
2 {
3    //visit each child node in forward order
4    for( int i = 0; i<node.getNumChild(); i++ ){
5      if( node.getChild(i) != null )
6      node.getChild(i).analyze( callback );
7    }
8 }
```

Figure 5.6 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

- select an analysis data type
- implement an appropriate `newInitialFlow` method
- 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 associate flow-data with particular nodes in the tree.

To demonstrate the process of implementing a depth-first analysis, we will present a simple example analysis. This analysis is intended to collect all names that are assigned to. It will perform two tasks. First, for each assignment statement in the tree, it will associate all names that are assigned to by that assignment statement to the assignment statement. Second, it will collect in one set, all names that are assigned to in the entire AST.

We will be storing a name as a `String`. Since we will be dealing with sets of strings, we can use the predefined `HashSetFlowSet` for our flow-data. In particular, we can make our flow-data be of type `HashSetFlowSet<String>`, a flow-set containing strings. Recall that this class was defined in *Section 5.2.1*.

This lets us create the shell of our analysis, which will be called `NameCollector`. The code so far is given in *Figure 5.7*.

This shell contains a `HashSetFlowSet` field and two accessor methods. One of these methods is for accessing a set of all names, the other is for accessing the set of names for a given assignment. If the assignment has no names associated with it, it returns `null`. This implementation assumes that the `flowSet`'s map is being used to associate the set to each assignment statement.

The analysis, as it stands now, will only traverse the AST. It won't actually collect any names. To add the name collecting behaviour, we need to override specific case methods.

```
public class NameCollector
   extends AbstractDepthFirstAnalysis<HashSetFlowSet<String>>
{
  private HashSetFlowSet<String> fullSet;
  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();
  }
}
```

Figure 5.7 Shell of example depth-first analysis `NameCollector`

We will implement our desired behaviour as follows:

```
for each AssignStmt s in the AST
  currentSet = new set
  collect all names in LHS being assigned to into currentSet
  flowSets[s] = currentSet
  add currentSet to fullSet
```

In order to do this, we start by implementing a `caseAssignStmt(...)` and a `caseName(...)`. The `caseAssignStmt(...)` will be used to setup `currentSet` as well as set a flag that keeps track of whether or not the analysis is in the left-hand side of an assignment. The assignment case will then traverse into its left-hand side, ignoring the right-hand side. Finally, it will map the result to the current node and add it to the total result. The `case-`

`Name(...)` method will be used to add to the current flow-set. It will only do this when it is in the left-hand side of an assignment. This results in the code in *Figure 5.8* being added to the `NameCollector` class.

```
1  private boolean inLHS = false;
2  public void caseName( Name node )
3  {
4    if( inLHS )
5      currentSet.add( node.getID() );
6  }
7
8  public void caseAssignStmt( AssignStmt node )
9  {
10   inLHS = true;
11   currentSet = newInitialFlow();
12   analyze( node.getLHS() );
13   flowSets.put(node,currentSet);
14   fullSet.addAll( currentSet );
15   inLHS = false;
16 }
```

Figure 5.8 `caseAssignStmt(...)` and `caseName(...)` for `NameCollector`

This implementation is a good first attempt. It will capture simple assignments, and it will even capture multi-assignment statements. There is however a problem with it. It ignores the fact that names can appear on the left-hand side of an assignment, but not be a name being assigned to. For example, the following line of code will produce both `"a"` and `"b"` for names being assigned to.

```
a(b) = 42;
```

This is incorrect since `b` is not being assigned to. This can happen when the name is used as an argument, for example, as an index to an array, or cell-array. For brevity we will focus on simple array indexing. To solve this problem, we need to avoid looking at names occurring in the arguments of an indexing expression. In the AST, indexing expressions are represented by `ParameterizedExpr`. To avoid going into the arguments of an indexing expression, we need to write a `caseParameterizedExpr(...)`. The implementation will simply ignore the arguments and traverse into the target expression of the indexing. The method is presented in *Figure 5.9*, and will be added to the class. This method simply

passes the target of the parametrization for analysis, ignoring the arguments.

```
1 public void caseParameterizedExpr( ParameterizedExpr node )
2 {
3   analyze(node.getTarget());
4 }
```

Figure 5.9 `caseParameterizedExpr(...)` for `NameCollector`

A similar `caseCellIndexExpr(...)` and `DotExpr` method would also be needed to avoid the other incorrect name inclusions.

The full code for the class definition of `NameCollector` can be seen in *Figure 5.10*.

5.2.4 Structural Analysis

Structural flow analysis is the core part of the analysis framework. This type of analysis can be used to compute complex information to approximate run-time behaviour of a given program. The implementation of these analyses is necessarily more complex. They require deeper knowledge of the languages semantics, as well as the ability to approximate run-time behaviour of the program. These issues come into play when dealing with control structures in the language, structures such as **`if`**, **`while`**, and **`for`** statements. The framework provides generic implementations of the procedures necessary for implementing a structural analysis.

Structural analysis requires a more complex API and more detailed information about nodes. To capture this, we define an interface for structural analysis. This interface is called `StructuralAnalysis<A>`, and it extends the `Analysis` interface. It provides an API for accessing analysis results, accessing current flow-data, and manipulating flow-data. It also provides some additional cases for finer grained treatment of nodes. These new methods are summarized in *Table 5.7*.

The framework also provides an abstract implementation for structural analyses. This abstract implementation, called `AbstractStructuralAnalysis` provides constructors and implementations for most of the API methods. This implementation is similar to the `AbstractDepthFirstAnalysis` implementation. It also provides a protected method `void analyze( ASTNode node )`. This method is intended to abstract away from the basic

