McGill University
School of Computer Science
Sable Research Group

Refactoring MATLAB

Sable Technical Report No. 2011-2

Soroush Radpour and Laurie Hendren

October 15, 2011

Contents

List of Figures

List of Tables

Refactoring MATLAB

Soroush Radpour and Laurie Hendren

October 15, 2011

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incrementally using a mixture of MATLAB scripts and functions and frequently build upon existing code which may use outdated features.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. Furthermore, correct refactoring of MATLAB is quite challenging because of its non-standard rules for binding identifiers. Even simple refactorings are non-trivial.

This paper presents the important challenges of refactoring MATLAB along with automated techniques to handle a collection of refactorings for MATLAB functions and scripts including: function and script inlining, converting scripts to functions, and converting dynamic *feval* calls to static function calls. The refactorings have been implemented using the McLAB compiler framework, and an evaluation is given on a large set of MATLAB benchmarks which demonstrates the effectiveness of our approach.

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 [1] and Griswold [2], and the well known book by Fowler [3]. 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`).

As we have collected and studied a large body of MATLAB programs, we have found that the code could benefit from refactoring for several reasons. First, the MATLAB language has evolved over

[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`.)

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, code cruft, obsolete syntax and new language features complicates this reuse. Since MATLAB doesn't 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[4] (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` also complicate programs and are often used inappropriately. Thus, MATLAB-specific refactorings, which eliminate these features, are also very useful.

In this paper 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. Finally, we present a refactoring to replace calls of `feval` to direct function calls.

We have implemented our refactoring transformations in our McLAB compiler framework[5], and evaluated the refactorings on a collection of 3057 MATLAB programs. We found that the vast majority of refactoring opportunities could be handled.

The main contributions of this paper are:

- Identifying a need for refactoring tools for MATLAB and the key static properties that must be checked for such refactorings.
- Introducing a family of behavior-preserving refactorings for MATLAB functions and scripts.
- An implementation of these refactoring transformations in McLAB.
- An evaluation of the refactorings on a large set of publicly-available MATLAB programs.

The remainder of this paper is structured as follows. In Section 2 we provide some motivating examples and background about kind analysis and name lookup. In Section 3 we present inlining refactorings for inlining functions and scripts. Section 4 describes the refactoring for converting scripts to functions and Section 5 presents a refactoring to replace calls to `feval` with direct function

```
1 function r = MultiplyCompatible(n, m)
2   ndims = size(n);
3   mdims = size(m);
4   r = ((length(ndims)==2) && ...
5      (length(mdims)==2) && ...
6      (ndims(2)==mdims(1)));
7 end
8
9 % kinds ...
10 % VAR: r,n,m,ndims,mdims,r
11 % FN: size, length
```

Listing 1: Function stored in `MultiplyCompatible.m`

calls. Section 6 evaluates the refactorings on our benchmark set, Section 7 gives related work and Section 8 concludes.

2 Background and Motivating Examples

In this section we introduce the key features of Matlab and to give some motivating examples for the kinds of refactorings that are useful and the Matlab-specific issues that must be considered.

2.1 MATLAB functions and scripts

Matlab programs consist of a collection of functions and scripts. Listing 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 the same as the number of rows of m.[2]

In general, Matlab functions have input parameters (n and m in Listing 1) and may also have output parameters (r in Listing 1). Parameters obey call-by-value semantics where semantically a copy of each input and output parameter is made.[3]

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.

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 2.3.

Matlab scripts are even more unstructured than functions. Scripts are simply a sequence of

[2]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 paper.

[3]Actual implementations of Matlab optimize this using either lazy copying using reference counts, or static analyses to insert copies only where necessary[6].

```
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: Script stored in `SMultiplyCompatible.m`

statements that can be invoked. For example, consider the script in Listing 2, which looks similar to the body of the function in Listing 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.[4] For example, if `SMultiplyCompatible` is invoked from a workspace which contains a variable **size**, then lines 1 and 2 of Listing 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.

2.2 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`.

[4]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 becomes the current workspace.

