UNDERSTANDING AND REFACTORING THE MATLAB LANGUAGE

by

Soroush Radpour

School of Computer Science
McGill University, Montréal

August 2012

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 very popular dynamic "scripting" language for numerical computations used by scientists, engineers and students world-wide. MATLAB programs are often developed incrementally using a mixture of MATLAB scripts and functions and frequently build upon existing code which may use outdated features. This results in programs that could benefit from refactoring, especially if the code will be reused and/or distributed. Despite the need for refactoring there appear to be no MATLAB refactoring tools available. Correct refactoring of MATLAB is quite challenging because of its non-standard rules for binding identifiers. Even simple refactorings are non-trivial. Compiler writers and software engineers are generally not familiar with MATLAB and how it's used so the problem has been left untouched so far.

This thesis has two main contributions. The first is McBench, a tool that helps compiler writers understand the language better. In order to have a systematic approach to the problem, we developed this tool to give us some insight about how programmers use MATLAB. The second contribution is a suite of semantic-preserving refactoring for MATLAB functions and scripts including: function and script inlining, converting scripts to functions, extracting new functions, and converting dynamic *feval* calls to static function calls. These refactorings have been implemented using the *Mc***LAB** compiler framework, and an evaluation is given on a large set of MATLAB programs which demonstrates the effectiveness of our approach.

Résumé

MATLAB est un « langage de script » dynamique utilisé à des fins de calcul numérique par des scientifiques, ingénieurs et étudiants du monde entier. Les programmes MATLAB sont souvent développés selon une méthode incrémentale, sur la base d'un mélange de scripts et fonctions MATLAB, et sont habituellement conçus à partir d'un code existant dont les fonctionnalités seraient obsolètes. Par conséquent, certains programmes pourraient bénéficier de réusinage, surtout si le code sera réutilisé et/ou distribué. Malgré ce besoin, il n'existe aucun outil MATLAB de ce genre. Le réusinage de MATLAB est assez difficile car les règles pour la liaison des identificateurs ne sont pas standards. Même une opération de maintenance simple revêt une certaine complexité. De plus, les créateurs de compilateurs et les ingénieurs en informatique ne sont généralement pas familiers avec MATLAB et la façon dont il est utilisé. C'est pourquoi à ce jour le problème n'a jamais été traité.

Cette thèse apporte deux contributions principales : d'une part la création de MCBENCH, un outil aidant les créateurs de compilateurs à mieux comprendre le langage. Afin d'avoir une approche systématique du problème, nous avons développé cet outil pour en savoir plus sur la façon dont les programmeurs utilisent MATLAB. L'autre contribution est une suite de réusinages préservant la sémantique des fonctions et scripts MATLAB : incorporation de fonctions et scripts, conversion de scripts en fonctions, extraction de nouvelles fonctions et conversion d'appels dynamiques *feval* en appels de fonction statique. Le cadriciel et compilateur *Mc*LAB a été utilisé pour la mise en œuvre de ces réusinages. De plus, une évaluation est donnée sur un large éventail de programmes MATLAB afin de démontrer l'efficacité de notre approche.

Acknowledgements

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

I would like to thank my advisor, Laurie Hendren, for providing me with the great opportunity to participate in the *Mc*LAB project and supporting me. She taught me much about research, compilers, writing and advising.

I would like to thank all the members of *Mc*LAB project for creating such a great framework. I appriciate the support and patience from my colleagues at *Mc*LAB as I finished my thesis concurrently with my other responsibilities.

Finally, I would like thank my parents and my brother for encouraging me and believing in me.

Table of Contents

Appendices

List of Figures

List of Tables

List of Listings

Chapter 1
Introduction

Refactoring may be defined as the process of applying a set of behavior-preserving transformations in order to change the structure of a program. The goal can be to improve readability, maintainability, performance or to reduce the complexity of code. Refactoring has developed for the last 20 years, starting with the seminal theses by Opdyke [Opd92] and Griswold [Gri91], and the well known book by Fowler [Fow99]. Many programmers have come to expect refactoring support and popular IDEs such as Eclipse, Microsoft's Visual Studio, Sun's NetBeans have integrated support for automated refactorings. However, the benefits of refactoring tools have not yet reached the millions of MATLAB programmers. Currently neither the proprietary Mathworks' MATLAB IDE, nor open-source tools provide refactoring support.

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

To study MATLAB programs, we built McBench query framework and gathered a large body of MATLAB programs. In our studies we have found that the code could benefit from

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

refactoring for several reasons. First, the MATLAB language has evolved over the years, incrementally introducing many valuable high-level features such as functions, nested functions, packages and so on. However, MATLAB programmers often build upon code available online or code found from books and frequently that code does not use the modern high-level features. Thus, although code reuse has been an essential part of the MATLAB eco-system, obsolete syntax and new language features complicates this reuse. Since MATLAB does not currently have refactoring tools, programmers either do not refactor, or they refactor code by hand which is time-consuming and error-prone. Secondly, the interactive nature of developing MATLAB programs promotes a style of programming in which the organization of functions and scripts is relatively unstructured and not modular. When developing small one-off scripts this may not be important, but when developing a complete application or library, refactoring the code to be better structured and more modular is key for reuse and maintenance.

Although desirable, developing correct and automatic refactorings for MATLAB is actually quite challenging. In particular, to ensure behavior-preserving refactorings, it is important to verify that identifiers maintain their correct kind (variable or function) and in the case of functions, identifiers must resolve to the correct function after refactoring. Furthermore, there are some MATLAB features that are undesirable. For example, MATLAB scripts are a hybrid of macros and functions and can lead to unstructured code which is hard to analyze and optimize. Dynamic features like `feval` which evaluates a string as though it were a MATLAB expression, also complicate programs and are often used inappropriately. Thus, MATLAB-specific refactorings, which eliminate these features, are also very useful.

1.1 Contributions

In this thesis we present the refactorings that we developed for MATLAB, as well as the reusable tools we built in our development process. Our first contribution is our query framework. We built the McBench query framework in order to help us understand MATLAB programs and identify refactoring opportunities. Our tool allows compiler writers to search for specific syntactic and semantic features in a large body of MATLAB code that we have collected. We also implemented various features to make it easier to read, understand and browse through the MATLAB source code. To make McBench reusable for other compiler researchers, we implemented an annotation framework that makes it possible to expand the information available for querying.

Our second contribution is an extension to the *Mc*LAB static analysis framework. Our extension can be used in data-flow analyses of functions to significantly improve the precision of the results in presence of script calls. A fixed-point analysis driver is implemented which builds a call graph from script calls and propagates summary information of scripts inter-procedurally through this graph. We also implemented Liveness and Reaching Definition Analysis using this extension.

Our final contribution is our refactoring tool. After studying MATLAB code we found that scripts are commonly used in MATLAB and can benefit from refactoring. To tackle this problem we have implemented *Convert Script to Function* and *Inline Script*. We also implemented the *Extract Function* refactoring which is one of the most commonly used refactorings [Fow99]. *Inline Function* refactoring was implemented to complement *Inline Script* so that our tool can inline both functions and scripts. The replace `feval` refactoring is an example of how a simple refactoring can get rid of nasty features.

1.2 Outline

This thesis is split into 6 chapters (including this introductory chapter). *Chapter 2* gives the necessary background information about the MATLAB language. This also includes an introduction of Kind Analysis and the MATLAB lookup semantics. *Chapter 3* presents the McBench framework for searching through a large body of MATLAB code. We also show some examples of how this framework can be used to understand MATLAB programs. *Chapter 4* introduces our extension to the *Mc*LAB static analysis framework that enables correct and more precise handling of script calls during flow analyses with minimal effort. This chapter also presents our implementation of Liveness Analysis, Reaching Definition Analysis and the Return Elimination algorithm that are used in our refactorings. In *Chapter 5* we present Inlining Scripts and Functions, Converting Scripts to Functions, Extract Function and Replace `feval` refactorings that we have implemented for MATLAB. Finally, *Chapter 7* presents our conclusions and future work.

Chapter 2
Background

MATLAB programs consist of a collection of functions and scripts. *Listing 2.1* illustrates a typical function called `MultiplyCompatible`. This function takes as input two arrays, `n` and `m`, and returns `true` if they are both 2-dimensional arrays and the the number columns of `n` is the same as the number of rows of `m`. [1]

```
function r = MultiplyCompatible(n, m)
  ndims = size(n);
  mdims = size(m);
  r = ((length(ndims)==2) && ...
    (length(mdims)==2) && ...
    (ndims(2)==mdims(1)));
end

% Kinds ...
% VAR: r,n,m,ndims,mdims,r
% FN: size, length
```

Listing 2.1 Function stored in `MultiplyCompatible.m`

MATLAB programs consist of a collection of functions and scripts. In general, MATLAB functions have input parameters and may also have output parameters. Parameters obey call-by-value semantics where semantically a copy of each input and output parameter is

[1] Note that we have put the Kind of each identifier in comments at the end of each function/script definition. This is just to help us explain Kind analysis later in this chapter.

made. MATLAB does not explicitly declare local variables, nor explicitly declare the types of any variables. Input and output parameters are explicitly declared as variables, whereas other variables are implicitly declared upon their first definition. For example, statements 2 and 3 define the variables `ndims` and `mdims`. Variables defined within a function body are local to the function unless they are explicitly declared to be global or persistent.

The globals structure maps global variable names to values. There is one such globals structure shared by all functions. A variable "x" is local within a function until a call to "global(x)" occurs within the function body. Currently it is possible for a function body to contain both local and global uses of "x". However, the current version of MATLAB issues warnings that future versions will not allow this. Presumably this means that in future versions, a call to "global(x)" will have to dominate all other occurrences of "x" within the function body.

The persistents structure maps (fully qualified function name × variable name) to values. Persistent variables are like global variables, but are associated with a specific named function only. Within function "f", a variable "x" is made persistent through a call "persistent(x)". Calls to persistent may only occur in function bodies (and not scripts) and a call to "persistent(x)" must dominate all other occurrences of "x" in the function body.

It is important to note that it is not possible to syntactically distinguish between references to variables and calls to functions. For example, `size(n)` on line 2 is a call to a function, whereas `ndims(2)` on line 6 is a reference to a variable, even though they use the same syntactic structure. This lack of syntactic distinction between variables and functions leads to complications that must be correctly handled by refactorings, as illustrated in *Section 2.2*.

MATLAB scripts are even more unstructured than functions. Scripts are simply a sequence of statements that can be invoked. For example, consider the script in *Listing 2.2*, which looks similar to the body of the function in *Listing 2.1*.

A script is executed in the workspace from which it was called, either the main workspace, or the workspace of the last-called function. In MATLAB, workspaces store the values of variables. There is an initial "main" workspace which is acted upon by commands entered into the main read-eval-print loop. There is a also a stack of workspaces corresponding the the function call stack. A call to a function creates and pushes a new workspace, which

```
1 ndims = size(n); % ndims has Kind VAR
2 mdims = size(m);
3 isCompatible = ((length(ndims)==2) && ...
4   (length(mdims)==2) && ...
5   (ndims(2)==mdims(1)));
6
7 % Kinds ...
8 % VAR: ndims,mdims, isCompatible
9 % ID: size, length
```

Listing 2.2 Script stored in `SMultiplyCompatible.m`

becomes the current workspace. For example, if `SMultiplyCompatible` is invoked from a workspace which contains a variable `size`, then lines 1 and 2 of *Listing 2.2*, would refer to elements of that variable. If the invoking workspace does not contain a variable called `size`, then lines 1 and 2 refer to a call to the built-in function `size`. Furthermore, if the script defines new variables, those will be put in the workspace of the caller. Clearly scripts are not very modular, and thus developing refactorings to eliminate them by inlining or converting scripts to functions is beneficial.

2.1 MATLAB programs

MATLAB programs are defined as directories of files. Each file of the form `f.m` contains either: (a) a script, which is simply a sequence of MATLAB statements; or (b) a sequence of function definitions. If the file `f.m` defines functions, then the first function defined in the file should be called `f` (although even if it is not called `f` it is known by that name in MATLAB). The first function is known as the *primary function*. Subsequent functions are *subfunctions*. The primary and subfunctions within `f.m` are visible to each other, but only the primary function is visible to functions defined in other `.m` files. Functions may be nested, following the usual static scoping semantics of nested functions. That is, given some nested function `f′`, all enclosing functions, and all functions declared in the same nested scope are visible within the body of `f′`.

MATLAB directories may contain special private, package and type-specialized directories, which are distinguished by the name of the directory. Private directories must be named `private/`, Package directories start with a '+', for example `+mypkg/`. The primary function in each file `f.m` defined inside a package directory `+p` corresponds to a function named `p.f`. To refer to this function one must use the fully qualified name, or an equivalent import declaration. Package directories may be nested. Type-specialized directories have names of the form `@<typename>`, for example `@int32/`. The primary function in a file `f.m` contained in a directory `@typename/` matches calls to `f(a1,...)`, where the run-time type of the primary (first) argument is `typename`.

2.2 Kind Information

Since MATLAB does not syntactically distinguish between variables and functions, modern implementations of MATLAB have added a static analysis which determines the Kind of each identifier at compile-time. In this chapter, we have indicated the results of the Kind analysis as comments at the end of each function/script definition.

Kind Analysis[DHR11] assigns one of the following Kinds to each identifier:

VAR**:** The identifier must be looked up as a variable in a workspace;

FN**:** The identifier must be looked up as a named function;

ID**:** The Kind is not known, so at runtime the identifier must first be looked up in the workspace and then if not found, it will be looked up as a function.