```
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();
    analyze( node.getLHS() );
    flowSets.put(node,currentSet);
    fullSet.addAll( currentSet );
    inLHS = false;
  }
  public void caseParameterizedExpr( ParameterizedExpr node )
  {
    analyze(node.getTarget());
  }
}
```

Figure 5.10 Full `NameCollector` definition

API Methods		
`Map<ASTNode,A>`	`getOutFlowSets()`	gets out flow-data
`Map<ASTNode,A>`	`getInFlowSets()`	gets in flow-data
`void`	`merge(A in1, A in2, A out)`	merges two flow-data putting the result into a third
`void`	`copy(A source, A dest)`	copies the flow source flow-data into dest
`A`	`getCurrentOutSet()`	gets the current out data being worked on
`void`	`setCurrentOutSet(A outSet)`	sets the current out data
`A`	`getCurrentInSet()`	gets the current in data
`void`	`setCurrentInSet(A inSet)`	sets the current in data
`A`	`newInitialFlow()`	gives an initial approximation for data

Traversal Methods	
`void`	`caseLoopVarAsInit( AssignStmt loopVar )`
`void`	`caseLoopVarAsUpdate( AssignStmt loopVar )`
`void`	`caseLoopVarAsCondition( AssignStmt loopVar )`

Table 5.7 Methods in the `StructuralAnalysis` interface

traversal mechanism. The forward and backwards implementations also use it to ensure that the flow-data is setup appropriately. When an analysis needs to analyze a particular node, this method should be used rather than relying on the basic traversal mechanism. There are, however, times when an analysis needs to call a case method directly on a node. In this situation, it is the programmers responsibility to ensure that the flow-data is setup appropriately. This is to ensure that case methods can be implemented assuming that they have appropriate flow-data.

As with `AbstractDepthFirstAnalysis`, this implementation doesn't provide a concrete `newInitialFlow` method. It also doesn't provide a `merge`, or `copy` method. This is because these details are specific to individual analyses. The `merge` method should be implemented to reflect the abstraction being computed by the analysis. The `copy` method should take into account if the copy should be deep or shallow. These are implementation details that require a deeper knowledge of the analysis being written.

`AbstractStructuralAnalysis` also defines a number of protected data members for use by the different implementations. These data members are summarized in *Table 5.8*. These members have to do with the data being computed by the given analysis. A standard analysis will fill the `outFlowSets` and `inFlowSets` with appropriate data for each AST node that has data defined for it. It will compute the values for out or in flow-data for a given node on the in or out flow-data, respectively, of the given node. The way this dependency is defined depends on what flavor of flow analysis is being defined. Further details on how these data members are used is presented later in this section.

`A`	`currentOutSet`	the out data for the node currently being worked on
`A`	`currentInSet`	the in data for the node currently being worked on
`Map<ASTNode,A>`	`outFlowSets`	a map from nodes to their associated out data
`Map<ASTNode,A>`	`inFlowSets`	a map from nodes to their associated in data
`ASTNode`	`tree`	the AST being operated on

Table 5.8 Data members in defined by `AbstractStructuralAnalysis`

Unlike `AbstractDepthFirstAnalysis`, `AbstractStructuralAnalysis` does not provide an implementation for `caseASTNode(...)`. This is because structural analyses are split into two flavours, forwards and backwards. Each of these flavours will require its own implementation of `caseASTNode(...)`. The forwards and backwards analyses are implemented in a similar way. For each, we define 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, flow-analyses. In the case of the loop cases, `caseForStmt(...)` and `caseWhileStmt(...)`, they provide the fixed-point computation procedure. 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 programmers should not have to override them.

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.

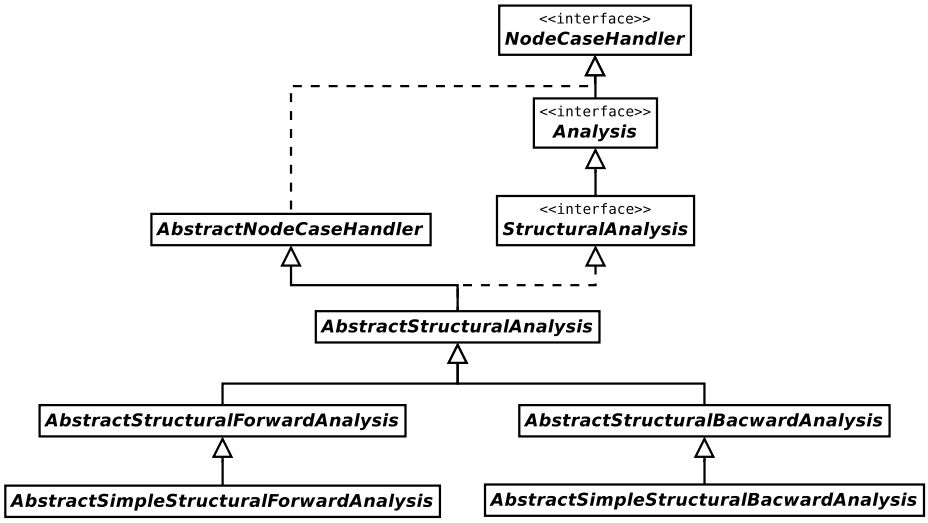


Figure 5.11 Class hierarchy snippet for structural analysis

Forward Analysis

A structural flow analysis is intended to flow information through a program in a particular direction. In the case of a forward analysis, the direction of flow is forward through the program. To accomplish this, the `AbstractStructuralAnalysis` defines data members that represent the in and out flow. In a forward analysis, the in flow at a given node is the information flowing into that node. When creating a forward analysis, a programmer must define how a given node's out flow is defined in terms of its in flow. In the framework, this is done by overriding appropriate node cases. In the implementation of these overridden node cases, the programmer must use the `currentInSet` and compute a value for `currentOutSet`. By default, if no overriding implementation is given, the in flow will passed along to the out flow directly. Providing appropriate implementations of desired node cases is only one step of implementing a flow-sensitive analysis. A list of all the steps involved in creating a new flow analysis is given in *Figure 5.16*

To create a forward analysis, a programmer must extend `AbstractStructuralForwardAnalysis`. This class implements the basic computations to perform a forward flow analysis. These computations include basic traversal of non-branching code, splitting and merging non-looping branching code, and the fixed-point computations for looping code. These computations are implemented in the case methods for various node types.

The `caseASTNode(...)` 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 `currentInSet` is set to the previous `currentOutSet`.

The `caseIfStmt(...)` and `caseSwitchStmt(...)` 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 `caseIfCondition(...)`. When a programmer provides true and false flow-data, `caseIfStmt(...)` will ensure that each branch of the **`if`** will have the appropriate in flow-data. The branching flow-data can be set and accessed through the methods described in *Table 5.9*.

`void`	`setTrueFalseOutSet(A tSet, A fSet)`	sets both true and false flows at once
`void`	`unsetTrueFalseOutSet()`	unsets both true and false flows
`A`	`getTrueOutSet()`	gets the true flow
`A`	`getFalseOutSet()`	gets the false flow

Table 5.9 Methods associated with branching analyses

The procedure for dealing with **`if`** statements is summarized by the diagram in *Figure 5.12*. This diagram illustrates how data is flowed through a typical **`if`**. The flow of data is represented by the arrows. First the condition is analyzed, then the resulting out flow from that is used to analyze the *then* part and *else* part. In the diagram the dashed arrow labelled “true flow” represents the true out flow and the dashed arrow labelled “false flow” represents the false out flow. The results of the *then* and *else* parts are then fed into the merge operation, depicted by the box containing a $\bowtie$ symbol. This merged result is the out

flow for the **if** statement.

The procedure for dealing with `switch` statements is similar to the one for **if** statements. However, there is no branching analysis for `switch` statements. The diagram illustrating the data flow is given in *Figure 5.13*. Keep in mind that, in MATLAB, a `case` can contain an arbitrary expression, and the result of evaluating it is used to decide if the case body is evaluated. Also recall that, unlike in C or Java, there is no fall through in MATLAB's `switch` statements, they simply execute the matching case's body and nothing else.