```
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 3: A function calling a script

2.3 Impact of kinds on refactoring

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 paper, we have indicated the results of the kind analysis as comments at the end of each function/script definition.

Kind analysis 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; or 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. 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. Let us consider the example in Listing 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) care must be taken with the identifier ndims. In MultiplyFn the identifier ndims refers to a function and will have kind Fn, whereas in MultiplyCompatible ndims is assigned to, and will have kind Var.

If we inlined without appropriately renaming ndims, as shown in Listing 4, we would introduce a kind error because the inlined source would use ndims in a conflicting manner. Thus, at compile-time a conflicting kind error would be triggered on line 8. We return to this example in Section 3.1.

2.4 Impact of function 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

```
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 4: Example of kind error due to script inlining

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.

2.5 Refactoring Scripts

Refactoring scripts by inlining them or converting them to functions is particularly beneficial, as scripts lead to non-modular programs, and scripts are hard to optimize for Just-In-Time compilers. However, scripts present more challenges than functions. There are two reasons.

First, the kind analysis for scripts leads to many identifiers being assigned a kind of Id. For example, note that identifiers **size** and **length** in the script in Listing 2 both have kind Id, whereas in the equivalent function in Listing 1 they both have kind Fn. Thus, if code from a script is inlined into a function or if a script is converted to a function, the kind analysis will often give a more specialized kind of Var or Fn, rather than Id. Each of these cases must be checked to ensure that this more specialized kind information does not change the meaning of the program.

Second, the lookup of function calls within the body of a script depends on the directory of the last-called function. Thus, if we convert a script s to a function s the last-called function changes. Before conversion the last-called function was the caller of s and after conversion it is s itself.

3 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.

3.1 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 4 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 [4] 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.

3.1.1 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.

3.1.2 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 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 a kind of 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 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.

3.1.3 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 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.

3.2 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**);.[5] Before applying this refactoring, we assume assume that all calls to scripts in f or g have already been inlined or converted to a function call using our other refactorings. This allows us to reason about definitions, uses and liveliness intra-procedurally.

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 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 $p_i = e_i$ is added for each output parameter p_i. Figure 1(a) shows an example function MultiplyFn2 and Figure 1(b) shows the initial inlining of the call to "isCompatible = MultiplyCompatible(a,b)".

We also handle inlining for functions with variable numbers of input or output arguments. MATLAB functions may define variable input arguments by by using "`varargin`" as the name of the last input argument. Similarly for variable output arguments the name of the last output parameter should be "`varargout`".

Our inlining refactoring handles these cases as follows. In the case where the function g has "`varargin`" as the last argument, the last assignment in the inlined is created with the rest of provided arguments $varargin = \{e_{ninput}, ..., e_{minput}\}$; instead where $ninput$ is number of arguments in g and $minput$ is the number of provided input arguments. In case there is an identifier with name $varargin$ in f, the variable $varargin$ and all its uses will get renamed. The solution is similar for $varargout$, if the last output argument is $varargout$ the statement is created as $varargout = e_{noutput}, ..., e_{moutput}$ instead and renaming is performed if necessary.

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 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.

[5]In MATLAB programs many function calls are not in this form directly, but are embedded in some more complex expression. Thus, we also implemented a transformation that can safely simplify a complex expression so that the call is extracted with this form.

```
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) original function

```
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) initial inlined code with kind error

```
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
```

(c) inlined with renamings