PREFIX**:** The identifier refers to a package, as the prefix of a fully-qualified function name. For example in the expression `mypkg.f`, `mypkg` would have the Kind PREFIX.

It is a compile-time error if an identifier has conflicting Kinds (one occurrence is a VAR and the other is a FN).

This static Kind assignment is now an integral part of the semantics of MATLAB, and refactorings must ensure that the meaning of identifiers is maintained and that the refactoring will not introduce any new Kind errors.

One twist in the Kind Analysis semantics that affects the refactorings is that the semantics are different for scripts and functions. Scripts in MATLAB are simply a sequence of statements that might be executed in different contexts. Since the set of variables in those contexts are not known during compile time, the initial assumption is that every identifier

```
function r = MultiplyFn(a, b)
  if (ndims(a)==3 && ndims(b)==3) % ndims has Kind FN
    r = Do3DMult(a,b);
  else
    n = a; m = b;
    SMultiplyCompatible;
    if (isCompatible)
      r = a * b;
    else
      error(’Matrix Dimension Error’);
    end
  end
end

% Kinds ...
% VAR: r, a, b, n, m
% FN: ndims, Do3DMult, SMultiplyCompatible
% ID: isCompatiblefunction
```

Listing 2.3 A function calling a script

needs a runtime lookup. There are two important implications of the difference between the Kind analysis. Firstly, the Kind information for scripts are much less precise. This negatively impacts our ability to analyze scripts. Secondly, any refactoring that moves code between scripts and functions has to verify that the change in Kind information doesn’t change the program behavior. Let us consider the example in *Listing 2.3*.

This function first checks to see if the number of dimensions of `a` and `b` are 3, and if so, calls a general multiplication function, otherwise it continues to check for the ordinary 2-D case. If we were to inline the call to the script `SMutiplyCompatible` (as given in *Listing 2.2*) care must be taken with the identifier `ndims`. In `MultiplyFn` the identifier `ndims` refers to a function and will have Kind FN, whereas in `SMultiplyCompatible` `ndims` is assigned to, and will have Kind VAR.

If we inlined without appropriately renaming `ndims`, as shown in *Listing 2.4*, we would introduce a Kind error because the inlined source would use `ndims` in a conflicting manner. Thus, at JIT compile-time a conflicting Kind error would be triggered on line 8.

```
function r = MultiplyFn(a, b)
  if (ndims(a)==3 && ndims(b)==3) % ndims has Kind FN
    r = Do3DMult(a,b);
  else
    n = a; m = b;

   % --- begin inlined script SMultiplyCompatible
   ndims = size(n); % ndims has Kind VAR - Kind error
   mdims = size(m);
   isCompatible = ((length(ndims)==2) && ...
      (length(mdims)==2) && ...
      (ndims(2)==mdims(1)));
   % --- end of inlined script SMultiplyCompatible

    if (isCompatible)
      r = a * b;
    else
      error('Matrix Dimension Error');
    end
  end
end

% Kinds ...
% VAR: r, a, b, n, m, isCompatible
% FN: Do3DMult, size, length
% ERROR: ndims
```

Listing 2.4 Example of Kind error due to script inlining

2.3 Impact of Function and Script Lookup on Refactoring

In MATLAB the lookup of a script/function is performed relative to: *f*, the current function/script being executed; *sourcefile*, the file in which *f* is defined; *fdir*, the directory containing the last called non-private function (calling scripts or private functions does not change *fdir*); *dir*, the current directory; and *path*, a list of other directories. When looking up function/script names, first *f* is searched for a nested function, then *sourcefile* is searched for a subfunction, then the private directory of *fdir* is searched, then *dir* is searched, followed by the directories on *path*.

In the case where there is both a non-specialized and type-specialized function matching a call, the non-specialized version will be selected if it is defined as a nested, subfunction or private function, otherwise the specialized function takes precedence.

Obviously if a piece of a program is moved from one directory to another, one must ensure that the function lookup remains the same. A simple example of a lookup problem would be if the function `MultiplyCompatible` was inlined into a a function which had a `private/` directory which included a new definition of the function `size`. The inlined version would now call the `private/size.m` function instead of the standard library function.

A further complicating factor for MATLAB is that some of the arguments to the lookup function use dynamic values. These are: *fdir* (changes each time a function is called), *dir* (can be changed by the *cd* function) and *path* (can be dynamically set in the program). The fact that the function lookup relies on some dynamic information means that a static refactoring must use a static approximation to estimate the function lookup results.

Our examination of a large set of benchmarks showed that the current directory and path do not normally change during the execution of the program. This is particularly true if the application has been written in a way that makes it somewhat portable.

Chapter 3
McBench: A Tool for Understanding MATLAB Programs

Having a good understanding of typical uses of a language helps compiler writers to identify opportunities and prioritize the improvements based on their relevance. In this chapter we talk about our approach in understanding how MATLAB programmers code. When writing a compiler optimization, it is helpful to know how often the particular pattern occurs and what are the contexts where the pattern occurs in. It can help to identify common coding patterns that can be refactored to improve code quality.

A compiler writer might want to know if a language builtin function is used frequently. Do the calls occur in loops and is the function worth optimizing? The question might be about language features. Do programmers use persistent variables? Or how do MATLAB programmers use exceptions? How are copy statements used in MATLAB? Semantic questions also come up frequently. Are there cases where the type of a variable is different on different branches?

Let's take a simple question like how often the MATLAB builtin function `i` is called. In many languages a simple regular expression that looks for pattern "`i(`" might give a good estimate. Unfortunately this is not the case in MATLAB. A call to function `i` looks exactly the same as an access to a variable with the same name. Although some of these questions can be answered using regular expressions, as we saw these lexical tools are very limited. Questions about nesting and contexts are even more difficult to answer using these tools, if not impossible.

In the following sections we describe McBench, our framework to let the user search through a large set of projects for specific patterns and see exactly where these patterns occur. *Section 3.1* talks about the different components of the framework. Then we briefly describe the MATLAB parse tree and its eXtensible Markup Language (XML) representation as produced by *Mc*LAB parser (*Section 3.2*) and how to add annotations (*Section 3.3*). Then we talk about the XPath query language (*Section 3.4*) and we show some examples of how McBench can be used to gather statistics about how MATLAB is used (*Section 3.5*).

3.1 McBench

In order to provide faster search through the code database, we decided to pre-process the source collection and run the queries on the processed data. Pre-processing starts with parsing the source collection from MATLAB to the *Mc*LAB abstract syntax tree (AST). Then the AST is converted to XML and during this process the XML output is annotated with semantic information for future queries.

We chose the XPath language [BBC$^+$07] to search through these XML documents. A set of macros were added to the language to make common queries easier to write. *Section 3.4* presents the language and our additions in more detail.

The framework consists of a web interface to add benchmarks, run queries and view the results. We also created two implementations of the query engine:

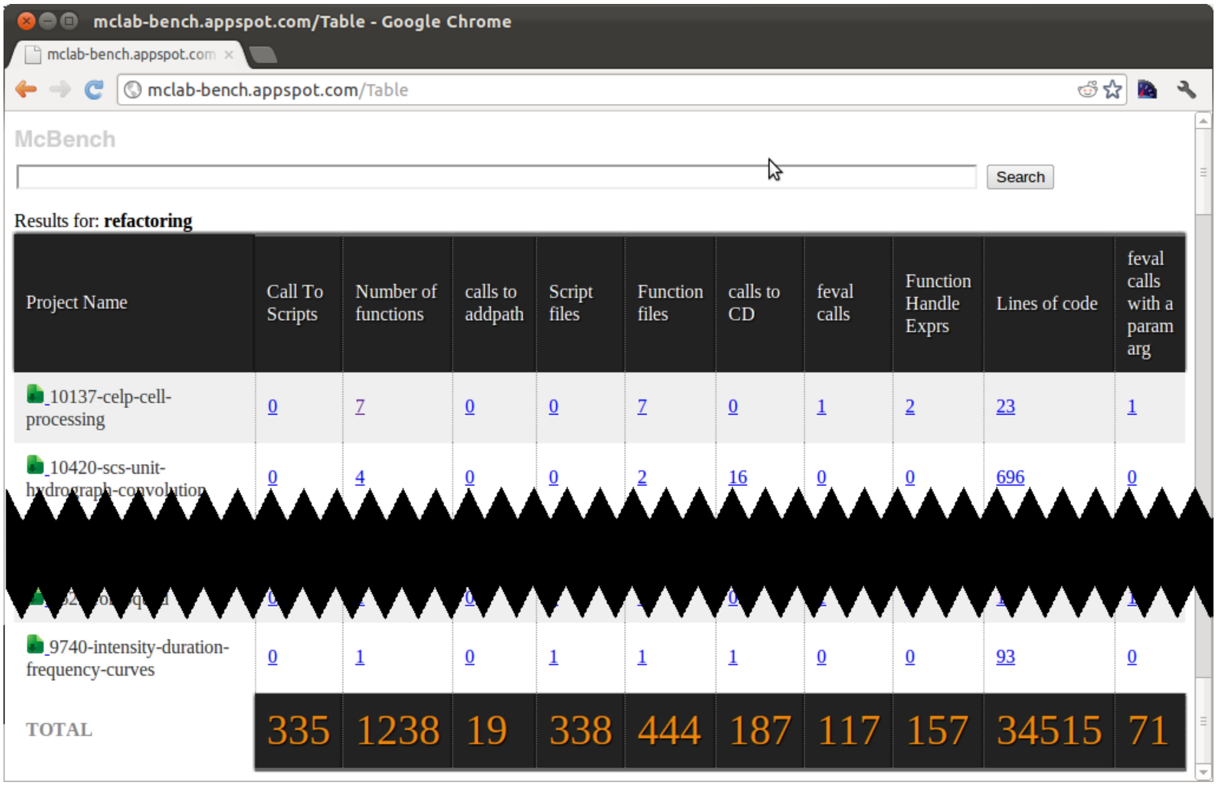
McBench Cloud: This is a scalable implementation of the engine using Google AppEngine technology and the MapReduce API [1]. A query is executed on each project as a separate task that can run in parallel.

McBench Django: This implementation runs on a single computer and uses the process pool design pattern to run the query on all of the available CPU cores.

We also created a web-based user interface to interact with the software. It provides the ability to run custom queries and predefined query-sets. *Figure 3.1* shows the McBench results summary page for the query-set that we used before we implement our refactorings.

[1] Available on `http://mclab-bench.appspot.com`

Each row represents one project and for each project the summary page lists the number of occurrences for each query. This number is linked to a detailed view that lists the occurences of the searched pattern per file. So if we click on the number in the "calls to addpath" column for the project "12137-pid-state-feedback-control-of-dc-motors" it will bring us to the page shown in *Figure 3.2*. The last row is the sum of the results for each column.


mclab-bench.appspot.com/Table - Google Chrome
mclab-bench.appspot.com
mclab-bench.appspot.com/Table
McBench
Search

Results for: **refactoring**

Project Name	Call To Scripts	Number of functions	calls to addpath	Script files	Function files	calls to CD	feval calls	Function Handle Exprs	Lines of code	feval calls with a param arg
10137-celp-cell-processing	0	7	0	0	7	0	1	2	23	1
10420-scs-unit-hydrograph-convolution	0	4	0	0	2	16	0	0	696	0
9740-intensity-duration-frequency-curves	0	1	0	1	1	1	0	0	93	0
TOTAL	335	1238	19	338	444	187	117	157	34515	71

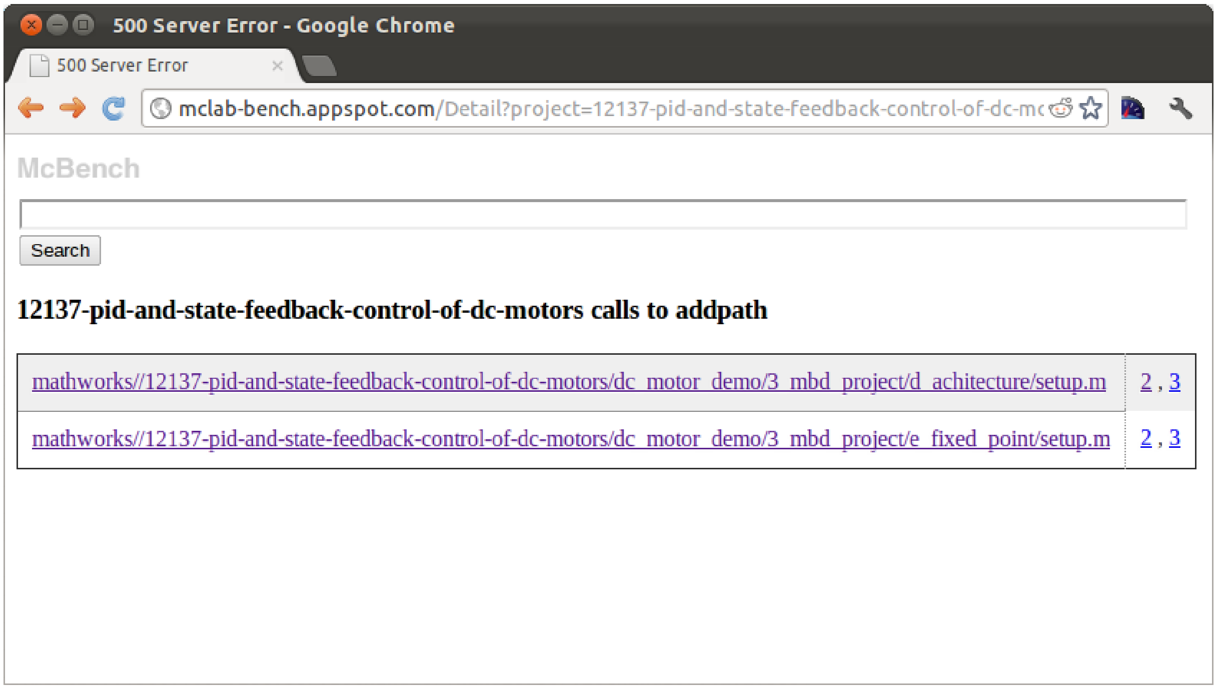
Figure 3.1 McBench result overview page for the refactoring query-set.