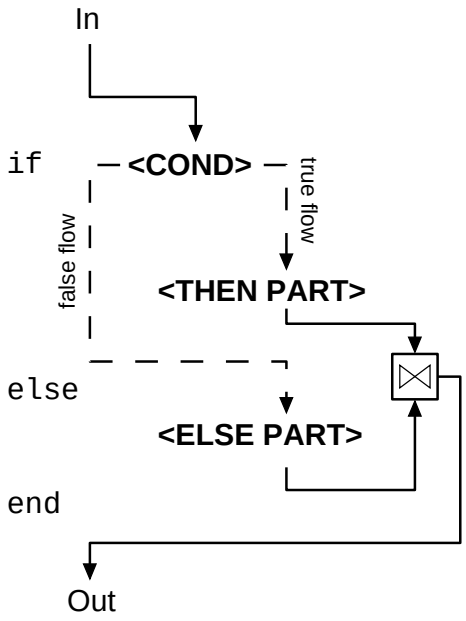


Figure 5.12 Forward data flow for **if** statements

The `caseForStmt(...)` and `caseWhileStmt(...)` implement the procedures for looping code. These procedures perform fixed-point computations and ensure that data is flowed correctly. This fixed-point computation is made more complex by the presence of `continue` and `break` statements. These statements can disrupt the normal flow through the body of a loop. To manage these we provide a stack called `loopStack`. This stack holds data associated with the loops being processed. This data is stored in an instance of a nested class called `LoopFlowsets`. A `LoopFlowsets` instance contains a reference to the loop node it's associated with, the in flow for that loop, and two lists of flow-data that store

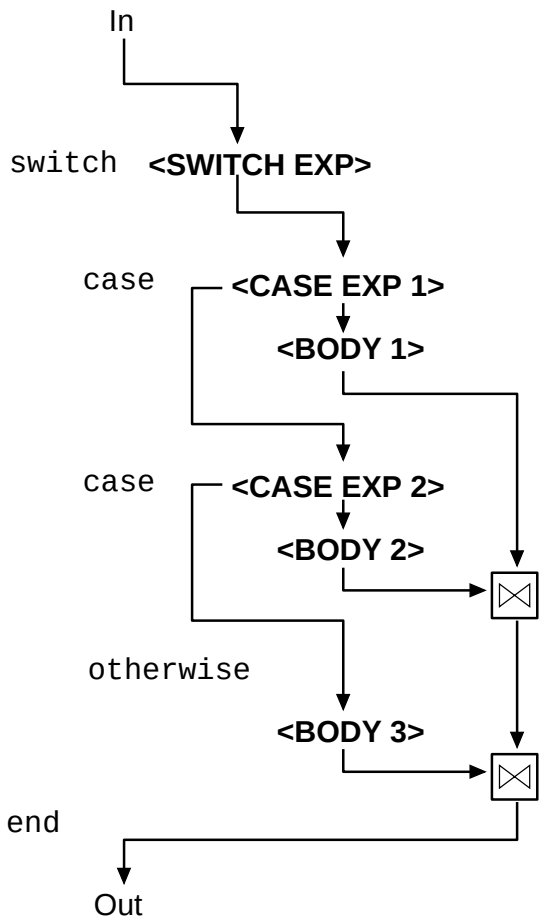


Figure 5.13 Forward data flow for `switch` statements

the out flows of all `continue` and `break` statements relevant to the loop. A list of the methods provided by this class is give in *Table 5.10*. The cases for loops setup a `LoopFlowsets` instance for the current loop and push it onto `loopStack`. The cases also make use of two abstract methods declared by `AbstractStructuralForwardAnalysis`. These methods are `processBreaks()` and `processContinues()`, each of which returns flow-data of type `A`. The intention is for a programmer to implement appropriate `caseBreakStmt(...)` and `caseContiueStmt(...)` methods that add data to the top of the `loopStack`. A programmer must also implement the `processBreaks()` and `processContinues()` methods. They should somehow combine all data from the appropriate list in the head of the `loopStack`.

Providing a simple implementation of these methods is the main focus of `AbstractSimpleStructuralForwardAnalysis`. The implementation for the two case methods will copy the in flow for the node and add it to the appropriate list at the head of the stack. The process methods are implemented to use the merge operator to merge all data in the appropriate list. These implementations should be satisfactory for many analysis implementations.

`void`	`initLists()`	initializes flow lists
`void`	`addContinueSet( A flowSet )`	add a flow to the `continue` list
`void`	`addBreakSet( A flowSet )`	add a flow to the `break` list
`List<A>`	`getBreakOutSets()`	returns the `break` list
`A`	`getLoopInFlow()`	returns the loop in flow
`void`	`setLoopInSet(A loopInFlow)`	sets the loop in flow
`List<A>`	`getContinueOutSets()`	returns the `continue` list
`ASTNode`	`getLoopNode()`	returns the associated loop node
`void`	`setLoopNode(ASTNode loopNode)`	sets the associated loop node

NOTE: the `List` being used here is the defined by `java.util.List`, not `ast.List`

Table 5.10 Methods provided by `AbstractStructuralForwardAnalysis.LoopFlowsets`

The flow of data for a basic **`while`** loop is depicted in *Figure 5.14*. The **`while`** loop being depicted contains two `continue` and two `break` statements. The boxes containing **C** and **B** represent `processContinues()` and `processBreaks()` respectively.

The fixed-point computation operates by storing the previous result of analyzing the condition check, and comparing it with the new result. If they are equal then a fixed-point is reached, then the rest of the computation continues, otherwise, another iteration of the fixed-point computation is executed.

It should be noted that this diagram has some dependency issues. The merge at the top of the diagram takes the in flow for the loop and the flow resulting from analyzing the body. The fixed-point check requires the out flow from the condition to be compared to the previous out of the condition. On the first iteration of the fixed-point computation, the body has not yet been analyzed, and there is no previous value for the condition. In order to bootstrap this process, the framework uses the `newInitialFlow()` method to approximate the result from the body, and the first fixed-point check is skipped. This value should be

designed to be safe for this purpose.

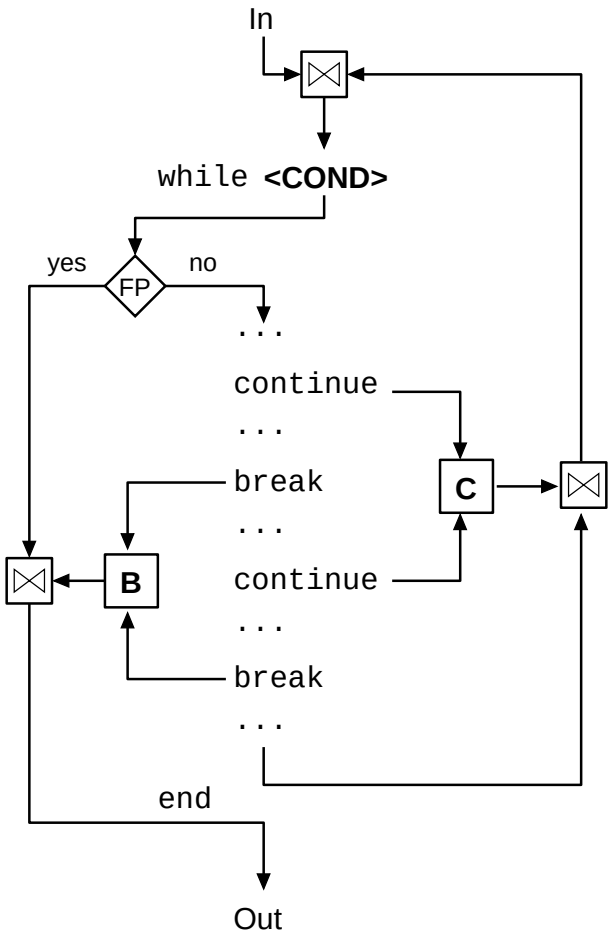


Figure 5.14 Forward data flow for `while` loops

The procedure for analyzing a `for` loop is very similar. The main difference is that instead of a simple condition, a `for` loop has a loop variable. A diagram depicting the procedure is given in *Figure 5.15*. The diagrams shows that analyzing the loop variable, depicted by **LV**, is split into its three phases: initialization, condition check, and update. Recall that in MATLAB, a `for` loop is always a for-each loop. The domain of the loop variable is determined by the result of an arbitrarily complex expression. The framework is designed to work with *McAST*, but a programmer must understand the semantics of the `for` loop. It is their responsibility to write their analysis to behave correctly when dealing with such complex `for` loops. The simplified IR, *McLAST*, describe in *Chapter 3*, and the

associated simplifications in *Chapter 4*, expose these semantics. *McLAST* guarantees that all `for` loops are simple range loops of the form `for i = start:step:stop`. This makes `for` loops much simpler to deal with.

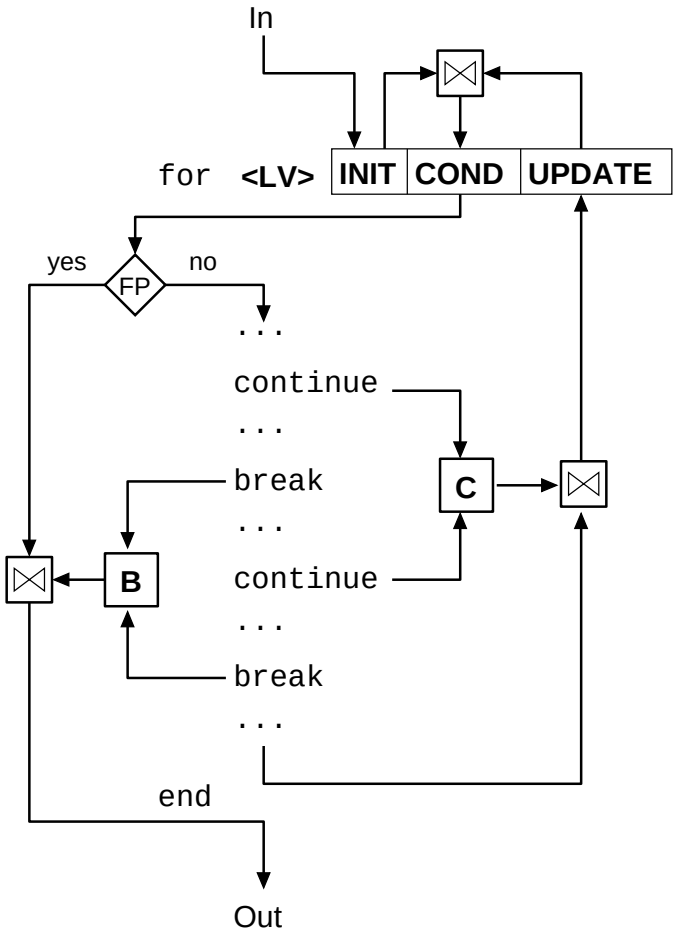


Figure 5.15 Forward data flow for `for` loops

These implementations should be adequate for most analyses. A programmer should generally not have to implement their own versions or even look into the source code for these methods.

To demonstrate the process of creating an analysis, we present an example analysis and step through the process. The example being implemented is the well known reaching definitions analysis. We will define this analysis as follows:

> For every statement *s*, for every variable *v* defined by the program, compute the set of all assignment statements that assign to *v* and reach *s*.

We will call our analysis `ReachingDefs`. To implement this analysis we will complete the tasks listed in *Figure 5.16*. We will start by picking a flow-data representation and merge operation.

- pick a flow-data representation
- define a merge operation
- define a copy operation
- define an initial flow
- define appropriate node cases
- define other necessary traversal methods
- ensure that data is copied and stored when needed

Figure 5.16 Steps to creating a new flow analysis

Based on the description of the analysis, we will define our flow-data as a mapping from variable names to sets of assignment statements. Each statement will have such flow-data associated with it. They will have both the flow-data from before evaluating the statement and after. Recall that the analysis framework provides a way of mapping particular AST nodes to flow-data. We will implement this data representation as a `HashMapFlowMap<String, Set<AssignStmt>>`. This lets us create a shell for our analysis. Code for this shell is given in *Figure 5.17*. Note that we are extending `AbstractSimpleStructuralForwardAnalysis`, and storing data as `HashMapFlowMap<String, Set<AssignStmt>>`. We are using `AbstractSimpleStructuralForwardAnalysis` because it gives a more complete implementation, and because the methods it provides are adequate for this analysis. More detail regarding this is given later in this section.

Recall from *Section 5.2.1* that to use one of the merge operations provided by `AbstractFlowMap`, either the value type must implement `Mergable` or an implementation of