```
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
```

(d) spurious copies removed

Figure 1: Example of Function Inlining

- 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 1(c) 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's 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 1(d) shows the result after copy elimination for our running example. Note that the copies to a, b and isCompatible have been removed.

4 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*. These are standard analyses which we have implemented in a way that enables our refactoring.

In our implementation of reaching definition analysis, every identifier is initialized to be have a special reaching definition of "`undef`". This means that if "`undef`" is not in the reaching definition set for an identifier at some program point p, then this identifier is definitely assigned in all the paths to p. Further, if the reaching definition of an identifier only contains "`undef`", the variable is not assigned to on any paths. Calls to scripts can change reaching definition and liveliness results so we look into the called scripts' body during the analyses.

Our liveness analysis is intra-procedural and handles global and persistent variables in a conservative manner. We safely approximate that all all MATLAB global variables are always live and persistent variables are live at the end of function with which they are associated. Also, in functions that have nested functions all the variables that are also used in nested functions are kept alive for simplicity.

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 $scriptLives(s) \cap callAssigned(c_i)$ and $output_i$ is defined as $scriptMayAssigned(s) \cap callLives(c_i)$. If any identifier in the $output_i$ is not in $scriptDefAssigned(s)$ an *OutputNot-DefinitelyAssignedException* is raised.

- The set $lookup_i$ is defined as: $\{\langle n : ResolveName(n)\rangle | n \in identifers(f) \wedge kind(n) \in \{\text{ID}, \text{FN}\}\}$

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 *InputArgsNot-MatchingException* 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 "~" 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's 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 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 "$\sim$" for those arguments that are not. The right hand side of the assignment is formed by simply a call to f with all the input arguments.

5 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, extracted from one of our benchmarks.[6] 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.

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.

6 Evaluation

In this section we present our evaluation of the refactoring algorithms on a large set of open-source MATLAB libraries and applications.

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

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% this program calculates and plots the wave-vector
% diagram (i.e.%photonic bands at constant frequency)
% ...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% the package contains the following programs:
%%%   pwem2Db.m - main program
%%%   epsgg.m - routine for calculating the matrix
%%%            of Fourier coeff of dielectric fn ...
clear all
tic
omega=0.45; % normalized frequency "a/lambda"
r=0.43; % radius of cylindrical holes
na=1; nb=3.45; % refractive indices
...
%%% matrix of Fourier coefficients
eps1 = feval ('epsgg',r,na,nb,b1,b2,N1,N2);
...
S=2.5; % point size for scatter plot
for j=1:length(BZx)
    %%% diagonal matrices with elements
    %%%      (kx+Gx) si (ky+Gy)
    [kGx, kGy] = feval('kvect2',BZx(j),BZy(j),
                              b1,b2,N1,N2);
    [P, beta]=feval('oblic_eigs',omega,kGx,kGy,
                              eps1,N);
    ...
end
```

Listing 5: Extracts from a script which uses `feval` (... corresponds to elided code)

6.1 Research Questions

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

RQ1 How many refatoring opportunities are available?

RQ2 How many times the algorithm could complete without any user validation?

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?

6.2 Experimental Setup and Benchmarks

In order to experiment with our analyses we gathered a large number of MATLAB projects.[7] The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing.

[7]Benchmarks were obtained from individual contributors plus projects from `http://www.mathworks.com/-matlabcentral/fileexchange`, `http://people.sc.fsu.edu/~jburkardt/m_src/m_src.html`, `http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/` and `http://www.mathtools.net/MATLAB/`. This is the same set of projects that are used in [4].

We analyzed 3057 projects composed of 11698 functions and 2349 scripts. The projects vary in size between 283 files in one project to a single file. A summary of the size distribution of the benchmarks is given in Table 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 which may mean that the average code quality is higher than many less used projects.

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 1: Distribution of size of the benchmarks

6.3 Inlining Scripts

As shown in Table 2, 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 2: Results from inlining all the calls to scripts

6.4 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, cases with variable arguments, 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.

Inlining result	# Number of call sites
Simple	527
Variable arguments	125
Renames required	2352
Name Resolution Change	0
Conflicting IDs	0
Total number of opportunities	3004

Table 3: Results from inlining all the calls to functions

As indicated in Table 3, there were 3004 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 with either our technique for handling varargs or 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 those 1456 statements copy propagation could remove 896 statements leaving only about 1 added statement on average.

6.5 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 4. 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's 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 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.

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 4: Results from converting scripts to functions

6.6 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.

6.7 Threats to Validity

The validity of each refactoring depends on the validity of the static analyses on which they are built. The kind and name analyses do not handle dynamic calls to `cd`, and `eval` is not handled by the liveness or reaching definition analysis. Further, as we pointed out earlier, renaming variables is only correct if that variable is not accessed via a dynamic feature.