The detailed view lists all the files in the project that have matches to our query along with the line numbers where the match is found. The file names are linked to the source-code viewer that shows MATLAB source files with the search results highlighted. For example if we want to view the calls to `addpath` in the project selected in *Figure 3.2*, we can click on one of the listed files. *Figure 3.3* shows the page that was accessed by clicking on the ".../e_fixed_point/setup.m" link. Apart from showing the source-code, the code viewer provides several features to ease browsing:

Syntax highlighting: Different terms are displayed using different colors and fonts according to their category which improves the readability.

Search: Clicking on an identifier brings all other mentions of the identifier in the same project.

Line numbers: Lines numbers allow users to refer to a specific part of a program easily



500 Server Error - Google Chrome

500 Server Error

mclab-bench.appspot.com/Detail?project=12137-pid-and-state-feedback-control-of-dc-mc

McBench

Search

12137-pid-and-state-feedback-control-of-dc-motors calls to addpath

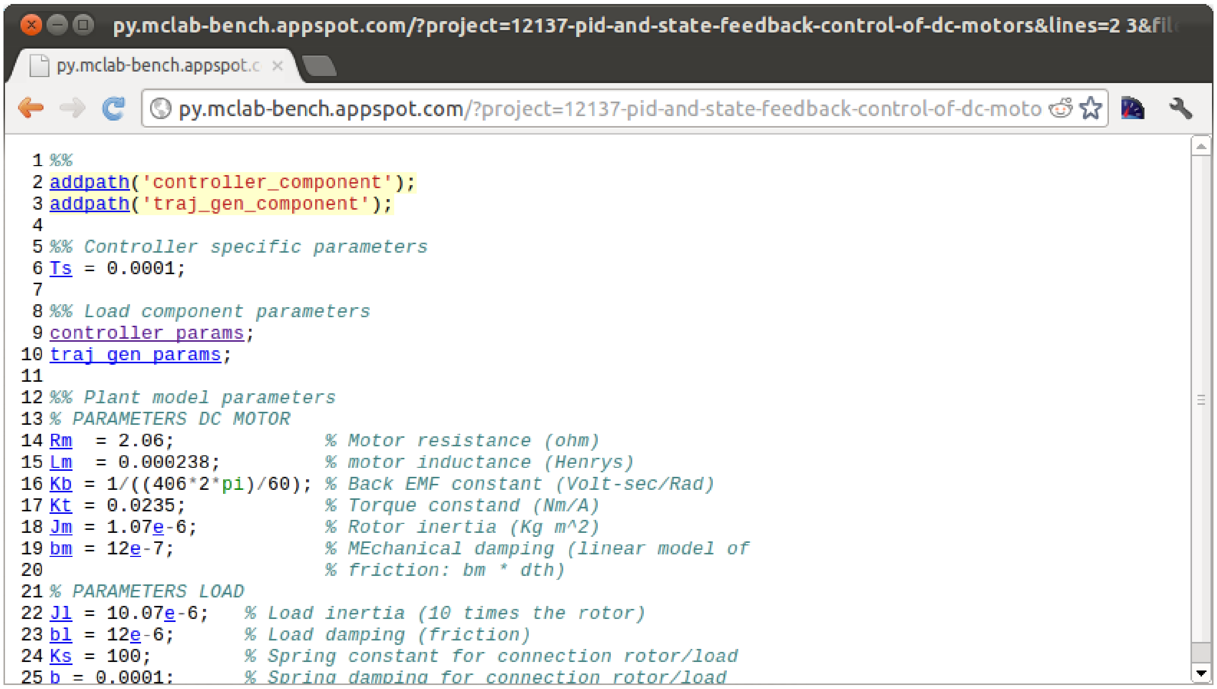
mathworks//12137-pid-and-state-feedback-control-of-dc-motors/dc_motor_demo/3_mbd_project/d_achitecture/setup.m	2 , 3
mathworks//12137-pid-and-state-feedback-control-of-dc-motors/dc_motor_demo/3_mbd_project/e_fixed_point/setup.m	2 , 3

Figure 3.2 McBench detailed result view for "calls to addpath" query.

Figure 3.4 shows an overview of the architecture and components of McBench framework.

3.2 McAST XML Structure

In order to understand and write queries for McBench we need to know what the XML output looks like. The XML closely replicates the AST from which it is created with small differences and addition of annotations.


py.mclab-bench.appspot.com/?project=12137-pid-and-state-feedback-control-of-dc-motors&lines=2 3&fil
py.mclab-bench.appspot.c
py.mclab-bench.appspot.com/?project=12137-pid-and-state-feedback-control-of-dc-moto

```
%%
addpath('controller_component');
addpath('traj_gen_component');

%% Controller specific parameters
Ts = 0.0001;

%% Load component parameters
controller_params;
traj_gen_params;

%% Plant model parameters
% PARAMETERS DC MOTOR
Rm  = 2.06;             % Motor resistance (ohm)
Lm  = 0.000238;         % motor inductance (Henrys)
Kb = 1/((406*2*pi)/60); % Back EMF constant (Volt-sec/Rad)
Kt = 0.0235;            % Torque constand (Nm/A)
Jm = 1.07e-6;           % Rotor inertia (Kg m^2)
bm = 12e-7;             % MEchanical damping (linear model of
                        % friction: bm * dth)
% PARAMETERS LOAD
Jl = 10.07e-6;   % Load inertia (10 times the rotor)
bl = 12e-6;      % Load damping (friction)
Ks = 100;        % Spring constant for connection rotor/load
b = 0.0001;      % Spring damping for connection rotor/load
```

Figure 3.3 McBench code viewer showing the source code of one of the results from *Figure 3.2*.

Each MATLAB project is represented by a CompilationUnits node. This node is annotated with the project root path so that it is easy to find the project subsequently. CompilationUnits is a collection Script and FunctionList nodes.

A Script node represents a script file in the project and it is annotated with the original file name. It also has a StmtList node as a child which represent its body. A FunctionList represents a file with one or more function definitions in it. Each function is represented by a Function node. Input and output arguments are listed in nodes InputArgs and OutputArgs which consist of a list of Name nodes. A Function can have nested functions which are represented as a nested FunctionList node inside the Function. The body of a Function is represented by a StmtList as well. A StmtList is a list that can contain the following type of nodes:

AssignStmt: Represent an assignment statement. Its first child node represents the Left Hand Side (LHS) and the second child represents the Right Hand Side (RHS) expression.

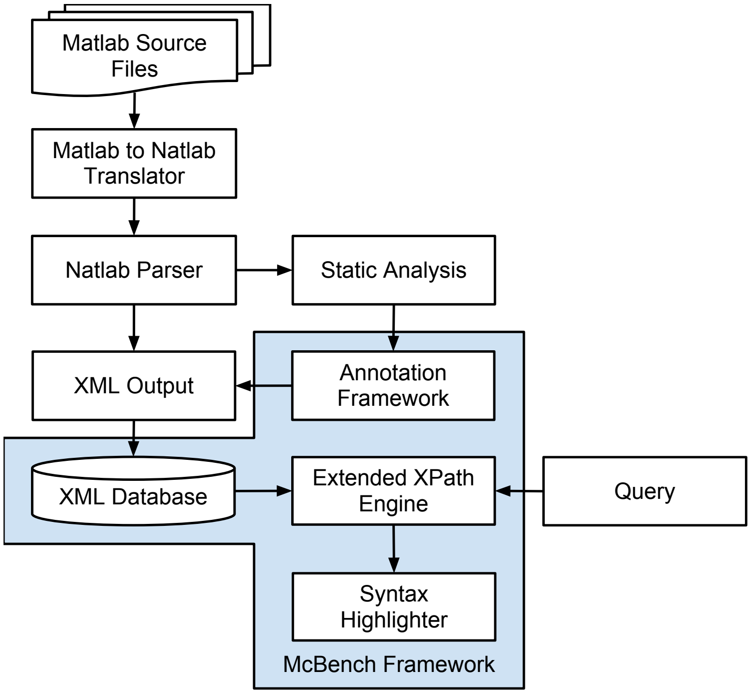


Figure 3.4 McBench architecture. McBench components are inside the highlighted region.

IfStmt: This node represent an if statement. Its first child node is `IfBlock` and it can have more `IfBlocks` which represent elseif syntax in MATLAB. An optional `ElseBlock` is the last node in the `IfStmt` node which represents the `else` part of the statement. The first child of the `IfBlock` node is the condition expression and the second child is the `StmtList` that should be executed if the condition holds.

WhileStmt: Represents a while loop. The first child is the condition expression and the second child is a `StmtList`.

ForStmt: Represents a for loop. The first child is an `AssignStmt`. Its LHS represents the loop variable and its RHS is the range for the loop. The second child of a `ForStmt` is a `StmtList` which is the loop body.

BreakStmt and **ContinueStmt:** These statements can only appear nested inside the loops and represent `break` and `continue` syntax for loop control.

GlobalStmt and **PersistentStmt:** These statements represents declaration of variables as global or persistent. These nodes can contain one or more `Name` nodes that identify the name of the variable.

ExprStmt: This node represents a simple call to a function. It can contain any `Expression` as the target.

There are 48 different types of expressions in *Mc*AST. Here are those that can be used as l-values and r-values.

NameExpr: This represents an access to an identifier. It might be a call a function or access to a variable. The only child is a `Name` node that contains the name of the identifier.

ParameterizedExpr: This node represent an access to an identifier with parameters. Again this can be a variable access or a function call. The first child can be a `NameExpr` (e.g. `disp(param)`), a `CellIndexExpr` (e.g. `A1,2(param)`), or a `DotExpr` (e.g. `A.B.f(param)`). The remaining children are the parameters to the access.

DotExpr: This node represents an access to a struct or a class. The first child can be a `NameExpr`, a `ParameterizedExpr` or a `CellIndexExpr`. The remaining children are the `Name` nodes which are paramaters for the access (e.g. in `a.b.c`, `b` and `c` are represented with `Name` nodes).

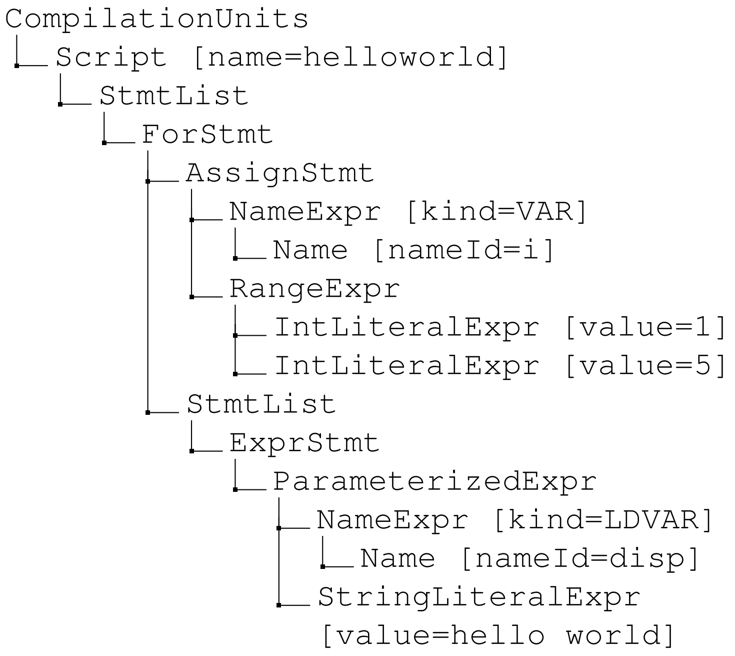
CellIndex: This node represents an access to a struct or an object. The first child can be a `NameExpr`, a `ParameterizedExpr` or a `DotExpr`. The remaining children are the indexing parameters.

MatrixExpr: This node represents a matrix literal definition syntax in MATLAB. It is also used for multi-return syntax (e.g. `[a, b] = call()`).

There are 43 more types of expressions for unary and binary operations and literal values. A complete list of these nodes are available in *Appendix A*. *Listing 3.1* shows a simple source code and *Figure 3.5* shows the corresponding XML document.

```
for i=1:5
  disp('hello world');
end
```

Listing 3.1 A simple script



```
CompilationUnits
 Script [name=helloworld]
    StmtList
       ForStmt
          AssignStmt
             NameExpr [kind=VAR]
                Name [nameId=i]
             RangeExpr
                IntLiteralExpr [value=1]
                IntLiteralExpr [value=5]
          StmtList
             ExprStmt
                ParameterizedExpr
                   NameExpr [kind=LDVAR]
                      Name [nameId=disp]
                   StringLiteralExpr
                   [value=hello world]
```

Figure 3.5 XML tree representation of the source code shown in *Listing 3.1*.

3.3 Annotations

Sometimes questions come up that need more semantic information about the code. For example it might be useful to know how often a variable is read before being assigned to. These questions are easily answered by running static analysis on the program, but new questions with slight changes to constraints might come up and having the information ready at hand is helpful. For example once you found the answer to the previous question, you might want to know how often it occures inside loops. To use our framework to solve this problem we needed to have the static analysis information available in the XML output. We decided to create the Annotations framework and let the programmers create new annotations and choose which set of annotations should be added to the XML output.