```
public class ReachingDefs
   extends AbstractSimpleStructuralForwardAnalysis
         <HashMapFlowMap<String,Set<AssignStmt>>>
{
    ...
}
```

Figure 5.17 Shell of the reaching definitions implementation

`Merger` must be provided. Since set does not implement `Mergable` we will have to provide a `Merger` implementation.

We are going to be using the union operation for merging, and we want our `Merger` to union the two sets it is given. We want union because when two program paths reach a confluence point, such as two branches of an **`if`**, all definitions that reached that point on either path will reach immediately after the confluence point. We implement our `Merger` as an anonymous class instance stored as a private member of the analysis we are writing. The code for our implementation is given in *Figure 5.18*.

```
private Merger merger = new Merger<Set<AssignStmt>>(){
   public Set<AssignStmt> merge(Set<AssignStmt> s1, Set<AssignStmt> s2)
   {
      Set<AssignStmt> ms = new HashSet<AssignStmt>( s1 );
      ms.addAll( s2 );
      return ms;
   }
};
```

Figure 5.18 Implementation of `Merger` for reaching definitions

With this definition of a merger we can define our merge operation. We will use the `union(...)` method defined by `AbstractFlowMap`. This makes our merge operation very short to write. The code for this method is given in *Figure 5.19*.

Next we must define a `copy(...)` method. Since our flow-map contains mutable data, we will want to perform a deeper copy than `AbstractFlowMap` provides. This will involve creating a new `HashMapFlowMap`, going through all the keys in the original map and putting the key with a copied set into the out map. We also define a single argument `copy(...)` that returns a copy of the input, for convenience. Code for these methods is given in *Figure*

```
public void merge( HashMapFlowMap<String,Set<AssignStmt>> in1,
              HashMapFlowMap<String,Set<AssignStmt>> in2,
              HashMapFlowMap<String,Set<AssignStmt>> out )
{
   in1.union( merger, in2, out );
}
```

Figure 5.19 Implementation of `merge(...)` for reaching definitions

5.20

```
public void copy( HashMapFlowMap<String,Set<AssignStmt>> in,
             HashMapFlowMap<String,Set<AssignStmt>> out )
{
   if( in == out )
      return;
   out.clear();
   for( String i : in.keySet() )
      out.put(i, new HashSet<AssignStmt>(in.get(i)));
}
public HashMapFlowMap<String,Set<AssignStmt>>
   copy( HashMapFlowMap<String,Set<AssignStmt>> in )
{
   HashMapFlowMap<String,Set<AssignStmt>> out = in.emptyMap();
   copy(in, out);
   return out;
}
```

Figure 5.20 Implementation of `copy(...)` methods for reaching definitions

Defining a `newInitialFlow(...)` method is slightly more complicated. The way we defined the analysis implies that variables that have not yet been defined should map to an empty set. This means our map should always contain entries for all variables defined in the program. One easy way to do this is to rely on the `NameCollector` analysis written in *Section 5.2.3*. Recall this analysis provides the set of all variable names defined in the program, and for each assignment, a set of variables defined by that assignment. We will use the former to define our initial flow, and the latter in our analysis. In order to have this information available we need to run and store the `NameCollector`. We will add a field to `ReachingDefs` to store the analysis, and run it in the constructor. We will also compute and store a prototypical initial flow in the constructor. Code for the constructor

and declarations of the needed data members is provided in *Figure 5.21*. The resulting definition for `newInitialFlow()` is very simple. Its code is given in *Figure 5.22*.

```
1  private HashMapFlowMap<String,Set<AssignStmt>> startMap;
2  private NameCollector nameCollector;
3  public ReachingDefs( ASTNode tree )
4  {
5     super(tree);
6     startMap = new HashMapFlowMap<String,Set<AssignStmt>>( merger );
7     nameCollector= new NameCollector(tree);
8     nameCollector.analyze();
9     for( String var : nameCollector.getAllNames() )
10       startMap.put( var, new HashSet<AssignStmt>() );
11 }
```

Figure 5.21 Implementation for reaching definition's constructor

```
1  public HashMapFlowMap<String,Set<AssignStmt>> newInitialFlow()
2  {
3     return copy(startMap);
4  }
```

Figure 5.22 Implementation of `newInitialFlow()` for reaching definitions

Finally we can define the appropriate node cases needed for the analysis. When defining our node cases, it's important to make sure that data is copied and stored correctly. The framework only guarantees that `currentInFlow` is set to the appropriate flow data. It doesn't ensure that this data is not shared. The previous node could have stored a reference to the flow-data somewhere else. In this case, modifying the data will change the result for the previous node. To avoid modifying data associated with other nodes, care must be taken to have a correct and consistent copying strategy. The simplest such strategy is to copy everywhere. This is not done by the framework to allow programmers to share data when they are sure it's safe in order to save memory and copying time.

It's also important to ensure that data is being associated with appropriate nodes. If care is not taken, then some data can be lost.

For the reaching definitions analysis we only need to define two cases. The first, and most important case is `caseAssignStmt(...)`. This case will do the work of updating the

flow information for the variables defined by this statement. It will use the `NameCollector` we ran in the constructor to get the variables that are defined by the given assignment. It will then create a new set containing only the given `AssignStmt` node and associate it with each of the defined variables. This means that immediately after this point, each of those variables only has one reaching definition. The case will also take care of storing the in and out flow to the given node, and copying appropriately. We store the flow information in the `inFlowSets` and `outFlowSets` maps, this is so it is available after the analysis is run.

Our copying strategy will be to assume the flow-data coming in is safe to store but not safe to modify. This means we will make a copy of the in flow-data for modification. This modified flow data becomes our out flow-data and is stored in `outFlowSets` for the given node. This means that the in flow-data of a given node can be shared with the out flow-data of a predecessor node. It is however not always going to be shared. This is because the predefined `caseForStmt(...)`, `caseWhileStmt(...)`, `caseIfStmt(...)`, and `caseSwitchStmt(...)` all perform aggressive copying to ensure no incorrect behaviour.

The definition of the analysis also requires that each statement have an associated reaching definitions mapping. The predefined cases always perform this mapping, and so does our `caseAssignStmt(...)`. In order to have all other statements mapped to reaching definition data, we can define a `caseStmt(...)` method. Recall from *Section 5.1* that this method will only be executed by other statements with undefined case methods. The definition for this method will be very simple, it will associate the in flow to the node, assign the out flow to be the in flow, and associated the out flow to the node. No copying is done, which is consistent with the assumption made for our copying strategy in `caseAssignStmt(...)`. The code for `caseAssignStmt(...)` is given in *Figure 5.23* and the code for `caseStmt(...)` is given in *Figure 5.24*.

This is sufficient to define our `ReachingDefs` analysis. A listing of the complete source code for this analysis is provided in *Appendix A*.

This example demonstrates the steps needed to define a new analysis. It should be noted that this example is naive, in that it does not take into account the far reaching side effects possible in MATLAB. For instance we ignore `eval` expressions that can cause new definitions. We also ignore that any function call can cause new definitions. A full and rigorous analysis would be more complex, and would rely on the results of other analyses,