7 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[7, 8, 9]. Our approach for refactoring MATLAB has similar aims in that we want to precisely state that conditions under which a refactoring is semantics-preserving.

There are also interesting approaches for other languages including Erlang[10, 11], Fortran[12, 13, 14], Haskell[15, 16] and Javascript[17], all of which present special benefits and challenges, just as our approach has special benefits and challenges for MATLAB. The work on JavaScript[17] has similarities with our work in that both JavaScript and MATLAB have some “nasty” 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. [13, 14] 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 [18]. 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.

8 Conclusion

In this paper 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.

To address this new refactoring domain we have 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 McLab compiler toolkit, and we applied the refactorings to a large number of Matlab applications. Our results show that, on this benchmark set, the refactorings can be effectively applied. We plan to continue our work, adding more refactorings, including performance enhancing refactorings and refactorings to enable a more effective translation of Matlab to Fortran.

Acknowledgments

This work has been supported by NSERC (Canada) and the Leverhulme Trust (UK). We would also like to give special acknowledgment to Frank Tip and Max Shäfer for introducing us to the world of refactoring and helping to define the direction of this work.

References

[1] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[2] W. G. Griswold, "Program restructuring as an aid to software maintenance," Ph.D. thesis, University of Washington, 1991.

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

[4] J. Doherty, L. Hendren, and S. Radpour, "Kind analysis for MATLAB," in *In Proceedings of OOPSLA 2011*, 2011.

[5] "McLab," http://www.sable.mcgill.ca/mclab/.

[6] N. Lameed and L. J. Hendren, "Staged static techniques to efficiently implement array copy semantics in a matlab jit compiler," in *CC*, ser. Lecture Notes in Computer Science, J. Knoop, Ed., vol. 6601. Springer, 2011, pp. 22–41.

[7] M. Schaefer and O. de Moor, "Specifying and implementing refactorings," *SIGPLAN Not.*, vol. 45, pp. 286–301, October 2010.

[8] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. D. Sutter, "Refactoring using type constraints," *ACM Trans. Program. Lang. Syst.*, vol. 33, pp. 9:1–9:47, May 2011.

[9] M. Schäfer, A. Theis, F. Steimann, and F. Tip, "A comprehensive approach to naming and accessibility in refactoring Java programs," IBM, Tech. Rep. IBM Research Report RC25201 (W1108-027), August 2011.

[10] K. Sagonas and T. Avgerinos, "Automatic refactoring of Erlang programs," in *Proceedings of the Eleventh International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming.* New York, NY, USA: ACM, Sep. 2009, pp. 13–24.

[11] H. Li, S. Thompson, and L. Lvei, "Refactoring erlang programs," in *In The Proceedings of 12th International Erlang/OTP User Conference*, 2006.

[12] J. Overbey, S. Xanthos, R. Johnson, and B. Foote, "Refactorings for Fortran and high-performance computing," in *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, ser. SE-HPCS '05. New York, NY, USA: ACM, 2005, pp. 37–39.

[13] J. L. Overbey, S. Negara, and R. E. Johnson, "Refactoring and the evolution of Fortran," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, ser. SECSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 28–34.

[14] J. L. Overbey and R. E. Johnson, "Regrowing a language: refactoring tools allow programming languages to evolve," *SIGPLAN Not.*, vol. 44, pp. 493–502, October 2009.

[15] H. Li, C. Reinke, and S. Thompson, "Tool support for refactoring functional programs," in *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, ser. Haskell '03. New York, NY, USA: ACM, 2003, pp. 27–38.

[16] D. Y. Lee, "A case study on refactoring in Haskell programs," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1164–1166.

[17] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, "Tool-supported refactoring for JavaScript," in *In Proceedings of OOPSLA 2011*, 2011.

[18] V. Menon and K. Pingali, "A case for source-level transformations in MATLAB," in *Proceedings of the 2nd conference on Domain-specific languages*, ser. DSL '99. New York, NY, USA: ACM, 1999, pp. 53–65.