The interface is similar to the *Mc*Lab analysis framework. For every node type the user can specify any modification to the corresponding XML node. For example to annotate the XML with Kind information the programmer needs to override the `NameExpr` node. The code for this is shown in *Listing 3.2*.

```
void caseNameExpr(NameExpr n, Element e)
{
  e.setAttribute(name, kind_analysis.get_result(n));
}
```

Listing 3.2 Default node case for `NameExpr` is overridden to add Kind Analysis information

We implemented annotations for Kind Analysis and Reaching Definition Analysis using this framework.

3.4 XPath Queries

XPath is query language specification by World Wide Web Consortium (W3C) designed to search and select nodes in XML documents. The basic idea is to select nodes by their paths. For example in *Figure 3.5* to address the for loop, a simple `/` separated list of nodes that have to be traversed to reach that node can be used:

```
/CompilationUnits/Script/StmtList/ForStmt
```

To refer to a node property instead of a child node the `@` symbol is used before the property name. For example the name of the script can be referred to as:

```
/CompilationUnits/Script/@name
```

Notice that the path might not be unique. In the above example if there was another script in the project, both of these names would be selected as results. To make the selection more specific, constraints can be added to the path inside brackets. This query uses a condition on the name of script (`[@name='helloworld']`).

```
/CompilationUnits/Script[@name='helloworld']/StmtList/ForStmt
```

Conditions can be combined using and and or operators. In the following example, the position(.)=1 condition says that if there are multiple matching nodes, only select the first one.

```
/CompilationUnits/Script[name='helloworld' and
    position()=1]/StmtList/ForStmt
```

Althought XPath can be used to uniquely address XML nodes, we used them to search through XML documents. So it is desirable to create queries that address a pattern instead of one specific node. For example a query to find all calls to disp can be written as:

```
/CompilationUnits//ParameterizedExpr[./NameExpr/Name/@nameId='disp']
```

The above query uses // to address nodes nested anywhere in CompilationUnits node. Inside conditions "." refers to the current node and "./" to its children.

Sometimes the queries get too long and hard to read so we added some functions to the language for common patterns.

is_call(target) This function return true if the current node is a call to a function or script named target. For example a query to find calls to function disp can be written as

```
//*[is_call('disp')]
```

scripts(): This function returns the name of all scripts in current project. For example a query to find calls to scripts can be written as

```
//*[is_call(scripts())]
```

functions(): This function returns the name of all functions in current project.

is_builtin(target): This function returns true if a builtin function with the same name exists in MATLAB.

3.5 Experiments

3.5.1 Setup

In order to have a representative collection of MATLAB code in our experiments we gathered a large number of MATLAB projects. Benchmarks were obtained from individual contributors plus projects from several online code repositories [2] [3] [4] [5]. This is the same set of projects that are used in [DHR11]. The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing. It includes 3057 projects composed of 13438 functions and 2145 scripts. The projects vary in size between 283 files in one project to a single file. A summary of the size distribution of the benchmarks is given in *Table 3.1* which shows that the benchmarks tend to be small to medium in size. However, we have also found 9 large and 2 very large benchmarks. The benchmarks presented here are the most downloaded projects among the mentioned categories.

Benchmark Category	# Benchmarks
Single (1 file)	2051
Small (2-9 files)	848
Medium (10-49 files)	113
Large (50-99 files)	9
Very Large ($\geq$ 100 files)	2
Total	3024

Table 3.1 Distribution of size of the benchmarks

In the following sections we present our query results for our refactoring query-set and then we present three example queries and their results on the benchmark set. These benchmarks are also used in our evaluation of refactorings that we will discuss in *Chapter 5*.

3.5.2 Example #1: Calls to `feval` with string literal targets

Calls to `feval` with string literal targets can be considered a code smell that can be usually refactored. We wanted to implement this refactoring so we looked into these calls in more detail.

The process of writing the query to find these calls can be divided into five steps.

1. Find Parameterized Expressions anywhere in the tree.

[2] http://www.mathworks.com/matlabcentral/fileexchange
[3] http://people.sc.fsu.edu/~jburkardt/m_src/m_src.html
[4] http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/
[5] http://www.mathtools.net/MATLAB/

```
//ParameterizedExpr
```

2. Make sure it is accessing a normal identifier (e.g. not `feval.x()` or `feval{1}()`). That is the first child node should be of type `NameExpr`.

   ```
   //ParameterizedExpr[./*[position()=1 and name(.)=’NameExpr’]]
   ```

3. Filter it by function name

   ```
   //ParameterizedExpr[./*[position()=1 and
      name(.)=’NameExpr’]/Name/@nameId=’feval’]
   ```

4. Filter the results by the node type of first argument. The first argument of function calls appear as the second child node of the `ParameterizedExpr`, after the node that represents the target.

   ```
   //ParameterizedExpr[./*[position()=1 and
      name(.)=’NameExpr’]/Name/@nameId=’feval’]/*[position()=2 and
      name(.)=’StringLiteralExpr’]
   ```

5. For most cases the above query is fine but it does not use Kind information. For example:

   ```
   feval=zeros(50);
   feval(’0’);
   ```

   Here the `feval(’0’)` is a variable access. Nevertheless it shows up in the results. We can use the annotated semantic data to filter out that result. As shown in *Figure 3.5*, Kind information is available in `kind` attribute of `NameExpr` nodes. So we can filter the `NameExpr` nodes for function calls using condition `[`@kind=’FUN’].

   ```
   //ParameterizedExpr[./*[position()=1 and name(.)=’NameExpr’ and
      ./@kind=’FUN’]/Name/@nameId=’feval’]/*[position()=2 and
      name(.)=’StringLiteralExpr’]
   ```

Notice that in the step 2, in order to refer to the first child of a Parameterized Expression we first list all its children using `/*` syntax and then filter the results using `position()` function.

Running this query shows 23 occurrences. As a result, we decided to implement a refactoring to replace these calls.

3.5.3 Example #2: Copy statements inside loops

In MATLAB every assignment is semantically a copy. These copies are sometimes unnecessary and a Virtual Machine implementation can avoid making these copies. In the *Mc*LAB Research Group a question came up regarding copy statements and how they are used. In particular we needed to see the if statements in form `a=b;` were common and whether these type of statements appear inside for loops. The query to find these statements can be written as below:

```
//ForStmt//AssignStmt[./*[position()=1 and name(.)='NameExpr'] and
   ./*[position()=2 and name(.)='NameExpr' and ./@kind='VAR']
```

We used the Kind Analysis results to filter out cases where the RHS is a function call without any parameter (e.g. `b=i`).

There were 105 of these cases. Reading the source codes for these cases helps the developers to understand the common patterns of copy-statement usage and find strategies that can be applied in real world programs.

3.5.4 Example #3: Variables with both Matrix and Scalar types

In MATLAB scalars are semantically 1×1 matrices. In a virtual machine implementation it is desirable to use a different type of variable for this specific type of matrices. This allows the virtual machine to use registers or stack variables and avoid dynamic memory allocation and pointer dereferencing. Static analysis can help to identify these scalar variables. An assignment of an integer value to a variable is a good indication for these variables. But matrices can grow and variables can get new values. So we wanted to write queries to detect variable growths and change of types where variables get both scalar and matrix literal values assigned to them.

There are a number of ways that a matrix can grow.

- Out of bound assignment (e.g. `a(end+1)=1;`)

- – New values gets assigned to the variable. (e.g. `a=zeros(size(a)+1)`);

An special case of the new value is the statement of the form

```
x = [x y];
```

This statement creates a new matrix is the concatenation of two matrices and assigns the resulting value to the first matrix. This kind of matrix growth is slow and can be detected without any shape analysis and additional annotations.

```
//AssignStmt[./*[position()=1 and
   name(.)=’NameExpr’]/Name/@nameId=./*[position()=2 and
   name(.)=’MatrixExpr’]/Row/NameExpr/Name/@nameId]
```

The query looks for assginment statments where the RHS is a matrix expressions that refers to the same identifier as the LHS of the assignment. There were 1998 cases of this type and more than half (1041) of them were inside a for loop. This is an opportunity for both compiler programmers and refactoring tools to take these type of statements into account and make an improvement that affects a large number of projects.

We also wanted to look at codes where a variable is used both as scalar and matrix inside one function. In order to write the query we split the task to three steps.

1. Find assignment that have Matrix literal expressions in the RHS.

   ```
   //AssignStmt[./*[position()=1 and name(.)=’NameExpr’] and
      ./*[position()=2 and name(.)=’MatrixExpr’ and ./Row]]
   ```

2. Find assignment that have integer literal expressions in the RHS.

   ```
   //AssignStmt[./*[position()=1 and name(.)=’NameExpr’] and
      ./*[position()=2 and name(.)=’IntLiteralExpr’]]
   ```

3. Join the results from the previous two queries.

   ```
   //AssignStmt[./*[position()=1 and name(.)=’NameExpr’] and
      ./*[position()=2 and name(.)=’MatrixExpr’ and ./Row] and
      ./*[position()=1 and
      ./Name/@nameId=//AssignStmt[./*[position()=1 and
      name(.)=’NameExpr’] and ./*[position()=2 and
      name(.)=’IntLiteralExpr’]]/*[position()=1]/Name/@nameId]]
   ```

4. Filter the results for only the cases where both `AssignStmt` nodes are inside the same function. The previous query will match even if the assignments are in two separate functions in a project.

   ```
   //AssignStmt[./*[position()=1 and name(.)=’NameExpr’] and
       ./*[position()=2 and name(.)=’MatrixExpr’ and ./Row] and
       ./*[position()=1 and
       ./Name/@nameId=ancestor::Function//AssignStmt[./*[position()=1
       and name(.)=’NameExpr’] and ./*[position()=2 and
       name(.)=’IntLiteralExpr’]]/*[position()=1]/Name/@nameId]]
   ```

   The key here, is the use of `ancestor::TYPE` syntax to refer to the nearest parent of type class TYPE. In this example, when we find an `AssignStmt` node that have `MatrixExpr` in RHS, we want to go up in the tree until we find the enclosing function and only search within that function for the second `AssignStmt`. So instead of `//AssignStmt` we use `ancestor::Function//AssignStmt`.

We found 53 occurrences of this pattern. These can be used to understand the use-cases as well as benchmarks to measure improvements.

3.5.5 Refactoring Query-set

We developed McBench primarily to investigate refactoring opportunities. We wrote following queries to examine different aspects of the language:

Calls to `feval` with string literal target: We will discuss this in the first example in *Section 3.5.2*.

Script files: This query shows us if scripts are used frequently in MATLAB programs. We found out that there are 2145 scripts in our project collection compared to 13438 functions and enabling the programmers to convert them to functions can make a positive impact in the MATLAB code quality.

Calls to scripts from functions: Calls to scripts from functions negatively affect modularity of the code. We wanted to find some code examples where this pattern occurs. We found 197 cases of this pattern and they helped us understand the common use cases of scripts.

3.6 Related Work

Code search is central piece to program comprehension and has been used for maintenance of large code bases. Research has been done extensively in this field to provide tools for programmers to find patterns in their own code base. Several programming languages have been created to let the programmers search through large code base ASTs.

In particular TAWK [GAM96] extends the pattern syntax of AWK language to support matching of ASTs. SCRUPLE [PP94] extends the target language with wild cards for expressions, statements or other syntactic elements. These wild cards can be used in the code snippets to write queries in a language close to the target and are thus easy to understand. In the GENOA language[Dev92] a programmer can specify how the AST should be travesed to find the answer. ACS [PKPZ11] is similar to SCRUPLE in the sense that uses an extended syntax of target language (SAP propietry language, ABAP) for writing queries and elements that are unknown can be replaced by "`...`" or regular expressions.

In order to leverage from the available technologies we decided to use XML and related tools. Search and storage of XML on databases have been research extensively. There were a number of XML search tools that we looked into:

XPath XPath [BBC+07] is a declarative language that provides the ability to navigate the XML document as a tree and select nodes that match a condition.

XQuery XQuery [CFR+01] is a high-level functional language that includes XPath as a sub-language. Functions and SQL like joins bring more flexibility to the language.

We decided to use XPath for its simpler and more concise syntax.

The use of XML to represent an AST and XPath to search through AST XML has been also proposed before. JavaML was introduced by Greg to represent Java AST in form of XML to allow software analyses tools leverage the availability and ease of use of XML tools and techniques [JB00]. Furthermore for ABAP language (Advanced Business Application Programming), ACS (ABAP Code Search) uses XPath and a proprietary database solution to run the queries on large code bases [PKPZ11]. Our approach is somewhat different in that we target our tool to compiler writers, not to search for a specific line in one project, but to understand how users code and common patterns in the language. Furthermore our search framework goes beyond syntactic matching and allows the programmer to extend the XML with semantic data and use this data in queries. To the best of our knowledge, this is the first tool targeted to MATLAB.

Chapter 4
Building Blocks for Refactoring

In this chapter we present the tools we created as building blocks for the refactorings. First we introduce our extensions to the *Mc*SAF [Doh11] static analysis framework to support calls to scripts in *Section 4.1*, and our implementation of liveness and reaching definitions based on this framework in *Section 4.2*. Finally, we present the Return Elimination algorithm in *Section 4.3*.