```
1  public void caseAssignStmt( AssignStmt node )
2  {
3     inFlowSets.put( node, currentInSet );
4     currentOutSet = copy(currentOutSet);
5     Set<String> defVars = nameCollector.getNames( node );
6      for( String n : defVars ){
7        Set<AssignStmt> newDefSite = new HashSet<AssignStmt>();
8        newDefSite.add( node );
9        currentOutSet.put( n, newDefSite );
10    }
11    outFlowSets.put(node, currentOutSet);
12 }
```

Figure 5.23 Implementation of `caseAssignStmt(...)` for reaching definitions

```
1  public void caseStmt( Stmt node )
2  {
3     inFlowSets.put( node, currentInSet );
4     currentOutSet = currentInSet;
5     outFlowSets.put( node, currentOutSet );
6  }
```

Figure 5.24 Implementation of `caseStmt(...)` for reaching definitions

such as the Kind Analysis describe in *Section 5.2.5*. Such an analysis is outside the scope of this section.

Backward Analysis

The implementation of a backward analysis is very similar to a forward analysis. Conceptually they are also very similar, the main difference being that, where a forward analysis flows information forward through the program, a backward analysis flows it backwards. This means that instead of defining the out flow in terms of the in flow, a programmer must define the in flow in terms of the out flow. The steps for creating a backwards analysis are the same as the ones given for forward analyses in *Figure 5.16*.

To create a backward analysis, a programmer must extend `AbstractStructuralBackwardAnalysis`. This class is similar to `AbstractStructuralForwardAnalysis`, but provides implementation details specific to backwards analyses.

As with other analysis types, `caseASTNode(...)` implements the basic traversal. It will loop through the children of the given node in reverse order, analyzing each one in turn. Like in a forward analysis, the `analyze(ASTNode node)` method is used to analyze a given node. Unlike forward analyses, the default implementation of this method guarantees that `currentOutSet` is set to the previous `currentInSet`. This enforces the backwards nature of such an analysis.

The `caseASTNode(...)` method provides default behaviour for non-branching code. Behaviour for branching code is given by `caseIfStmt(...)`, `caseSwitchStmt(...)`, `caseWhileStmt(...)`, and `caseForStmt(...)`. In principle, these work very similar to their forwards analysis counterparts, but obviously with the direction of flow reversed. There are some important details that differ between the two versions, so we will provide brief descriptions.

The `caseIfStmt(...)` for backwards analyses does not provide branching analysis functionality. Besides this difference, the `caseIfStmt(...)` and `caseSwitchStmt(...)` are very similar to their forwards counterparts. Diagrams representing how information flows through these statements are given in *Figure 5.25* and *Figure 5.26* respectively.

The loop cases differ more from their forwards versions, compared to the other cases. This is due to the added complexity of performing a fixed-point computation, and due to `continue` and `break` statements. Like `AbstractStructuralForwardAnalysis`, `AbstractStructuralBackwardAnalysis` uses a stack called `loopStack` to keep track of loop flow-data. It also provides a nested class implementation of `LoopFlowsets`. This implementation stores the AST node and out flow-data for the loop being analyzed. It is also used for storing and accessing data for dealing with `break` and `continue` statements. In a backwards analysis, each `break` statement and each `continue` statement for a given loop are expected to have the same out flow, respectively. Consequently `LoopFlowsets` only stores a single out flow for `break` statements and a single out flow for `continue` statements for the given loop. A summary of the methods provided by this implementation of `LoopFlowsets` is given in *Table 5.11*. The out flow for `break` statements is set by the `setupBreaks()` method, which is called at the beginning of the loop case. The out flow for `continue` statements is set by the `setupContinues()` method, which is called before analyzing the body of the loop in each iteration of the fixed-point computation. Diagrams

depicting the flow of information through **`while`** and **`for`** statements are given in *Figure 5.27* and *Figure 5.28*.

Like for forward analyses, the framework provides a simple abstract implementation for backward analyses called `AbstractSimpleStructuralBackwardAnalysis`. This class provides implementations for methods relating to `break` and `continue` statements. It provides a `setupBreaks()` implementation that simply sets the out set for `break` statements to the current out flow. The implementation for `setupContinues()` does the same for `continue` statements. The implementations for the `break` and `continue` cases sets the in flow to the appropriate out flow from the head of the `loopStack`, performing a copy of the data for safety.

`A`	`getBreakOutFlow()`	returns the out flow-data for `break` statements in this loop
`void`	`setBreakOutFlow(A outFlow)`	sets the out flow-data for `break` statements in this loop
`A`	`getLoopOutFlow()`	returns the out flow-data for the current loop
`void`	`setLoopOutFlow(A outFlow)`	sets the out flow-data for the current loop
`A`	`getContinueOutFlow()`	returns the out flow-data for `continue` statements in this loop
`void`	`setContinueOutFlow(A outFlow)`	sets the out flow-data for `continue` statements in this loop
`ASTNode`	`getLoopNode()`	returns the AST node for the current loop
`void`	`setLoopNode( ASTNode loopNode )`	sets the AST node for the current loop

Table 5.11 Methods provided by `AbstractStructuralBackwardAnalysis.LoopFlowsets`

To demonstrate the implementation of a backwards analysis, a live-variable analysis has been created and included as an example in the framework. The source code for this analysis is also provided in *Appendix C*. This implementation uses a depth first analysis called `UseCollector`. This analysis is similar to the `NameCollector` given in *Figure 5.10*, but instead of collecting all name that are assigned to, it collects all names that are used as possible variable accesses. To accomplish this, the analysis uses the kind analysis to

determine if a name is a variable or not. The kind analysis is briefly describe in *Section 5.2.5*. The source code for `UseCollector` is given in *Appendix B*.

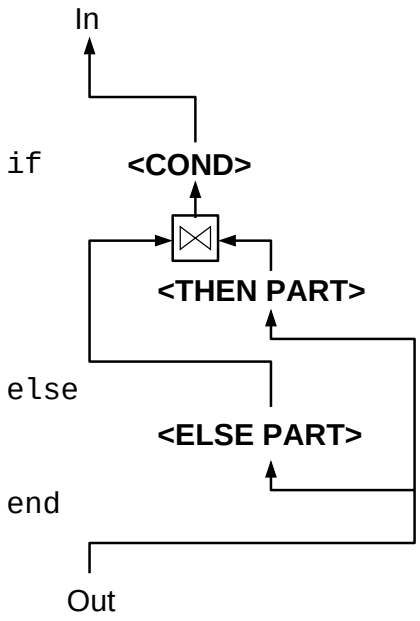


Figure 5.25 Backward data flow for `if` statements

Important Points

In order to implement a correct flow analysis, it's important to remember a few things. These are summarized here.

- A programmer must keep MATLAB semantics in mind. This is particularly important when dealing with *McAST* instead of *McLAST*.
- A programmer must ensure that the contract for in and out flow-data is respected. For a forwards analysis this means having `currentOutFlow` set to the appropriate value at the end of a case method. It also means that, when not using `analyze(...)` to analyze a particular sub-node (e.g. calling a case method for that node directly), the `currentInFlow` is set to an appropriate value for that node. For a backwards analysis, the contract is reversed.

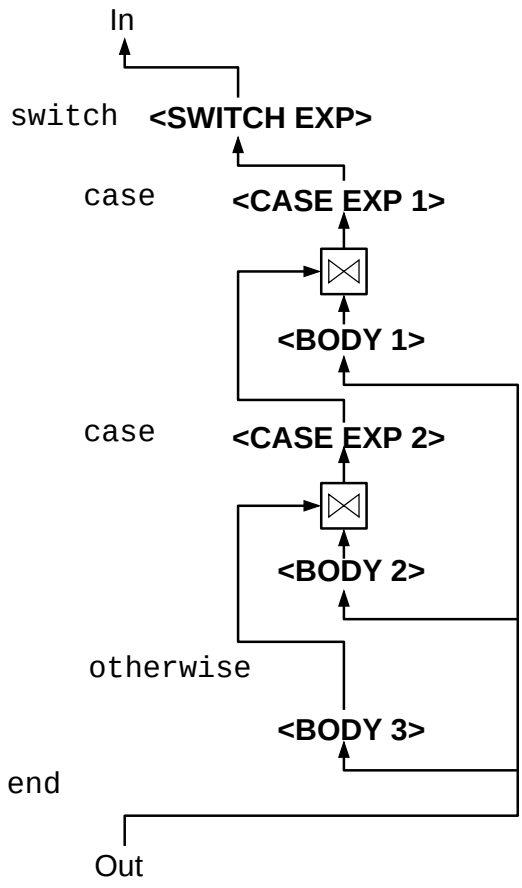


Figure 5.26 Backward data flow for `switch` statements

- It is a programmer's responsibility to copy flow-data appropriately. The contract for flow-data only says that the data in `currentInFlow` or `currentOutFlow` is appropriate, not that they are safe to modify.

5.2.5 Implemented Analyses

Besides the example analyses discussed in this chapter, two important analyses have been created as part of this thesis. These analyses were created to provide information to other parts of the compiler. The first analysis is the Kind Analysis, the second is the Handle Propagation Analysis.

The Kind Analysis has been mentioned throughout this thesis. It provides basic infor-

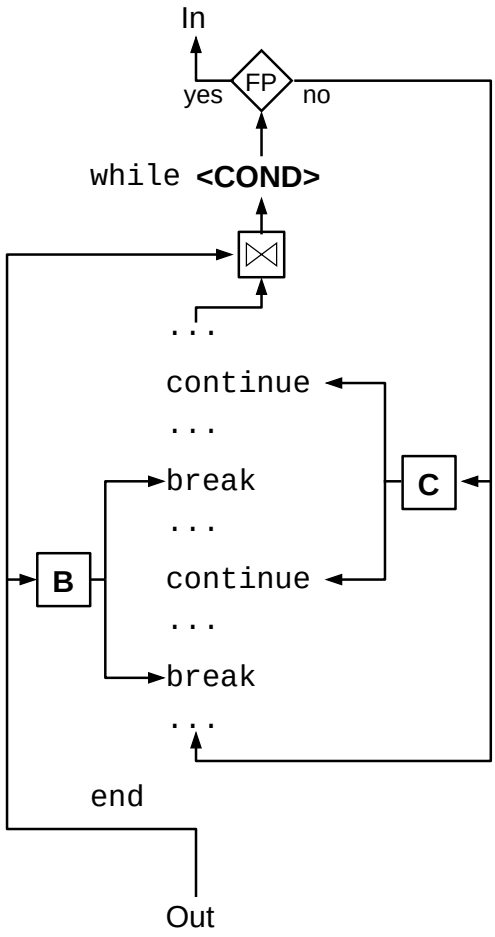


Figure 5.27 Backward data flow for `while` loops

mation about the kind of a given identifier. By this, we mean if a given identifier refers to a function or a variable. At every statement in the program, the Kind Analysis can be queried to check if a given identifier name is considered a variable or function. These are represented by the values *VAR* and *FUN*. When the analysis cannot determine the exact kind for an identifier, it will assign it the kind *ID*. The analysis also detects certain errors, such as when an identifier is used as both a variable and a function. Further information on this analysis can be found in the Kind Analysis paper [DHR11].

The Handle Propagation Analysis was created as a step towards constructing an accurate call graph for MATLAB programs. It identifies handle creation sites and propagates the handle information through the program. This allows other analyses to determine if

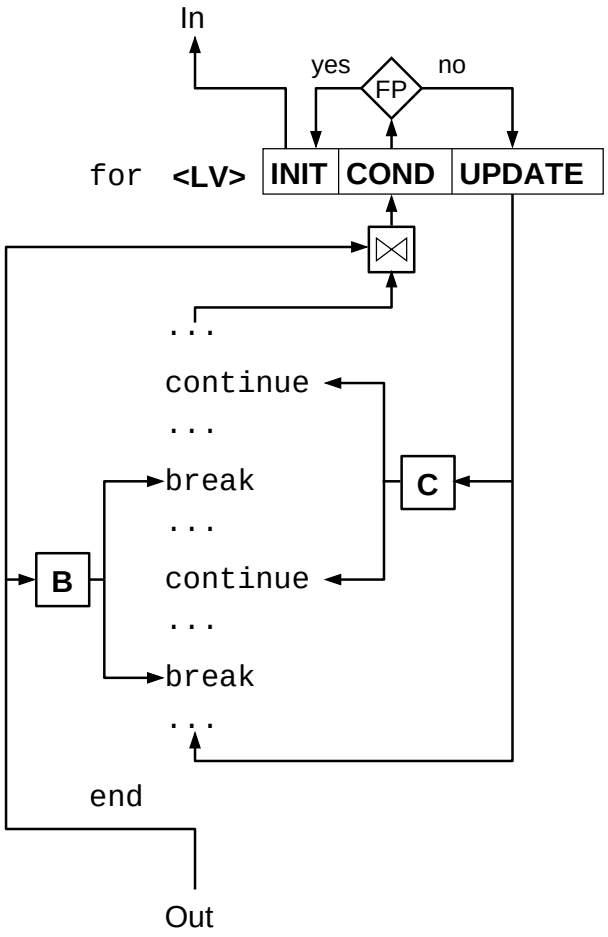


Figure 5.28 Backward data flow for `for` loops

a parameterized expression can in fact be a function call, despite the target not being an obvious function.

The source code for these analyses are include as part of the *McLAB* project. They represent fundamental analyses, and are a practical application of this analysis framework.

Chapter 6
Analysis Framework Extensibility

Up to this point we have discussed the analysis framework in the context of the MATLAB language. The *Mc*LAB framework is intended to be extensible to new language features. This means that the analysis framework must also be extensible. This section describes the extensibility of the analysis framework.

First of all, we must define what we mean by extensibility. The system is designed to be extensible in the sense of creating new language extensions. The *Mc*LAB framework defines a root language based on MATLAB called NATLAB. This root language is defined by the `natlab` Java package. A language extension represents a new language based off of another language, called the base language. The new language will then extend the scanner, parser, and AST definitions from the base language. These three components have been created using tools that make such extensions simple to create. The important thing to note is that these extensions should be considered separate from the languages they are based on. It is this type of extensibility that we have focused on for the analysis framework.

6.1 Classification of Extensions

There are three basic types of language extension, with relation to the analysis framework. The first, is a language extension that adds new AST nodes, but these nodes do not play a part in any analysis. These are nodes that either have no semantic content, such as a new comment node, or are nodes that analyses will never encounter, such as syntactic sugar that

gets desugared before analyses are run. This type of extension is very simple to implement. Such an extension simply needs to include all necessary files from its base language, including AST files, other JastAdd files, necessary Java class files, and the scanner and parser files. Note that if the base language is itself an extension of another language, then all files that are needed from the deeper base language must also be included. Also note that this inclusion does not mean the files are copied. Instead, the build process of the extension language needs to have access to them, either for inclusion in the JastAdd, scanner, or parser code generation process, or as part of the Java class path. Once all needed files are included, the extension needs to define its new AST nodes. As an example of this type of extension, a model language extension, called `ExExtension1`, has been created, and is available in the *Mc***LAB** project. This extension defines a new node `IncStmt` that is intended to be a unary increment statement. This new statement is treated as syntactic sugar and is desugared into an assignment statement. For example, `a++` will be desugared into `a=a+1`.

The other two types of language extensions involve adding nodes that *will* play a part in analyses. They are distinguished by the type of nodes they add. The first of these other types will only add nodes that don't introduce new forms of control flow. This could include new operators or new types of simple statements. An example of this type of extension is provided in the *Mc***LAB** project. The name of this example is `ExExtension2`. It is actually an extension of `ExExtension1` and demonstrates the process of extending a language that is itself an extension. `ExExtension2` doesn't add new nodes, but instead changes the meaning of `IncStmt`. In this extension `IncStmt` is no longer syntactic sugar. This means that the analysis framework needs to be extended to take the new statement into account. In addition to extending the framework, any analyses that will be used in the extension also need to be extended. To demonstrate this, `ExExtension2` includes an extended version of the `NameCollector` analysis described in *Section 5.2.3*. The process of extending the analysis framework in this way is straight forward and would be a good candidate for automation. Extending individual analyses, on the other hand, requires knowledge of the analyses being extended, and of the semantics related to any added AST nodes.

The final type of language extension is one that adds new control flow nodes. An example of this type of extension would be adding a new parallel `for` loop. This is the most complex form of language extensions. It requires that the analysis framework be

extended to include new fixed-point computations and proper control flow traversal for the new nodes being added. On the bright side, this should usually capture all the added semantics of these nodes, allowing existing analyses to be used without extending them to handle the new control flow nodes.

6.2 How Extensions are Supported in *McSAF*

The rest of this chapter discusses how *McSAF* was designed to allow these types of extensions. In particular we focus on the second and third type of language extensions, since the first type is achieved mostly thanks to the JastAdd tool.

The extensibility of the framework is built into the packaging of the classes and the class hierarchy. The node case handler related classes belong to the `nodecases` package, the analysis classes belong to the `analysis` package. In order to understand the extensibility of *McSAF*, we need to describe the class hierarchy in more detail. These details are the crux of the framework's extensibility. The sub-class relationship represented by those hierarchies is still correct; there are however additional classes and interfaces mixed in. These additional classes and interfaces provide the actual definitions and implementations for NATLAB and any extensions. The classes we've talked about so far are simply the user-facing names for those components. An analysis programmer should only have to be concerned with those classes, not any of the additional classes discussed in this section. These user-facing classes are in fact basically empty except for code to specify the implementation class that contains its content and other boiler plate code needed for it to compile and run correctly.

In order to make the structure more concrete, we demonstrate it with the hierarchy behind forward analyses. This include the `NodeCaseHandler` down to `AbstractStructuralForwardAnalysis`. `AbstractSimpleStructuralForwardAnalysis` is not included for the sake a brevity. The user-facing class hierarchy for these classes can be seen in *Figure 5.11*.

To start we focus on the node case handlers. Recall that classes related to this portion of the framework are located in the `nodecases` package. This package contains one user-facing interface, `NodeCaseHandler`, and one user-facing abstract class, `AbstractNodeCaseHandler`. Since these are user-facing, they will be basically empty. Their content

will come from corresponding files in the `nodecases.natlab` package. The corresponding files are `NatlabNodeCaseHandler` and `NatlabAbstractNodeCaseHandler`. It is the files in `nodecases.natlab` that contain the actual content. `NatlabNodeCaseHandler` contains all the method definitions described in *Section 5.1*, and `NatlabAbstractNodeCaseHandler` contains all the default implementations. `NodeCaseHandler` and `AbstractNodeCaseHandler` simply contain code to define the interface or class and extend the corresponding `nodecases.natlab` version. Source code for the actual implementation of `NodeCaseHandler` and `AbstractNodeCaseHandler` are given in *Figure 6.1*.