4.1 Static Analysis For Scripts

In MATLAB, programs consist of functions and scripts. Functions have a set of input and output parameters and a scope which makes it possible to use normal intra-procedural flow analysis techniques. Scripts, on the other hand, run in the caller's workspace which makes it difficult to analyze them unless some context about the caller's workspace is available. Moreover, scripts that are called within functions can modify the data in the caller and any data-flow analysis needs to be aware of these side-effects. For example let's see how the Constant Propagation Analysis would work in MATLAB. *Figure 4.1* shows a simple project. Function `f` is the main function that calls into function `f2` and script `s1`. In the first line of the function `f`, the variable `v` is assigned to the constant value 5. In the next line there is a call to `f2` which assigns the constant value 1 to `v` but since each function call creates a new workspace, the variable `v` in `f2` does not share its value with the one in `f`. So after the call to `f2` returns, the variable `v` still has 5 as its value. But in the script `s1` which is called in

```
function f()
  v = 5;
  f2()
  s1();
  disp(v);
end
```

(a) Function f

```
v = 3;
s2();
```

(b) Script s1

```
v = 1;
```

(c) Script s2

```
function f2()
  v = 1;
end
```

(d) Function f2

Figure 4.1 A basic MATLAB project with two functions and two scripts

the next line, this is not the case. The variable v in s1 is the same variable present in f and the assignment in the first line of s1 changes its value to the constant value 3. Furthermore in s2 the variable v gets the value 1. So in last line of function f, disp(v) actually prints out 1. In this example, an intra-procedural analysis that does not correctly handle scripts would say that the value of v would be 5 in the last line of f. A correct, but conservative analysis that handles scripts properly can only say that the value of v can not be inferred intra-procedurally after a call to a script. One possible approach might be to inline all calls to scripts and then run the analysis, however this approach will fail in presence of recursive scripts and correct handling of script sematics is hard as shown in *Section 5.1.1*.

Since a conservative approach is suboptimal in presence of scripts and since analyzing scripts was crucial to our refactorings, we implemented an extension to *Mc***SAF** to support calls to scripts. For each script a summary is created using a flow-analysis. In the constant propagation example, using our framework one can implement a summary analysis to find all the assigned variables in the called scripts and kill the constant propagation results for those variables. Another summary analysis could run the Constant Propagation Analysis on the scripts and find the constant values after the calls to scripts and update the results for the calling function using the summary data. Summaries are propagated in the framework until a fixed-point is reached. When a fixed-point is reached for every script the results can be used in the analysis of the calling function.

```
v = 3;
B();
```

(a) Script A

```
if (cond)
  A();
v = 1;
C();
```

(b) Script B

```
v = 5;
F();
```

(c) Script C

```
function F()
   v = 2;
   A();
end
```

(d) Function F

Figure 4.2 A small recursive program

To speed-up the fixed-point analysis, first we create a script call graph. We only visit a node in a strongly connected component if all of the successor nodes are visited first. For each strongly connected component, we iteratively propagate results until a fixed point is reached. To see how it works let's consider the example in *Figure 4.2*. Let's say we are analyzing the function F and we encounter the call to the script A. To build the summary for the script A, the first step is building a script call graph. The scripts A and B have calls to each other which forms a recursion. The script C is also called from B. The call graph only includes nodes A, B and C since these are the scripts that are reachable from the function F. Since the call graph is built only for script calls, it does not include the call to function F from C. *Figure 4.3* shows the script call graph for the above example.

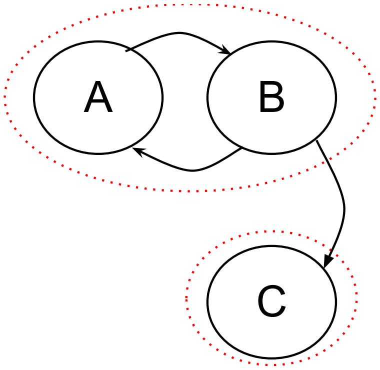


Figure 4.3 Script call graph for example in *Figure 4.2*. Strongly connected components are shown with dotted lines.

The scripts A and B are in the same component, with the script C as its successor. The script C is in a separate component with no successors. So the algorithm starts by computing the summary for C. Then it iteratively updates the results for A and B until there is no change. *Listing 4.1* shows the algorithm in more details.

```
function ScriptAnalysisDriver(Callgraph)
  N = StronglyConnectedComponents(Callgraph)
  E = {}
  for ni ∈ N
    for nj ∈ N
      if ∃ s ∈ ni and ∃ s2 ∈ nj such that s calls s2
        E = E ∪ {⟨ni, nj⟩} # For each call add an edge from caller to callee

  results = {⟨ni, ∅⟩ | ni ∈ N}
  for ni in post_order(C, E) # For each component
    leader_node = ni[0]; # get the first member of the component
    q = queue();
    q.push(leader_node);
    while q.size > 0 # Repeat until there aren't any changes
      s = q.pop()
      summary = analyze(s)
      if ⟨s, summary⟩ ∉ results
        results = results ∪ {⟨s, summary⟩}
        for s2 ∈ ni
          if s2 calls s
            q.push(s2)
```

Listing 4.1 Summary-approach flow analysis for scripts

4.2 Liveness and Reaching Definitions Analysis

As we will see in the refactorings, Liveness and Reaching Definitions analyses are very fundamental tools. In this section we present the MATLAB versions of these analyses that we developed to support the refactoring transformations presented in *Chapter 5*

4.2.1 Liveness Analysis

Liveness analysis computes, for each program point *p*, the set of variables that are used before being defined on some path from *p*. We implemented this analysis using a backward flow sensitive analysis with proper handling for script calls using our *Mc*SAF extension. Similar to Liveness Analysis in other languages our implementation for MATLAB takes variable accesses, assignments and output parameters into account:

- **Output parameters:** All the output parameters are added to the live set at the end of program.
- **Variable access (e.g. '= a'):** In this case variable that is being accessed is added to the live set.
- **Variable assignment (e.g. 'a ='):** In this case the variable that is being assigned to is killed from the live set unless it is a global variable.

There are also cases in MATLAB which aren't common in other languages. In our implementation we have considered these special cases:

- **Global Variables (e.g. 'global a'):** In MATLAB several functions can declare a variable as global and in that case they all share the same variable. Since it is not trivial to find all the accesses to global variables, we make a conservative assumption that for every global variable there is a variable access at every program point so they always stay in the live set.
- **Persistent Variables (e.g. 'persistent a'):** Persistent variables are similar to static variables in C or Pascal. Their value is saved in global variables but they aren't accessible in other functions. In this case it is sufficient to assume that there is an access to the variable at the end of program that stores the local variable in a global storage.
- **Save workspace or variable (e.g. 'save matrix' and 'save matrix a'):** In this case all the target variables are added to the live set.
- **Clear variables (e.g. 'clear a'):** In this case all target variables are removed from the live set.
- **Script Call:** For script calls we compute a summary that contains the live variables in the beginning of each script. Then all these live variables are added to the current live set. This is implemented using our extension to the analysis framework that we

presented in *Section 4.1*.

4.2.2 Reaching Definitions Analysis

In Reaching Definitions Analysis, for each access to a variable we compute the set of possible program-points where the variable is defined. We implemented this analysis using a forward flow sensitive analysis that also uses our *Mc*SAF extension to support calls to scripts. For Reaching Definition Analysis these are the common cases that need to be considered:

- **Variable assignment (e.g. 'a ='):** In this case all the definitions of the variable that is being assigned to is killed except the GLOBAL_DEF and a new definition for this program point is added.
- **Input parameters:** Each input parameter is assigned a special PARAM_DEF value. This kills the UNDEF value for input parameters.

Apart from the common cases there are few cases special to MATLAB semantics that we considered in our implementation:

- **Global Variables (e.g. 'global a'):** An special GLOBAL_DEF is added to the definitions of the variable.
- **Persistent Variables (e.g. 'persistent a'):** In this case variable will have a special PERSISTENT_DEF at the beginning of the program.
- **Load variables (e.g. load matrix'):** In this case a special LOAD_DEF is added to all the variables in the current function or script but nothing is killed since a load might fail.
- **Clear variables (e.g. 'clear a':)** In this case all target variables are assigned to UNDEF.
- **Undefined Variables:** At the beginning of the program every identifier in the function or script body is added the current set with a special UNDEF value.
- **Script Call:** For script calls first the script results is computed using the same Reaching Definition Analysis. Then for every variable v and reaching definition set r in the script results, we do the following steps:
  - If r does not include UNDEF, we kill the definitions of v and replace it SCRIPT_DEF.

- If *r* includes UNDEF as well as some other definitions, a SCRIPT_DEF is added to the definitions of *v*.
- If *r* only includes UNDEF, no change is necessary to the current set.

Table 4.1 shows an example function with step-by-step results for each of the analyses.

		Liveness	Reaching Definitions
		∅	<A,[UNDEF]>, <B,[UNDEF]>, <C,[UNDEF]>
1	A = 10	A	<A,[1]><B,[UNDEF]><C,[UNDEF]>
2	B = 10	A,B	<A,[1]><B,[2]><C,[UNDEF]>
3	if (A<12)	A,B	<A,[1]><B,[2]><C,[UNDEF]>
4	C = B + A	∅	<A,[1]><B,[2]><C,[4]>
5	end	∅	<A,[1]><B,[2]><C,[UNDEF,4]>
6	C = 11	C	<A,[1]><B,[2]><C,[6]>
7	B = 11	B,C	<A,[1]><B,[7]><C,[6]>
8	A = B + C	∅	<A,[8]><B,[7]><C,[6]>

Table 4.1 A simple script with results for both Liveness (backward analysis) and Reaching Definitions Analysis (forward analysis) for each program point

Notice that in some cases a variable might have multiple reaching definitions. In the cases where one of these definitions is UNDEF, the variable might not be initialized in some of the paths. In the example shown in *Table 4.1*, variable C has both UNDEF and line-4 as definitions which shows that it is not initialized in the case the condition in line-3 does not hold. If UNDEF is not in the set, then the variable must be defined along all paths. We use this feature to compute Definitely-Assigned Analysis with Reaching Definitions Analysis at the same time.

4.3 Return Elimination

In MATLAB, programs can have return statements which cause the function or script to immediately return. There is also an implicit return at the end of every script or function. When we move code between function and script boundaries return statements don't keep their meanings. Converting return statements to explicit control flow makes it simpler to reason about the control flow. We implemented a simplification that eliminates return statements and replaces them with conditionals. We start by collecting the locations of return statements. We check if there is at least one return statement, otherwise there is no reason to modify the code. If there is a return statement, first the body is wrapped inside a for loop that runs only once and a flag is initialized at the beginning.

```
for TEMP_RETURN=1:1
  RETURN_FLAG = false;
  body
end
```

For each return location, we replace `return` statements with an assignment to RETURN_FLAG and a break statement.

```
RETURN_FLAG = true;
break;
```

The last step is for each loop inside the body that had a nested return statement, wrap the following sibling statements in an if statement:

```
while  loop
  ...
end
if (~RETURN_FLAG)
  following − siblings
end
```

This makes sure that if the loop was broken as a result of a previous `return` the program does not continue after the loop. *Figure 4.4* shows a complete example of this simplification. In the first two lines of *Figure 4.4* (b), the RETURN_FLAG is initialized and the auxiliary for loop is added. Lines 3, 4 and 5 code comes directly from the original program, unmodified. The return statement in line 4 of (a) is then replaced by an assignment to RETURN_FLAG at line 6 and a break from the enclosing loop at line 7 of (b). At line 10 of (b), right after the loop, we check the flag and avoid running the subsequent statements in the case that the flag is set to true.

We will use this simplification and the analyses for the refactorings described in the following chapters.

```
for i=1:5
  disp(’hello’);
  if (i>3)
    return;
  end
end
disp(’world’);
```

(a)

```
RETURN_FLAG=false;
for TEMP_RETURN=1:1
  for i=1:5
    disp(’hello’);
    if (i>3)
      RETURN_FLAG = true;
      break;
    end
  end
  if (~RETURN_FLAG)
    disp(’world’);
  end
end
```

(b)

Figure 4.4 A MATLAB script with a return statement before (a) and after (b) the return simplification

Chapter 5
Refactoring MATLAB

In this chapter we introduce a family of behavior-preserving and automated refactorings aimed at restructuring functions and scripts, and calls to functions and scripts. We start with a standard refactoring, function-inlining, which demonstrates the key concepts of ensuring that the Kind and lookup of identifiers remains correct. Function-inlining is useful in MATLAB for efficiency reasons as many JIT-level optimizations work best intra-procedurally. Thus, the function inlining refactoring may be useful both for the programmer and for other compiler tools. We then describe two refactorings for scripts, inlining scripts into functions and converting scripts to functions. Both of these are useful for eliminating scripts. Then we introduce our *Extract Function* refactoring and a refactoring to replace calls of `feval` to direct function calls. In the next chapter we present the evaluation of our refactorings.

5.1 Inlining Scripts and Functions

In this section we present our approach for the *Inline Function* and *Inline Script* refactorings. The programmer identifies a particular call site which should be refactored by inlining. There are several reasons why MATLAB programmers may want to apply such a refactoring. They may want to inline calls to scripts in order to eliminate them. They may want to inline functions at key call sites to enable other MATLAB optimizations or tools, or as a precursor to another refactoring.

Our approach is to create an inlining candidate and then analyze if the inlining is safe or not. Inlinings that are safe are performed, whereas inlinings that definitely are not safe generate an error message and will not be performed. Inlinings that may be safe under a reasonable assumption generate warnings to the programmer, so the programmer can decide whether to proceed or not.

If an inlining is performed, the inlining procedure attempts to keep the original identifier names, renaming identifiers only when necessary to ensure the same semantics.

5.1.1 Inline Script

The *Inline Script* refactoring proceeds as follows. Given a call site c in function f that calls script s, the refactoring procedure creates f_s, a copy of f with s inlined, and then verifies that f_s has the same behaviour as f. To create function f_s, if s contains return statements, a transformation is applied to s to only have one exit point at the end of the script. Then the call site c is replaced with the body of s. *Listing* **??** illustrates the result of this first step, inlining the call to script `SMultiplyCompatible`.

The verification phase starts with checking the lookup semantics. Scripts run in the same workspace and function lookup environment as the script call site with the exception that scripts don't have access to nested functions in the caller function - *subfunctions* or functions inside `private` folder of the calling function are still accessible. The inliner checks to see if any possible call site that was originally in s can resolve to a nested function in f_s and if so issues a *NameResolutionChangeException*. In this case the refactoring cannot be done.

The next step is to verify f_s regarding the Kind analysis semantics. To perform the verification the flow-sensitive Kind analysis presented in [DHR11] must be run on the original script s, the original version of f, and the inlined copy f_s. Given the Kind analysis results, all identifiers in f_s are verified as follows.

Simple checks that immediately pass

Any identifier i that is in f, but not in s, needs no further verification since introducing the body of s into f cannot possibly impact the Kind of i.

Any identifier *i* which has the same Kind in *s*, *f* and f_s will have the same meaning in the inlined version and so no further verification is necessary.

Any identifier *i* which is not defined in *f*, but has the same Kind in *s* and f_s also retains its meaning and no further verification is necessary.

Kind conflicts resolved by variable renaming

An identifier *i* with Kind FN in *f* and Kind VAR in *s* or vice-versa will lead to a Kind mismatch error for f_s. This is precisely the problem demonstrated in *Listing 2.4*, where `ndims` has Kind FN at line 2 and Kind VAR at line 7. This means that the refactoring is not behavior-preserving, because the inlined version would result in a compile-time Kind error, whereas the original version would not. This mismatch can be resolved by applying a variable renaming refactoring. If *i* initially had the Kind VAR in *s*, then a copy of *s* is created in which *i* is renamed to a fresh name, otherwise a copy of *f* is created in which *i* is renamed to a fresh name. After renaming, the inlining refactoring is restarted. In our example from *Listing 2.4* the variable `ndims` at lines 7, 10 and 12 would be renamed to `ndims2`.

Such a renaming is usually semantics preserving, except when the variable being renamed is referenced via a dynamic feature like `eval`. For example, it would be incorrect to name variable `x` in the statement sequence `x = 3; eval('x=x+1'); y = x;` since eval would not refer to the renamed `x`.

It would be possible to warn the user of such renamings so that the user can verify that the renamed variable is not being accessed via a dynamic feature.

Kind specializations

The remaining cases all involve situations where the original Kind of an identifier *x* (in either *s* or *f*) was ID, and the inlined version f_s has a more specialized Kind for *x* (VAR or FN). This is a potential problem because an identifier with Kind ID has a very general lookup (first the current workspace is searched for variable and if a variable is not found then a function lookup is used). If a more precise Kind is assigned to the inlined identifier, then the lookup is specialized to that Kind (VAR is only looked up as a variable in the workspace and FN is only looked up as a function). Since the lookup becomes more specific the behaviour may change. Thus, we must consider two cases, when an ID is specialized to a VAR, and when an ID is specialized to a FN.

An ID x with a Kind that is specialized to VAR is semantics-preserving if all uses of x have definitely been preceded by a an assignment to x. In this case the lookup will always find the variable in the current workspace, and thus an ID lookup is the same as a VAR lookup. Thus, for these situations we check that x is assigned on all paths, and if so, we allow the refactoring. If x is not assigned on all paths we reject the factoring with a *IDNotDefAssignedException*.

An ID g with a Kind that is specialized to FN is in practice usually also semantics-preserving. The only case in which this occurs is when there is no explicit definition of g in f_s (otherwise g would have Kind VAR) and g is found the library of named functions (i.e. there does exist a named function called g). Thus, it is highly likely that the programmer intends this to be looked up as a function. In this case we issue a warning that we are assuming that g refers to a function and the user can accept the refactoring if this assumption is correct. The assumption would only be incorrect if g was being assigned to via a dynamic feature.

Our example from *Listing 2.4* illustrates the most common case of specialization. In the inlined version both `size` and `length` have Kind FN, whereas in the script they had Kind ID.

5.1.2 Inline Function

The *Inline Function* refactoring allows the programmer to identify a call site c inside a function f in form of `[output]=g(input);` and it inlines the call.

The function inliner creates the function f_g. f_g is created as a copy of f with the call site replaced with a statement sequence. For each input expression e_i which corresponds to parameter $inparam_i$ a new assignment statement $p_i = e_i$ is created at program-point c. The body of g is transformed using our Return Elimination so as to have only one exit point is then inserted after the last assignment for input arguments. After that assignment, assignments of the form $p_i = e_i$ are added for each output parameter p_i. *Figure 5.1*(a) shows

```
function r = MultiplyFn2(a, b)
  if (ndims(a)==3 && ndims(b)==3)
    r = do3DMult(a,b);
  else
    isCompatible =
        MultiplyCompatible(a,b);
    if (isCompatible)
      r = a * b;
    else
      error('Matrix Dimension
          Error');
    end
  end
end

% Kinds ...
% VAR: r, a, b, isCompatible
% FN: ndims, do3DMult,
    MultiplyCompatible,
% error
```

(a)

```
function r = MultiplyFn2(a, b)
  if (ndims(a)==3 && ndims(b)==3)
    r = do3DMult(a,b);
  else
    % ---- start of inlined call
    n = a;
    m = b;
    ndims = size(n);
    mdims = size(m);
    r = ((length(ndims)==2) &&
         ...
         (length(mdims)==2) && ...
         (ndims(2)==mdims(1)));
    isCompatible = r;
    % -- end of inlined call
    if (isCompatible)
      r = a * b;
    else
      error('Matrix Dimension
          Error');
    end
  end
end

% Kinds ...
% VAR: r, a, b, n, m, mdims,
    isCompatible
% FN: do3DMult, size, length,
    error
% ERROR: ndims
```

(b)

Figure 5.1 Original function (a) and inlined version before the necessary renames where `ndims` has a Kind conflict (b).

an example function `MultiplyFn2` and *Figure 5.1*(b) shows the initial inlining of the call to "`isCompatible = MultiplyCompatible(a,b)`".

A key step is deciding whether or not to accept the inlining by verifying the conditions. If the conditions are verified a clean up process removes as many unnecessary introduced variables as possible.

The verification process starts with matching the name resolution results. For every identifier in g with Kind FN, the program checks if the lookup returns the same results

```
function r = MultiplyFn2(a, b)
if (ndims(a)==3 && ndims(b)==3)
    r = do3DMult(a,b);
  else
    % ---- start of inlined call
    n = a;
    m = b;
    ndims2 = size(n);
    mdims = size(m);
    r2 = ((length(ndims2)==2) && ...
         (length(mdims)==2) && ...
         (ndims2(2)==mdims(1)));
    isCompatible = r2;
    % -- end of inlined call
    if (isCompatible)
      r = a * b;
    else
      error('Matrix Dimension Error');
    end
  end
end

% Kinds ...
% VAR: r, r2, a, b, n, m, mdims, ndims2, isCompatible
% FN: do3DMult, error, size, length, ndims
```

(a)

```
function r = MultiplyFn2(a, b)
  if (ndims(a)==3 && ndims(b)==3)
    r = do3DMult(a,b);
  else
    % ---- start of inlined call
    ndims2 = size(a);
    mdims = size(b);
    r2 = ((length(ndims2)==2) && ...
         (length(mdims)==2) && ...
         (ndims2(2)==mdims(1)));
    % -- end of inlined call
    if (r2)
      r = a * b;
    else
      error('Matrix Dimension Error');
    end
  end
end

% Kinds ...
% VAR: r, r2, mdims, ndims2
% FN: do3DMult, error, size, length, ndims
```

(b)

Figure 5.2 Inlined version of `MultiplyFn2` after the necessary renames (a) and spurious copies removed (b)

before and after inlining and otherwise rejects the refactoring by raising a *NameResolutionChangeException*.

The next step is to verify the Kind analysis results using the following rules.

- For every identifier that is only present in one of the functions f or g no further verification is necessary.
- For every identifier with Kind FN in both f and g, no further verification is necessary.
- For every identifier with Kind VAR in one of the functions f or g, and Kind VAR, ID or FN in the other, a rename refactoring is triggered for the variable. Note that

in script inlining we only needed to do renaming for conflicts between VAR and FN because a script uses the same workspace as its caller. However, when inlining a function, we are merging the workspaces of f and g and if an identifier occurs in both f and g we must distinguish them by renaming.

– For every identifier with Kind ID in one of the functions f or g, and with the Kind ID or FN in the other function an *IDConflictException* is raised, and the refactoring will not be done. The rational for this decision is that within functions identifiers will only have a Kind ID when there is neither an explicit assignment nor a function of that name in the library. This implies that the identifier is being defined through some dynamic feature, and thus the inlining is not safe.

Figure 5.2(a) shows the result of our example after the verification and renaming has been done. Note that variable `ndims` was renamed due to a Kind conflict, and variable `r` was renamed because this was a VAR in both the caller and the callee.

At this point the verification is complete and if no exceptions were raised, then f_g has the same behaviour as f. However, the inlined code may have a significant number of new copy statements (one for each input and output parameter). Thus, to make the output code cleaner, for each new assignment statement that was introduced for the parameters another refactoring process checks if it is necessary and if not removes the assignment and performs a copy propagation. More precisely, for each statement *stmt* in the form $p = e$; where e is also a variable, we want to replace every use of p in f_g with e. In order to do that we compute the use-def relationships. For every use of p defined by *stmt* the algorithm uses *Reaching Copy Analysis* to see if the use is a copy of e in the statement *stmt*. If all the uses were copies of the definition in *stmt*, the assignment statement can be removed and all the uses of p are changed to use e.

Figure 5.2(b) shows the result after copy elimination for our running example. Note that the copies to `a`, `b` and `isCompatible` have been removed.

5.2 Converting Scripts to functions

Given that MATLAB scripts are very non-modular, a refactoring that converts scripts into functions is useful for improving the overall structure of MATLAB programs. The programmer provides a complete program, and also identifies the script to be converted to a function. If the refactoring can be done in a semantics-preserving manner, the *Script-to-Function* refactoring converts the script to a function and replaces all calls to the script with calls to the new function. Although useful, this refactoring is more complex than either function or script inlining.

This refactoring requires the use of two additional analyses, *Reaching Definitions* and *Liveness* which we described in *Chapter 4*.

To convert a script s to function f we need to: (1) determine input and output arguments that will work for all calls to s, and (2) make sure that program behaviour will stay the same after conversion.

To determine the input and output arguments, We first compute $scriptDefAssigned(s)$, the set of variables that are definitely assigned by s (i.e. all variables that don't have UNDEF in the reaching definitions at the end of script). We also compute $scriptMayAssigned(s)$, the set of variables that are assigned at least in one path to end of s (i.e. have at least one reaching definition at the end of script that's not UNDEF). Finally, we compute $scriptLives(s)$, the set of live identifiers with Kind VAR or ID at the beginning of the body of s.

In order to build the function f some information about the contexts that script s is being used is necessary. For each call c_i to the script s, the following steps are performed:

- If the call site is inside some other script s', a script a *ScriptCallFromScriptException* is raised. The lack of structure in scripts makes it impossible to compute the set of inputs and outputs for the script s.
- For each call site c_i, the set $callAssigned(c_i)$ of definitely assigned variables and the set $callLives(c_i)$ of live variables are computed at program point of c_i. The set $input_i$ is defined as

  $$input_i : scriptLives(s) \cap callAssigned(c_i)$$

  and $output_i$ is defined as

  $$output_i : scriptMayAssigned(s) \cap callLives(c_i)$$

  If any identifier in the $output_i$ is not in $scriptDefAssigned(s)$ an *OutputNotDefinitely-AssignedException* is raised.

- The set $lookup_i$ is defined as:

$$lookup_i\{\langle n : ResolveName(n)\rangle | n \in identifers(f) \wedge Kind(n) \in \{\textsc{Id}, \textsc{Fn}\}\}$$

After computing the $input_i$, $output_i$ and $lookup_i$, for each call site, first we verify that all the call sites have the same input set. If there was any difference in any of the sets an *InputArgsNotMatchingException* is raised. If they all match the set is used as the set of input arguments for the function f. The output arguments are constructed using $\bigcup_{i=1}^{n} output_i$. Some of the outputs from script s might not be used at a specific call site (i.e. that identifier is not live). But the refactoring can continue and the unused outputs can be ignored using "$\sim$" syntax or a temporary variable. Then the function f is built using the constructed inputs, outputs and the body of s.

The next step is checking name resolution results. For every identifier n with Kind ID or FN in f, the pair $\langle n : ResolveName(n)\rangle$ should match the pair in $lookup_1$, ..., $lookup_n$. If there were any mismatches a *NameResolutionChangeException* is raised. To perform *ResolveName*, f is assumed to be a primary function in the same folder as s.

The final step is to check Kind results. Similar to script inlining, identifiers with Kind ID can turn to FN, or remain ID and identifiers with Kind VAR and can cause a Kind conflict. The precise rules are:

- Identifiers that stay VAR or FN don't need any further verification.
- Identifiers with Kind ID in both s and f might be referring to variables created dynamically in the calling functions. Since the function f is no longer running in the calling function environment and workspace, it can not access to those variables. So for any identifier with Kind ID in function f an *UnresolvedIDException* is raised.
- For all identifiers with Kind ID in s and Kind FN in f it is possible to warn the user that the refactoring is assuming the ID is a function, which is the usual case.
- For all identifiers with Kind VAR in s and Kind conflict in f an *UnresolvedKindConflictException* is raised. This type of Kind conflict can not be resolved with renaming because it is not clear when the identifier was meant be to a function and when it was meant to be a variable.