```
package nodecases;

public interface NodeCaseHandler
   extends nodecases.natlab.NatlabNodeCaseHandler
{
}
```

```
package nodecases;

public abstract class AbstractNodeCaseHandler
   extends nodecases.natlab.NatlabAbstractNodeCaseHandler
{
}
```

Figure 6.1 Actual `NodeCaseHandler` and `AbstractNodeCaseHandler` source code

Of course, since the hierarchy relationship depicted in *Figure 5.11* must hold, `AbstractNodeCaseHandler` must implement the `NodeCaseHandler` interface. The source code only states that `AbstractNodeCaseHandler` extends `NatlabAbstractNodeCaseHandler`, and doesn't mention any interface. For `AbstractNodeCaseHandler` to implement `NodeCaseHandler`, `NatlabAbstractNodeCaseHandler` must implement it. In general, only user-facing names should appear in source code, and when possible, interface references should be used instead of class references. An obvious exception to this is the definition of the user-facing classes and interfaces, which must use the non user-facing names.

The analysis classes are defined in a similar way. The full class hierarchy of the node cases and forward analysis related classes is depicted in *Figure 6.2*.

One difference with some of the analysis classes is that, since the abstract analysis

implementations define some constructors, the user-facing versions must also have those constructors defined. The behaviour will simply call the `super` version of the constructor.

6.2.1 Example Extension

So how can this be extended? As was stated, a language extension must be completely separate from its base language. Let's say that we are creating a language extension called EXT, which extends NATLAB. This extension defines a new node type `FooStmt` extending `Stmt`, and we want to extend the analysis framework to handle this new node. Here we will focus on extending the node case handler.

The very first thing that must be done is that `FooStmt` must be given an `analyze(NodeCaseHandler handler)` method. This method must call the `caseFooStmt(...)` on the handler. This is done by using a JastAdd aspect, defining the method. The example extension `ExExtension2` has this defined by its `ASTAnalyze.jadd` file.

Since the base node case handler didn't have a `caseFooStmt(...)`, we will need to extend the node case handler interface. To do this we create a new interface, `ExtNodeCaseHandler`, which is part of the `nodecases.ext` package. This new interface extends `NatlabNodeCaseHandler`, and include a new case method `caseFooStmt(FooStmt node)`.

In order for this extended interface to be used, a new version of `NodeCaseHandler` must be created. Like the NATLAB `NodeCaseHandler`, this should be in the `nodecases` package. The new one should be identical to the base version, except, instead of extending `nodecases.natlab.NatlabNodeCaseHandler`, it will extend `nodecases.ext.ExtNodeCaseHandler`.

`AbstractNodeCaseHandler` should be extended in a similar way, by creating `ExtAbstractNodeCaseHandler`. This should include the default forwarding behaviour for `caseFooStmt(...)`, which will forward to `caseStmt(...)`. A new `AbstractNodeCaseHandler` is created in a way similar to how the new `NodeCaseHandler` was created.

By judiciously including correct files in the build process, the EXT language code will have access to its node case handling classes, which includes the extended versions.[1]

[1]The build processes for `ExExtension1` and `ExExtension2` demonstrate how to include base language classes correctly. See the directory `mclab/Project/languages` in the *McLAB* project for these two extensions.

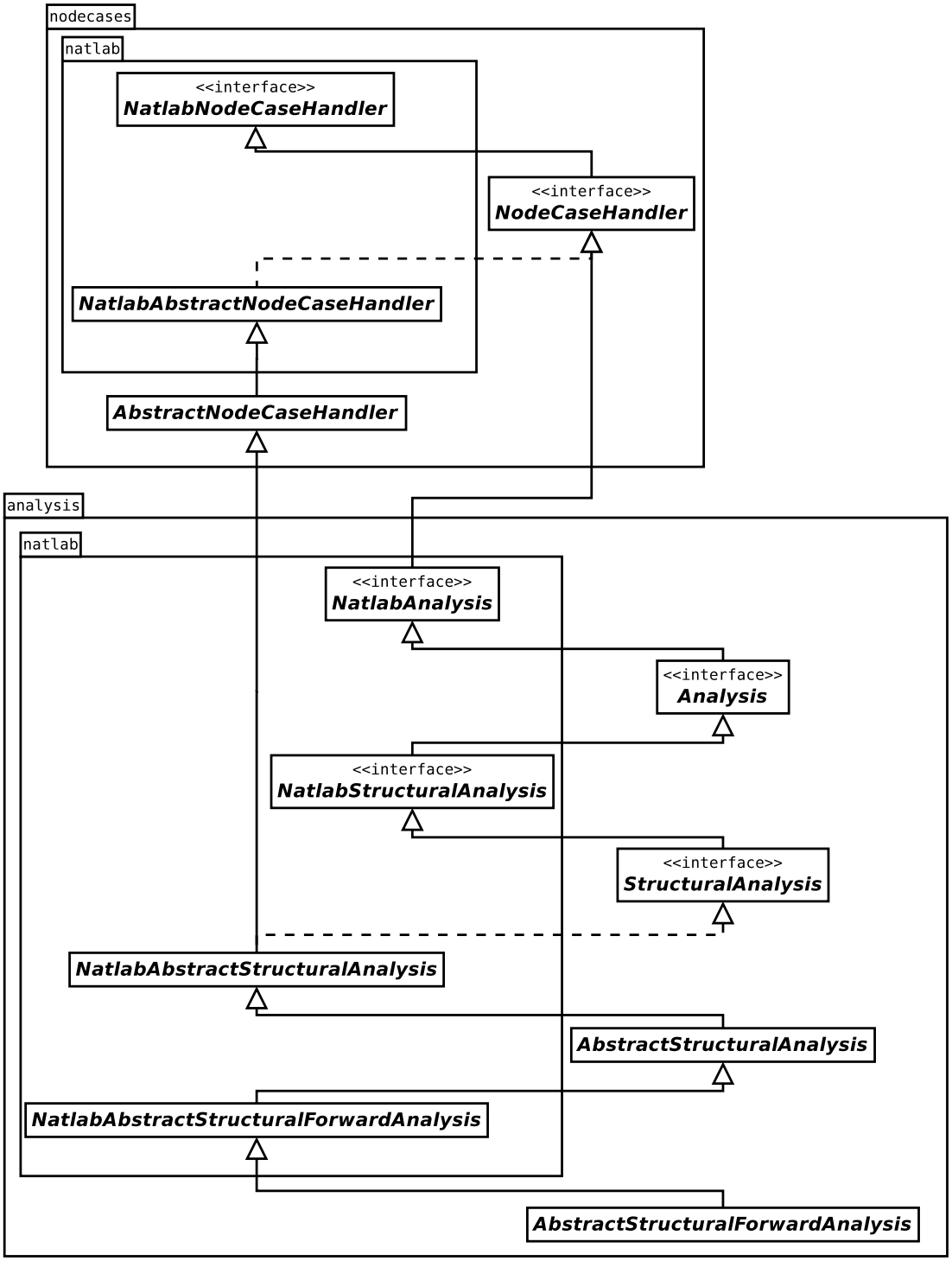


Figure 6.2 Class hierarchy for forward analyses including extensability details

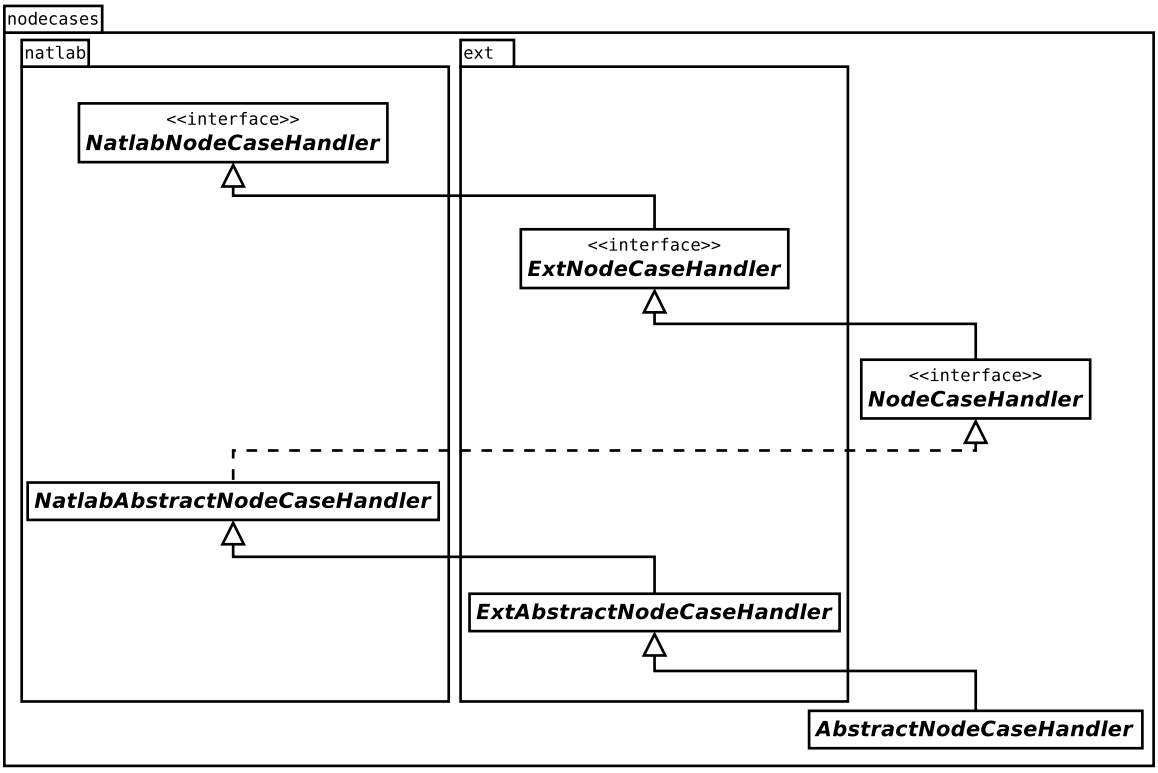
This process allows any node case handlers written for the base language, such as the statement counter described in *Section 5.1*, to run in EXT. This is because such code should only have used the user-facing names. For example, the statement counter was implemented by extending `AbstractNodeCaseHandler`. In EXT `AbstractNodeCaseHandler` includes `caseFooStmt(...)` with default forwarding behaviour.

Of course, if the new nodes require special behaviour in any of these traversals or analyses, then they would need to be extended. For example, the `IncStmt` needed special behaviour in the `NameCollector` analysis, since it defines a new name. That is why `ExExtension2` includes an extended version of `NameCollector` called `ExtendedNameCollector`. This demonstrates the process of extending an existing analysis to work with a new language extension.

In *Figure 6.3*, the class hierarchy for the node cases in EXT is given. It should be noted that all the classes in the `nodecases.natlab` package are located in the NATLAB language, and are not copied into the EXT. The `ExtNodeCaseHandler` and `ExtAbstractNodeCaseHandler` classes are located in the EXT language, and only include definitions for the added node. The `NodeCaseHandler` and `AbstractNodeCaseHandler` files are new files existing in EXT. They are identical to NATLAB's versions of these files, except they extend the classes in the `nodecases.ext` package. The class path when compiling and executing code in the EXT language, should be set so that these newer versions of `NodeCaseHandler` and `AbstractNodeCaseHandler` take priority over NATLAB's versions.

6.2.2 Other Issues

When extending, there are some extra considerations to be aware of. First of all, the structural forward analyses rely on a helper node case handler called `AnalysisHelper`. The cases in this class receive the callback from an `analyze(...)` call, perform some bookkeeping, and forward to the same case in the analysis being performed. The behaviour of this class is very simple, and should be obvious from its source code. In fact this code would also be a good candidate for automated code generation. When extending the analysis framework, the `AnalysisHelper` and `BackwardsAnalysisHelper` must be extended appropriately. The `ExExtension2` language extension includes examples of extending these



Note that the `NodeCaseHandler` and `AbstractNodeCaseHandler` classes are new versions of those classes, included in EXT.

Figure 6.3 Class hierarchy of the extended node case handler

two classes.

Creating a major language extension, one containing new control flow and loops, which requires the use of previous analyses, some of which requiring new node case implementations, and requiring new analyses to be written is not a small task. Such an extension requires knowledge of the framework, analyses, and semantics of both MATLAB and the extension language. In such a case, the framework provides a clear way of implementing the extension, and it does so in a way that can cleanly separate the extended functionality from the base functionality.

6.3 Summary

The analysis framework described in this chapter and *Chapter 5* provides a way of defining new intraprocedural analyses for the MATLAB language. It allows a programmer to create several different types of analyses, ranging from simple traversals to flow-sensitive analyses with fixed-point computations. It also defines a basic traversal mechanism that has become a useful tool in other parts of the *McLAB* project. This framework also provides a means to adapt to new language extensions. For simple extensions, it allows for very simple adaptation. When more complex extensions are needed, it provides a clear way of performing that extension, which avoid the need to re-implement or copy the entire framework. To demonstrate the process of creating language extensions, two example extensions are provided. These extensions are named `ExExtension1` and `ExExtension2`. They are available as part of the *McLAB* project, and can be found in the directory `mclab/Project/languages`.

Chapter 7
Related Work

The *Mc*LAB 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. There have been other projects that provide such a framework for other languages. There have also been other projects that have performed static analysis on MATLAB, but with no focus on making the analysis system an open research tool. In this chapter, we will discuss some of the existing work that relates to the contributions of this thesis.

7.1 Soot

Soot[VRHS+99] is an optimization framework for Java. This framework was created as an open research tool. Like *Mc*LAB, Soot provides different intermediate representations. It also provides a framework for creating static analyses. Soot is an open source tool, and more information can be found at the project home page[1].

7.2 JastAdd

The JastAdd toolkit is designed for creating extensible compilers. This toolkit was used in the development of *Mc*LAB. One feature of JastAdd that was not discussed in great detail

[1] http://www.sable.mcgill.ca/soot/

in this thesis is its attribute grammar system. JastAdd allows a developer to define attributes as part of their AST grammar. 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 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[NNHME09] has been done to implement flow analysis for Java using the JastAdd extensible Java compiler[EH07a].

7.3 MATLAB Related Work

In the past, there has also been some work towards compiling MATLAB. There was the Falcon project[RP99, DRG+95], which aimed to compile MATLAB code into FORTRAN. Falcon focuses on type inference and code inlining to produce FORTRAN code.

The Magica tool[JB02] 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.

Another compiler project for MATLAB is MaJIC [AP02]. MaJIC incorporates a Just-In-Time(JIT) compiler component. This allows it to achieve speedups similar to those produced by Falcon, without sacrificing the interactive nature of MATLAB.

There has also been some work towards source-to-source transformations[MP99]. These transformations are intended to improve performance by taking advantage of more efficient ways of writing MATLAB code. This is possible because there are certain language features that MATLAB performs more efficiently than others. For instance, using loops in MATLAB can result in fairly slow code. If the looping code can be rewritten to take advantage of MATLAB's vector operations, it can greatly improve execution speed. In fact, there is offi-

cial documentation[2] describing manual techniques for vectorizing code.

These projects differ from this thesis in that their main goal was to improve the performance of MATLAB programs. *Mc*SAF, 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 *Mc*SAF.

7.4 *Mc*LAB Related Work

More recently, the *Mc*LAB project has produced work related to optimizing and compiling MATLAB code. *Mc*FOR [Li09] is a static back-end that produces FORTRAN code. *Mc*FOR uses type inference to produce efficient FORTRAN code. It estimates array shapes and sizes in order eliminate array reallocation and array bounds checks, in order to reduce execution overhead. Work on this portion of *Mc*LAB is an ongoing project. The new work is being based off of, and incorporates *Mc*SAF and the work done towards this thesis.

*Mc*JIT [CBHV10] takes a dynamic approach to MATLAB compilation. It incorporates a virtual machine called *Mc*VM that acts as an interpreter. It uses profiling information to determine when to initiate just-in-time compilation, where it produces LLVM[Lat02] machine code. *Mc*JIT can take advantage of run-time information to produce specialized versions of compiled functions. *Mc*VM and *Mc*JIT use the front-end portion of *Mc*LAB in order to scan and parse inputted MATLAB. The analyses performed by *Mc*JIT were implemented separately from work done in this thesis. There is work currently being done to incorporate some of the contributions of this thesis into *Mc*VM. The first step of this effort is making the Kind Analysis information available to the VM.

[2] http://www.mathworks.com/support/tech-notes/1100/1109.html

Chapter 8
Conclusions and Future Work

MATLAB is a popular language for scientific and numerical programming. Due to its closed source and proprietary nature, there is a high overhead to researching compiler techniques targeted towards MATLAB and scientific languages. The *Mc***LAB** project tries to overcome this by developing open tools and frameworks aimed at MATLAB compiler research. As part of this project, and the topic of this thesis, we have developed the *Mc***LAB** Static Analysis Framework. The analysis framework is designed to make it simple to develop new analyses for MATLAB programs. It was also designed to allow the framework and existing analyses to be extended to new language features.

Developing this framework has required an investigation of MATLAB semantics. This investigation was necessary because there is no official specification for MATLAB; the language is defined by the latest implementation and a collection of informal documentation. It has also involved the definition of a simplified intermediate representation for MATLAB, called *Mc***LAST**. *Mc***LAST** is a restricted version of *Mc***AST**. This representation was necessary to make creating analysis simpler. It accomplishes this by restricting the complexity of expressions and statements, and exposing some of MATLAB's semantics, making them more explicit.

Having defined *Mc***LAST**, it was also necessary to implement a transformation to simplify *Mc***AST** into *Mc***LAST**. We also developed a tool for verifying that a given AST satisfies *Mc***LAST**'s restrictions.

The framework itself is an intraprocedural static analysis framework. It allows for

several different types of analyses to be written. These include a simple traversal based analysis that can be used to implement context insensitive analyses. Fixed-point based flow-sensitive analyses can also be written. These flow-sensitive analyses can be either forward or backward analyses.

Using this framework some example analyses were created. In addition, some generally useful analyses have also been created. In particular, the Kind Analysis was created and is used in other parts of *Mc***LAB** project.

The contributions of this thesis provide important tools for future research into compiler techniques targeting MATLAB and scientific programming. They will facilitate future development of program analyses by providing simpler and more exposed semantics, by providing a framework for simplifying the task of creating such analyses, example analyses that use this framework, and fundamental analyses that can provide basic information to future analyses.

8.1 Future Work

*Mc***SAF** is already being used in *Mc***LAB** in an integral way, but the development of *Mc***SAF** has opened up avenues for future work. The most obvious work to do is to continue using *Mc***SAF** to create new analyses. Either implementing standard analyses using the framework, or by creating entirely new analyses related to MATLAB and scientific programming. It would also mean creating new language extensions and writing analyses for them.

Another excellent opportunity for future work is the development of an interprocedural component of the analysis framework. Some work has already been done towards this goal. This work includes the Handle Propagation Analysis, which was created as a step towards creating an accurate call graph for MATLAB programs.

Finally, there is also the opportunity to create tools that will ease the burden of creating new language extensions. Much of the code that needs to be written for a language extension follows a very precise pattern. Tools could be created to generate this code, allowing the creator of an extension to focus on the design of the actual language extension.

Appendix A
Full Reaching Definitions Analysis Code

```
1  package natlab.toolkits.analysis.test;
2
3  import analysis.*;
4  import ast.*;
5  import java.util.Set;
6  import java.util.HashSet;
7
8  /**
9   * A simple forward analysis example, computing reaching definitions.
10  *
11  * @author Jesse Doherty
12  */
13 public class ReachingDefs
14    extends
15       AbstractSimpleStructuralForwardAnalysis<HashMapFlowMap<String,
16                                                Set<AssignStmt>>>
17 {
18    private Merger merger = new Merger<Set<ASTNode>>(){
19       public Set<ASTNode> merge( Set<ASTNode> s1, Set<ASTNode> s2 )
20       {
21          Set<ASTNode> ms = new HashSet<ASTNode>( s1 );
22          ms.addAll( s2 );
23          return ms;
24       }
```

```
25    };
26
27    private HashMapFlowMap<String,Set<AssignStmt>> startMap;
28    private NameCollector nameCollector;
29
30    public ReachingDefs( ASTNode tree )
31    {
32       super(tree);
33       startMap = new HashMapFlowMap<String,Set<AssignStmt>>(merger);
34       nameCollector= new NameCollector(tree);
35       nameCollector.analyze();
36       for( String var : nameCollector.getAllNames() )
37          startMap.put( var, new HashSet<AssignStmt>() );
38    }
39
40    /**
41     * Defines the merge operation for this analysis.
42     *
43     * This implementation uses the union method defined by {@link
44     * AbstrcatFlowMap}. Note that the union method deals with the
45     * cases where {@literal in1==in2}, {@literal in1==out} or
46     * {@literal in2==out}.
47     */
48    public void merge( HashMapFlowMap<String,Set<AssignStmt>> in1,
49                  HashMapFlowMap<String,Set<AssignStmt>> in2,
50                  HashMapFlowMap<String,Set<AssignStmt>> out )
51    {
52       in1.union( merger, in2, out );
53    }
54
55    /**
56     * Creates a copy of the FlowMap with copies of the contained set.
57     */
58    public void copy( HashMapFlowMap<String,Set<AssignStmt>> in,
59                  HashMapFlowMap<String,Set<AssignStmt>> out )
60    {
61       if( in == out )
```