After the verification process each call c_i to script s is replaced with an assignment. The

```
function c=foo(a, b)
 for i=1:10
   c = a*i+b
   if (c>100)
     return;
   end
 end
 disp(c)
end
```

(a) Original Function

```
function c=foo(a, b)
 RETURN_FLAG=false;
 for TEMP_RETURN=1:1
   for i=1:10
     c=a*i+b;
     if (c>100)
       RETURN_FLAG=true;
       break
     end
   end
   if (~RETURN_FLAG)
     disp(c);
   end
 end
```

(b) After Return Elimination

Figure 5.3 An example function for *Extract Function* refactoring

left hand side of the assignment is formed by putting o_j for every output argument o_j in f that is also present in $output_i$ and putting "~" for those arguments that are not [1]. The right hand side of the assignment is formed by simply a call to f with all the input arguments.

5.3 Extract Function

Extract Function makes it possible to split functions with long bodies to into smaller ones. The refactoring takes a sequence of statements as an input and turns them into a new function. During the study of our MATLAB projects collection, we found out that the average number of lines of code per function is 22.7. According to English et. al. this number is 12.8 for Java [MP09]. This shows that MATLAB functions tend to be longer and in more need of this refactoring.

Figure 5.3(a) shows an example function where we want to extract the loop body as a new function. The first step in the refactoring is to remove the return inside the loop. *Figure 5.3*(b) shows the original function after this transformation.

Next we create the new function with the for loop as its body and a list of inputs and

[1] In MATLAB, you can ignore some of the outputs of a function call by using "~". For example in `[~,a]=size(b);`, the variable `a` will contain the size for the second dimention of `b`.

```
function [RETURN_FLAG, c]=NEW_FUNC(a, b)
  for i=1:10
   c=a*i+b;
   if (c>100)
    RETURN_FLAG=true;
    break
   end
  end
end

function c=foo(a, b)
 RETURN_FLAG=false;
 for TEMP_RETURN=1:1
  [RETURN_FLAG, c]=NEW_FUNC(a, b);
  if (~RETURN_FLAG)
   disp(c);
  end
 end
```

(a) After extracting function, RETURN_FLAG might be undefined after the call

```
function [RETURN_FLAG, c]=NEW_FUNC(a, b, RETURN_FLAG)
  for i=1:10
   c=a*i+b;
   if (c>100)
    RETURN_FLAG=true;
    break;
   end
  end
end

function c=foo(a, b)
 RETURN_FLAG=false;
 for TEMP_RETURN=1:1
  [RETURN_FLAG, c]=NEW_FUNC(a, b, RETURN_FLAG);
  if (~RETURN_FLAG)
   disp(c);
  end
 end
```

(b) Final version of the extracted function and the call

Figure 5.4 The steps showing the creation of the new function and the final result

outputs. Since `a`, `b` are live at the beginning of the function and they are both defined, we add them to the input arguments. `RETURN_FLAG` is live after the sequence and is assigned in the sequence body so it should be in the outputs. *Figure 5.4*(a) shows the new function with these inputs and outputs. But in the case where the `RETURN_FLAG=true` does not get executed the return value of `RETURN_FLAG` is undefined. This is a problem since previously if that section of code didn't get executed the value of `RETURN_FLAG` would have stayed as the old value (false).

To fix this problem we add all the output arguments that are not definitely assigned inside the sequence but might be defined along some paths to s in the original function to the input arguments. This way, even if the variable does not get a new value, the old value would stay intact. *Figure 5.4*(b) shows the final version program, after adding these output arguments to the inputs.

Our refactoring takes a sequence of statements s such that the sequence corresponds to a sequence in the AST inside the function f as the input and starts by eliminating the return statements in s using the algorithm described in *Chapter 4*. Then a new function is created with the sequence of statements s as its body.

Similar to the Script to Function algorithm, we use Reaching Definition Analysis and Liveness Analysis to determine the function parameters. We run both analyses on the original function f that contains s and on our newly created function f_n. All the Live variables at the beginning of the function f_n that have any reaching definitions other than UNDEF or GLOBAL_DEF at the beginning of s are added to the input arguments. All the Live variables right after the last statement in s that have any reaching definitions other than UNDEF or GLOBAL_DEF at the end of f_n are added to the output arguments. If any of these output arguments has UNDEF in its reaching definitions at the end of f_n, it is not definitely assigned. If it is definitiely assigned before s it is also added to input arguments (similar to `RETURN_FLAG` in the above example). Otherwise the process fails. For every global variable in f that is mentioned in f_n, we define them as globals in the new function f_n.

The next step is checking Kind Analysis and name resolution results. For every identifier n with Kind FN in f_n, the lookup result for n in the context of f should match the lookup results for n in context of f_n. If there are any name resolution changes a *NameResolutionChangeException* is raised. The Kind of all the identifiers in f_n should also match their Kind in f, otherwise a *UnresolvedKindConflictException* is raised. Furthermore no identifier in f_n should have the Kind ID, otherwise a *UnresolvedIDException* is raised. Then the statement sequence s is replaced with an assignment that has a call to f_n with the input arguments on the RHS and the output arguments on the LHS, similar to the statement that replaces a script call after converting the script to a function.

5.4 Replacing `feval`

The MATLAB builtin function `feval` takes a reference to a function (a function handle or a string with the name of the function) as an argument and calls the function. If an `feval` can be replaced by a direct call to a function, this leads to cleaner and more efficient code.

Somewhat to our surprise, we found numerous cases where programmers used a string literal in `feval`, for example `feval('myfunc',x)`. Consider the code in *Listing 5.1*, ex-

```
1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  % this program calculates and plots the wave-vector
3  % diagram (i.e.%photonic bands at constant frequency)
4  % ...
5  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6  %%% the package contains the following programs:
7  %%% pwem2Db.m - main program
8  %%% epsgg.m - routine for calculating the matrix
9  %%% of Fourier coeff of dielectric fn ...
10 clear all
11 tic
12 omega=0.45; % normalized frequency "a/lambda"
13 r=0.43; % radius of cylindrical holes
14 na=1; nb=3.45; % refractive indices
15 ...
16 %%% matrix of Fourier coefficients
17 eps1 = feval (’epsgg’,r,na,nb,b1,b2,N1,N2);
18 ...
19 S=2.5; % point size for scatter plot
20 for j=1:length(BZx)
21    %%% diagonal matrices with elements
22    %%% (kx+Gx) si (ky+Gy)
23    [kGx, kGy] = feval(’kvect2’,BZx(j),BZy(j),
24                      b1,b2,N1,N2);
25    [P, beta]=feval(’oblic_eigs’,omega,kGx,kGy,
26                      eps1,N);
27    ...
28 end
```

Listing 5.1 Excerpts from a script which uses `feval` (... corresponds to elided code)

tracted from one of our benchmarks.[2] It appears that every time the programmer invokes his own function, he uses `feval` (lines 17, 23 and 25). This must have been a programming misunderstanding, as there is no valid reason to use `feval` rather than a direct call in this program.

The only, not very valid reason, that one might use `feval` with a string constant is to force a function call when there is also a variable with the same name. For example, if a function defined the variable `i`, but the programmer also wanted to access the built-in library function called `i`, they might use `feval(’i’)`. It would seem much better in this

[2] Excerpts from http://mathworks.fr/matlabcentral/fileexchange/22774-wave-vector-diagram-for-a-2d-photonic-crystal/content/pwem2Db.m.

case to rename the variable so as to avoid the conflict.

Our refactoring tool looks for those calls to `feval` which have a string constant as the first argument, and then uses the results from Kind analysis to determine if an identifier with Kind VAR with the same name exists. If there is no such identifier in the function, the call to `feval` is replaced with a direct call to the function named inside the string literal. Of course, with more complex string and call graph analyses one could support even more such refactorings. However, it is interesting that such a simple refactoring is useful.

Chapter 6
Evaluation and Related work

In this chapter we present our evaluation of the refactoring algorithms on the same set of MATLAB projects that we gathered for McBench described in *Chapter 3*.

In order to measure the effectiveness of our approach, we aim to answer these questions for each refactoring:

RQ1 How many refactoring opportunities are available?

RQ2 How many times the algorithm could complete without renaming any identifiers?

RQ3 How many times there were assumptions that needed to be verified by the programmer?

RQ4 How many times each exception occurs?

RQ5 How invasive are the changes to the user code?

6.1 Inlining Scripts

As shown in *Table 6.1*, to answer the research questions for script inlining, we counted: **RQ1**, every call to a script from a function as an inlining opportunity (191 calls); **RQ2**, the number of simple cases with and without renaming (104) which corresponds to the number of inlinings that succeed without user intervention; **RQ3**, the number of times some that IDs were changed to FNs; and **RQ4**, the number of times each exception occurs. The results show that more than half the inlining refactorings finished without any user intervention

(104 of 191). For 77 cases the user has to verify that there is no hidden variable definition, and for 10 out of 191 cases the inlining was not possible. For **RQ5**, the only change to the source codes that was necessary to finish in this refactoring was renaming variables, which is not a significant change to the program.

Inlining result	# call sites
Simple with no renames	104
Renames required	0
ID to FN warning	77
Name Resolution Change	0
Unassigned IDs	10
Total number of opportunities	191

Table 6.1 Results from inlining all the calls to scripts

6.2 Inlining Functions

RQ1 For inlining functions, we counted each function call of form `[output]=g(input);` where the target was not a MATLAB builtin as an inlining opportunity. We measured:

– **RQ2**, the number of simple cases , and cases with renames.

– **RQ4**, the number of cases where the process failed with some exceptions. For this refactoring there weren't any cases where name resolution changes (*NameResolutionChangeException*) or an ID that is not definitely assigned turns to VAR (*IDNotDefAssignedException*) .

For **RQ3**, there are no situations where user intervention is needed. In this algorithm, it will either succeed or fail.

As indicated in *Table 6.2*, there were 2879 call sites, and all could be successfully inlined. 527 of those were the simple case where no renaming was required. All of the remaining cases could be handled by renaming. To answer **RQ5** we also measured the number of new statements that were added and the number of times these statements were removed. For the simple case (527 call sites) there were 1456 new statements (on average fewer than 3 statements) added to the code for assigning input and output arguments; Of

Inlining result	# Number of call sites
Simple	527
Renames required	2352
Name Resolution Change	0
Conflicting IDs	0
Total number of opportunities	2879

Table 6.2 Results from inlining all the calls to functions

those 1456 statements copy propagation could remove 896 statements leaving only about 1 added statement on average.

6.3 Converting Scripts to Functions

To measure **RQ1** for converting scripts to functions, each script is considered a candidate. Answers to **RQ2**, **RQ3** and **R4** are available in *Table 6.3*. In particular the table shows the number of: simple cases where no user intervention was necessary (Simple); times that Kind result for some identifiers changed from ID to more specialized Kind FN; cases where there is a possible change in the name resolution; cases where a script is called from other scripts and as a result there isn't enough context information available; times where the input arguments don't match at every call site; cases where some of the IDs couldn't be resolved to either VAR or FN; and cases where the resulting function had conflicting Kinds. It is important to note that all of those 705 cases where there were unresolved IDs were inside scripts that weren't called inside the project. These scripts were actually single file projects that were meant to be used in other projects with some variables set before they get called. Aside from these cases, the vast majority of the remaining cases are successfully refactored, making this a very useful refactoring for cleaning up MATLAB programs that use scripts.

To answer **RQ5** we measured the number of variables that have to be passed as parameters to the created functions. A large number of input and output parameters can clutter the code. So the function should only contain the necessary parameters. For those scripts that were called at least once the number of inputs range between 0 to 5 with the average

Conversion result	# Scripts
Simple	201
Warnings for IDs changed to FNs	1294
Name Resolution Change	0
Unresolved IDs	705
Call from script	148
Input Arguments mismatch	1
Unresolved Kind Conflicts	0
Total number of opportunities	2349

Table 6.3 Results from converting scripts to functions

of 1 and the number of outputs range between 0 to 12 with the average of 1.1. This shows that the algorithm is fairly efficient in choosing a minimal set of parameters.

6.4 Extract Function

Source code usually has sections that are more suitable to be extracted as a separate function as their data and control flows are less interwoven with other sections of the code. Programmers make this decision on which sequence of statements makes more sense to be extracted as a new function. Since we wanted to automate the evaluation, we decided to use a heuristic to find a suitable region for a new function.

The region starts at the beginning of the function and ends at one of the statements in the outermost nesting of the function. Our algorithm looks for functions with at least 7 statements in the outermost level. This gives us some flexibility in the choices for the region. Since we want the region to contain some reasonable amount of code, we pass the first few statements until the region contains at least 30 AST nodes. We don't want to move all the body of the original function to the new function either. So we stop the search when we reach there are less than 30 AST nodes left in the original function outside the region. Among these choices, we find the one that will need the minimum number of input and output arguments. We only extract the region if the minimum number of arguments is less than 15. *Figure 6.1* shows these constraints.