```
62          return;
63       out.clear();
64       for( String i : in.keySet() )
65          out.put(i, new HashSet<AssignStmt>(in.get(i)));
66    }
67
68    /**
69     * Creates a copy of the given flow-map and returns it.
70     */
71    public HashMapFlowMap<String,Set<AssignStmt>>
72       copy( HashMapFlowMap<String,Set<AssignStmt>> in )
73    {
74       HashMapFlowMap<String,Set<AssignStmt>> out =
75          new HashMapFlowMap<String,Set<AssignStmt>>();
76       copy(in, out);
77       return out;
78    }
79
80    public HashMapFlowMap<String,Set<AssignStmt>> newInitialFlow()
81    {
82       return copy(startMap);
83    }
84
85    public void caseAssignStmt( AssignStmt node )
86    {
87       inFlowSets.put( node, currentInSet );
88       currentOutSet = copy(currentInSet);
89       Set<String> defVars = nameCollector.getNames( node );
90
91       for( String n : defVars ){
92          Set<AssignStmt> newDefSite = new HashSet<AssignStmt>();
93          newDefSite.add( node );
94          currentOutSet.put( n, newDefSite );
95       }
96       outFlowSets.put(node, currentOutSet);
97    }
98
```

```
99      public void caseStmt( Stmt node )
100     {
101        inFlowSets.put( node, currentInSet );
102        currentOutSet = currentInSet;
103        outFlowSets.put( node, currentOutSet );
104     }
105
106  }
```

Listing A.1 `ReachingDefs` analysis code

Appendix B
Variable Use Collector Code

```
1  package natlab.toolkits.analysis.test;
2
3  import java.util.*;
4  import ast.*;
5  import analysis.*;
6  import natlab.toolkits.analysis.varorfun.*;
7
8  /**
9   * @author Jesse Doherty
10  */
11 public class UseCollector
12    extends AbstractDepthFirstAnalysis<HashSetFlowSet<String>>
13 {
14    private VFPreorderAnalysis kindAnalysis;
15
16    private HashSetFlowSet<String> fullSet;
17    private boolean inLHS = false;
18
19    public UseCollector(ASTNode tree)
20    {
21       super(tree);
22       fullSet = new HashSetFlowSet<String>();
23       kindAnalysis = new VFPreorderAnalysis(tree);
24       kindAnalysis.analyze();
```

```
25      }
26
27      public HashSetFlowSet<String> newInitialFlow()
28      {
29         return new HashSetFlowSet<String>();
30      }
31
32      /**
33       * Gets a set of all uses for the given tree.
34       */
35      public Set<String> getAllUses()
36      {
37         return fullSet.getSet();
38      }
39      /**
40       * Gets a set of uses in the given statement.
41       */
42      public Set<String> getUses( Stmt node )
43      {
44         HashSetFlowSet<String> set = flowSets.get(node);
45         if( set == null )
46            return new HashSet<String>();
47         else
48            return set.getSet();
49      }
50
51      /**
52       * Finds all uses in the given assignment. It makes sure that the
53       * target of the assignment isn't considered a use.
54       */
55      public void caseAssignStmt( AssignStmt node )
56      {
57         HashSetFlowSet<String> prevSet = currentSet;
58         inLHS = true;
59         currentSet = newInitialFlow();
60
61         analyze(node.getLHS() );
```

```
62          inLHS = false;
63          analyze( node.getRHS() );
64
65          flowSets.put(node, currentSet);
66          fullSet.addAll( currentSet );
67
68          if( prevSet != null )
69             prevSet.addAll( currentSet );
70          currentSet = prevSet;
71       }
72
73       /**
74        * Finds all uses in the given statement.
75        */
76       public void caseStmt( AssignStmt node )
77       {
78          HashSetFlowSet<String> prevSet = currentSet;
79          currentSet = newInitialFlow();
80
81          caseASTNode( node );
82
83          flowSets.put(node, currentSet);
84          fullSet.addAll( currentSet );
85
86          if( prevSet != null )
87             prevSet.addAll( currentSet );
88          currentSet = prevSet;
89       }
90
91       /**
92        * Makes sure that targets of assignments aren't considered
93        * uses. Also makes sure that the arguments are seen.
94        */
95       public void caseParameterizedExpr( ParameterizedExpr node )
96       {
97          analyzeAsNotLHS( node.getArgs() );
98          analyze( node.getTarget() );
```

```
99    }
100
101   //NOT: More cases would be needed to make complete.
102
103   /**
104    * Checks if the name is possible a variable, and not the target
105    * of an assignment; if it is, adds it.
106    */
107   public void caseNameExpr( NameExpr node )
108   {
109      if( !inLHS ){
110         if( maybeVar( node ) )
111            currentSet.add(node.getName().getID());
112      }
113   }
114
115   /**
116    * Helper method to analyze a given node, making sure it is
117    * treated like it isn't the target of an assignment. It saves and
118    * restores the state of {@code inLHS}
119    */
120   private void analyzeAsNotLHS( ASTNode node )
121   {
122      boolean bakInLHS = inLHS;
123      inLHS = false;
124      analyze( node );
125      inLHS = bakInLHS;
126   }
127
128   /**
129    * A helper method to abstract away the test to see if an name
130    * expression might be a variable.
131    */
132   public boolean maybeVar( Expr expr )
133   {
134      if( expr instanceof NameExpr ){
135         NameExpr nameExpr = (NameExpr)expr;
```

```
136          if( nameExpr.tmpVar )
137             return true;
138          else{
139             Name name = nameExpr.getName();
140             if (kindAnalysis.getFlowSets().containsKey(name)){
141                kindAnalysis.analyze();
142             }
143             VFDatum kind =
144                kindAnalysis.getFlowSets().get(name).contains(
145                                  nameExpr.getName().getID()
146                                                        );
147             return (kind!=null) && (kind.isVariable() || kind.isID());
148          }
149       }
150       return false;
151    }
152 }
```

Listing B.1 Variable use collector code

Appendix C

Full Maybe Live Variable Analysis Code

```
1  package natlab.toolkits.analysis.test;
2
3  import analysis.*;
4  import natlab.toolkits.analysis.varorfun.*;
5
6  import ast.*;
7
8
9  import java.util.Set;
10 import java.util.HashSet;
11
12 /**
13  * Performs a naive Live Variable analysis. It ignores the possibility
14  * of variables being used by function calls, script calls, and
15  * evals. Basically it ignores dynamic behaviour and the lack of
16  * scope.
17  *
18  * @author Jesse Doherty
19  */
20 public class MaybeLive
21    extends
22       AbstractSimpleStructuralBackwardAnalysis<HashSetFlowSet<String>>
23 {
24
```

```
25    private NameCollector nameCollector;
26    private UseCollector useCollector;
27
28
29    public MaybeLive( ASTNode tree)
30    {
31       super(tree);
32
33       nameCollector = new NameCollector(tree);
34       nameCollector.analyze();
35       useCollector = new UseCollector(tree);
36       useCollector.analyze();
37    }
38
39    /**
40     * Merges the two sets by using the union defined by {@link
41     * HashSetFlowSet}.
42     */
43    public void merge( HashSetFlowSet<String> in1,
44                  HashSetFlowSet<String> in2,
45                  HashSetFlowSet<String> out )
46    {
47       in1.union( in2, out );
48    }
49
50    /**
51     * Copies {@code in} into {@code out} by using {@code in}'s {@code
52     * copy(...)} method.
53     */
54    public void copy( HashSetFlowSet<String> in,
55                  HashSetFlowSet<String> out )
56    {
57       in.copy(out);
58    }
59
60    /**
61     * Returns a copy of {@code in} by using it's {@code copy()}
```

```
62      * method.
63      */
64     public HashSetFlowSet<String> copy( HashSetFlowSet<String> in)
65     {
66        return in.copy();
67     }
68
69     /**
70      * The initial flow is an empty set. Initially, no variables are
71      * live.
72      */
73     public HashSetFlowSet<String> newInitialFlow()
74     {
75        return new HashSetFlowSet<String>();
76     }
77
78     /**
79      * Creates the in-flow for an assignment statement. It uses the
80      * {@link NameCollector} and {@link UseCollector} to find the
81      * variable names to remove and add, respectively. It associates
82      * the out and resulting in to the given node.
83      */
84     public void caseAssignStmt( AssignStmt node )
85     {
86        outFlowSets.put( node, currentOutSet );
87        //HashSetFlowSet<String> workingInFlow = copy( currentOutSet
88        //);
89        currentInSet = copy( currentOutSet );
90
91        Set<String> defVars = nameCollector.getNames( node );
92        Set<String> useVars = useCollector.getUses( node );
93
94        for( String def : defVars )
95           currentInSet.remove( def );
96        for( String use : useVars )
97           currentInSet.add( use );
98
```

```
99          inFlowSets.put( node, currentInSet );
100
101      }
102
103      /**
104       * Creates the in-flow for an arbitrary statement. Uses the {@link
105       * UseCollector} to find names to add to the flow.
106       */
107      public void caseStmt( Stmt node )
108      {
109          outFlowSets.put( node, currentOutSet );
110          HashSetFlowSet myInSet = copy(currentOutSet);
111
112          caseAST( node );
113
114          Set<String> useVars = useCollector.getUses( node );
115
116          for( String use : useVars )
117              myInSet.add( use );
118
119          currentInSet = myInSet;
120          inFlowSets.put( node, currentInSet );
121      }
122
123  }
```

Listing C.1 `MaybeLive` analysis code

Bibliography

[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.

[CBHV10] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through just-in-time specialization. In *International Conference on Compiler Construction*, March 2010, pages 46–65.

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

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

[EH07a] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, Montreal, Quebec, Canada, 2007, pages 1–18. ACM, New York, NY, USA.

[EH07b] Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69:14–26, December 2007.

[JB02] Pramod G. Joisha and Prithviraj Banerjee. Magica: A software tool for inferring types in MATLAB. Technical report, Department of Electrical and Computer Engineering, Northwestern University, Oct 2002.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[Li09] Jun Li. McFOR: A MATLAB to FORTRAN 95 compiler. Master's thesis, August 2009.

[Mata] Matlab. Official matlab documentation. Home page `http://www.mathworks.com/help/techdoc/`.

[Matb] Matlab. The Language Of Technical Computing. Home page `http://www.mathworks.com/products/matlab/`.

[MP99] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *In Proceedings of the Second Conference on Domain-Specific Languages*, 1999, pages 53–65.

[NNHME09] Emma Nilsson-Nyman, Görel Hedin, Eva Magnusson, and Torbjörn Ekman. Declarative intraprocedural flow analysis of Java source code. *Electron. Notes Theor. Comput. Sci.*, 238:155–171, October 2009.

[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.

[TAH10] Anton Dubrau Toheed Aslam, Jesse Doherty and Laurie Hendren. Aspectmatlab: An aspect-oriented scientific programming language. In *AOSD '10: Proceedings of the 9th international conference on Aspect-oriented software development*, Rennes and St. Malo, France, 2010, pages 181–192. ACM, New York, NY, USA.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, 1999, pages 125–135.