We ran this algorithm over all the programs in our benchmarks. We found that out of

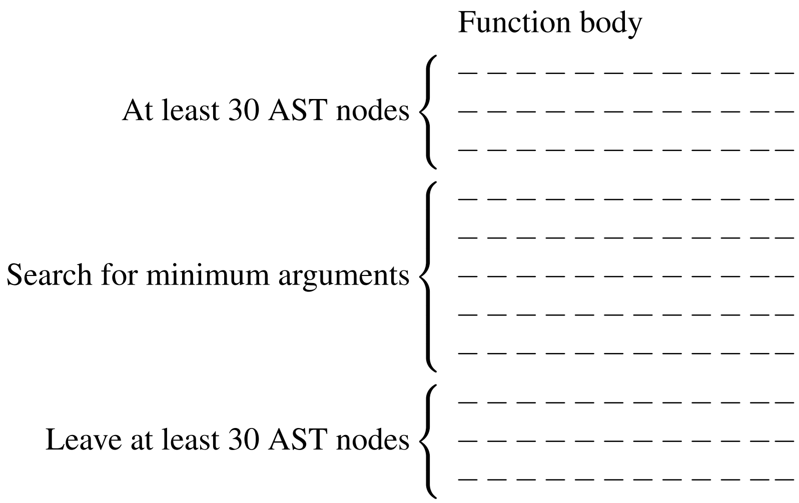


Figure 6.1 An example showing constraints used to select refactoring region

13438 functions (**RQ1**), we could break 6536 functions into smaller ones (**RQ2**). The average number of arguments to the newly created functions were 2.8 (**RQ5**). This means that the selection algorithm was effective in selecting regions with minimal inter-dependency. *Figure 6.2* shows the distribution of the number of arguments among these 6546 functions. The average number of nodes in the new functions was 105.6 compared to 476.5 nodes in the original functions.

RQ3 and **RQ4**: There were no exceptions during the execution and user validation is not necessary for this algorithm.

Figure 6.3 shows an example of running our heuristic on one of the functions from our benchmark set.

6.5 Replacing `feval`

There were 23 calls to `feval` with a string literal argument as target and all of them could be converted to direct function calls. These are the same cases we found using McBench.

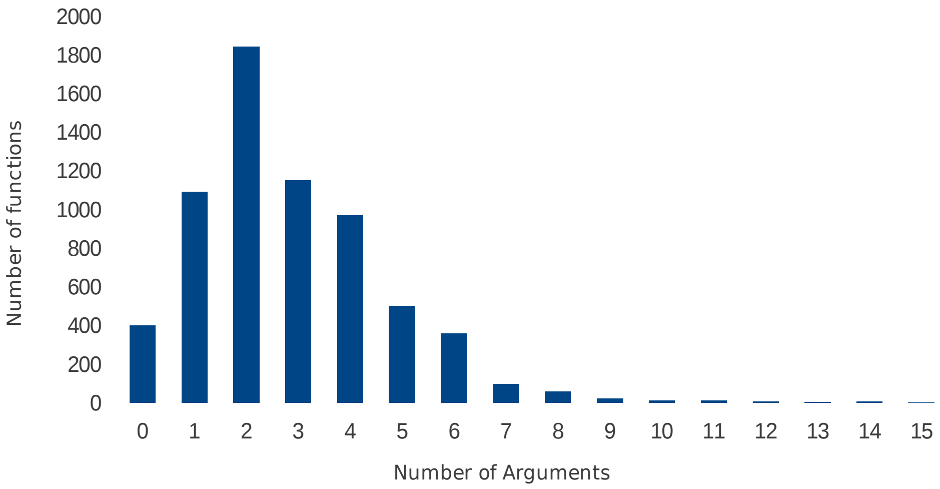


Figure 6.2 Distribution of number of arguments for the new functions.

6.6 Related Work

There is a wide variety of work on factoring covering a large number of programming languages. In particular, there is a considerable body of work on automatic refactoring for statically typed languages such as Java with quite well developed and rigorous approaches for specifying correct refactorings. Similar to [SdM10] we reuse microrefactorings such as variable rename and return elimination to make the specification and implementation of more complex refactorings simpler. In their approach they also extend the Java language to make the microrefactorings even simpler Our approach for refactoring MATLAB has similar aims in that we want to precisely state that conditions under which a refactoring is semantics-preserving. However, our approach is probably not as elegant as the recent Java approaches. We think this stems at least partly from the language we are targeting - MATLAB semantics are not as well defined, nor as regular as modern languages like Java.

There are also interesting approaches for other languages including Erlang, Fortran, Haskell[LRT03, Lee11] and Javascript, all of which present special benefits and challenges, just as our approach has special benefits and challenges for MATLAB. Our `feval` refactoring is similar to Tidier[SA09] refactoring that turns `apply(Func, [args])` in Erlang

```
function [zipfilename] = exportToZip(funcname, zipfilename)
  if (~iscell(funcname))
    funcname = {funcname};
  end
  if isempty(funcname)
    error('No function names specified');
  end
  if (~iscellstr(funcname))
    error('Function names must be strings');
  end
  req = cell(size(funcname));
  for i = (1 : numel(funcname))
    req{i} = mydepfun(funcname{i}, 1);
  end
  req = vertcat(req{:});
  req = unique(req);
  d = i_root_directory(req);
  n = numel(d);
  for i = (1 : numel(req))
    req{i} = req{i}(((n + 1) : end));
  end
  zip(zipfilename, req, d);
  fprintf(1, 'Created %s with %d entries\n', zipfilename, numel(req));

end
```

(a) Original function

```
function [zipfilename] = exportToZip(funcname, zipfilename)
  [zipfilename, req] = TMPNAME(funcname, zipfilename)
  req = vertcat(req{:});
  req = unique(req);
  d = i_root_directory(req);
  n = numel(d);
  for i = (1 : numel(req))
    req{i} = req{i}(((n + 1) : end));
  end
  zip(zipfilename, req, d);
  fprintf(1, 'Created %s with %d entries\n', zipfilename, numel(req));
end

function [zipfilename, req] = TMPNAME(funcname, zipfilename)
  if (~iscell(funcname))
    funcname = {funcname};
  end
  if isempty(funcname)
    error('No function names specified');
  end
  if (~iscellstr(funcname))
    error('Function names must be strings');
  end
  req = cell(size(funcname));
  for i = (1 : numel(funcname))
    req{i} = mydepfun(funcname{i}, 1);
  end
end
```

(b) After extraction

Figure 6.3 Result of running the *Extract Function* heuristic to split a function to two parts

to the simpler remote function call `Func(args)` when the last argument is a list of elements with statically known types.

Most related to our work is the work on JavaScript[FMM+11]. JavaScript has similarities with our work in that both JavaScript and MATLAB have some dynamic features which pose challenges for automated refactoring. Some of our goals are also similar, in that both approaches suggest some language-specific refactorings that help clean up the code.

Our approach shares an important similarity with the Fortran refactoring work. Overbey et. al. [ONJ09, OJ09] point out the benefits of refactoring for languages that have evolved over time. This is also one our main motivations for refactoring MATLAB. Although the specific refactorings are quite different, the motivation and the applicability of our approaches is very similar.

We are not aware of any refactoring work for MATLAB, but there is one related paper on source-level transformation for MATLAB [MP99]. In this work the authors show that a variety of source-level transformations can have important performance benefits. These transformations go beyond the typical loop transformations and capture MATLAB-specific behaviour such as converting loops to calls to libraries and restructuring loops to avoid incremental array growth. Automating these transformations would be an interesting next step, and our foundational analyses and refactorings should aid in that process.

Chapter 7
Conclusions and Future Work

In this thesis we have identified an important domain for refactoring, MATLAB programs. Millions of scientists, engineers and researchers use MATLAB to develop their applications, but no tools are available to support refactoring their programs. This means that it is difficult for the programmers to improve upon old code which use out-of-date language constructs or to restructure their initial prototype code to a state in which it can be distributed. Such refactoring tools are especially needed because MATLAB programmers are often not professional programmers and the frequently proceed by reusing old code, and programming using program features that are not ideal for their purposes.

To address this new refactoring domain we developed McBench tool to make it easier to understand how the MATLAB programmers write their code and we made the tool extensible so that it can be reused for other research projects. We have also developed a set of refactoring transformations for functions and scripts, including function and script inlining, converting scripts to functions, and eliminating simple cases of `feval`. For each refactoring we established a procedure which defined both the transformation and the conditions which must be verified to ensure that the refactoring is semantics-preserving. In particular, we emphasized that both the kinds of identifiers and the function lookup semantics must be considered when deciding if a refactoring can be safely applied or not.

We have implemented all of the refactorings presented in the paper using our ***Mc*LAB** compiler toolkit, and we applied the refactorings to a large number of MATLAB applications. Our results show that, on this benchmark set, the refactorings are applicable.

McBench is already being used in *Mc*LAB to understand how the programmers use different aspects of MATLAB language. An excellent opportunity for future work is to develop a XPath query builder that would make it even easier to use McBench. New annotations for type information and points-to analysis will make it possible to write completely new kinds of queries.

There is an excellent opportunity to build upon our refactoring tools to write new refactorings including performance enhancing refactorings and refactorings to enable a more effective translation of MATLAB to Fortran. The script to function refactoring is planned to be used in *Mc*LAB Fortran compiler.

McBench can be repurposed as a MATLAB lint tool. A collection of code smell detection queries can show the user places that can benefit from refactoring. An inline comment viewer to complement the code highlighter can precisely show the code section along with a description of the reasons to avoid specified code patterns. This can help MATLAB programmers to improve their codes and learn more about coding best practices.

List of AST nodes

Annotation	AssignStmt	Attribute	BreakStmt
CellArrayExpr	CellIndexExpr	ClassDef	ClassEvents
DefaultCaseBlock	CompilationUnits	ContinueStmt	ColonExpr
DotExpr	ElseBlock	EndExpr	Event
ExpandedAnnotation	ExprStmt	FPLiteralExpr	ForStmt
FunctionHandleExpr	FunctionDecl	Function	FunctionList
GlobalStmt	IfBlock	IfStmt	InputParamList
IntLiteralExpr	LambdaExpr	MatrixExpr	Methods
NestedFunctionList	NameExpr	Name	OutputParamList
ParamDeclList	ParameterizedExpr	PersistentStmt	Properties
Property	PropertyAccess	RangeExpr	ReturnStmt
ShellCommandStmt	Script	Row	Signature
ArrayTransposeExpr	StringLiteralExpr	SuperClass	StmtList
SwitchCaseBlock	SwitchStmt	TryStmt	VariableDecl
WhileStmt	AndExpr	EDivExpr	ELDivExpr
EPowExpr	EQExpr	ETimesExpr	GEExpr
GTExpr	LEExpr	LTExpr	MDivExpr
MinusExpr	MLDivExpr	MPowExpr	MTimesExpr
ShortCircuitAndExpr	OrExpr	PlusExpr	NEExpr
ShortCircuitOrExpr	MTransposeExpr	UPlusExpr	NotExpr
UMinusExpr			

Table A.1 All the node types that *Mc*LAB XML output might contain.

Bibliography

[BBC+07] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Simeon. Xml path language (xpath) 2.0, world wide web consortium recommendation rec-xpath20-20070123, 2007.

[CFR+01] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. Xquery: A query language for xml. working draft (feb.), world wide web consortium, 2001.

[Dev92] P.T. Devanbu. Genoa - a customizable, language- and front-end independent code analyzer. In *Software Engineering, 1992. International Conference on*, 1992, pages 307 –317.

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

[Doh11] Jesse Doherty. Mcsaf: An extensible static analysis framework for the matlab language. Master's thesis, McGill University, September 2011.

[FMM+11] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for JavaScript. In *In Proceedings of OOPSLA 2011*, 2011.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GAM96] W.G. Griswold, D.C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, mar 1996, pages 144 –153.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. Ph.D. thesis, University of Washington, 1991.

[JB00] Greg J and Badros. Javaml: a markup language for java source code. *Computer Networks*, 33(1-6):159 – 177, 2000.

[Lee11] Da Young Lee. A case study on refactoring in Haskell programs. In *Proceeding of the 33rd international conference on Software engineering*, Waikiki, Honolulu, HI, USA, 2011, ICSE '11, pages 1164–1166. ACM, New York, NY, USA.

[LRT03] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Uppsala, Sweden, 2003, Haskell '03, pages 27–38. ACM, New York, NY, USA.

[MP99] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *Proceedings of the 2nd conference on Domain-specific languages*, Austin, Texas, United States, 1999, DSL '99, pages 53–65. ACM, New York, NY, USA.

[MP09] English M. and McCreanor P. Exploring the differing usages of programming language features in systems developed in c++ and java. Limerick, Ireland, 2009, PPIG '09.

[OJ09] Jeffrey L. Overbey and Ralph E. Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. *SIGPLAN Not.*, 44:493–502, October 2009.

[ONJ09] Jeffrey L. Overbey, Stas Negara, and Ralph E. Johnson. Refactoring and the evolution of Fortran. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009, SECSE '09, pages 28–34. IEEE Computer Society, Washington, DC, USA.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[PKPZ11] O. Panchenko, J. Karstens, H. Plattner, and A. Zeier. Precise and scalable querying of syntactical source code patterns using sample code snippets and a database. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, june 2011, pages 41 –50.

[PP94] S. Paul and A. Prakash. A framework for source code search using program patterns. *Software Engineering, IEEE Transactions on*, 20(6):463 –475, jun 1994.

[SA09] Konstantinos Sagonas and Thanassis Avgerinos. Automatic refactoring of Erlang programs. In *Proceedings of the Eleventh International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, September 2009, pages 13–24. ACM, New York, NY, USA.

[SdM10] Max Schaefer and Oege de Moor. Specifying and implementing refactorings. *SIGPLAN Not.*, 45:286–301, October 2010.