Refactoring MATLAB

Soroush Radpour[1,2], Laurie Hendren[2], and Max Schäfer[3]

[1] Google, Inc.
soroush@google.com
[2] School of Computer Science, McGill University, Montreal, Canada
hendren@cs.mcgill.ca
[3] School of Computer Engineering, Nanyang Technological University, Singapore
schaefer@ntu.edu.sg

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incrementally using a mixture of MATLAB scripts and functions, and frequently build upon existing code which may use outdated features.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: converting scripts to functions, extracting functions, and converting dynamic function calls to static ones. 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 transforming a program in order to improve its internal structure without changing its external behavior. 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, and Oracle's NetBeans have integrated tool support for automating simple refactorings. However, the benefits of refactoring tools have not yet reached the millions of MATLAB programmers. Currently neither Mathworks' proprietary 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.[4] There are

[4] The most recent data from MathWorks shows one million MATLAB users in 2004, with the number doubling every 1.5 to 2 years; see www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf.

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 the years, incrementally introducing many valuable high-level features such as (nested) functions, packages and so on. However, MATLAB programmers often build upon code available online or examples from books, which often do not use the modern high-level features. Thus, although code reuse is an essential part of the MATLAB eco-system, code cruft, 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 an incremental style of programming that often results in relatively unstructured and non-modular code. 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.

Refactoring MATLAB presents new research challenges in two areas: (1) ensuring proper handling of MATLAB semantics; and (2) developing new MATLAB-specific refactorings. The semantics of MATLAB is quite different from other languages, thus even standard refactorings must be carefully defined. In particular, to ensure behavior preservation, refactoring tools have to verify that identifiers maintain their correct kind [4] (variable or function), and that their binding is not accidentally changed. MATLAB-specific refactorings include those which help programmers eliminate undesirable MATLAB features. For example, MATLAB scripts are a hybrid of macros and functions, and can lead to unstructured code that is hard to analyze and optimize. Thus, an automatic refactoring which can convert scripts to functions is a useful refactoring transformation which helps improve the structure of the code. Dynamic features like `feval` also complicate programs and are often used inappropriately. Thus, MATLAB-specific refactorings, which convert `feval` to more static constructs are also useful.

In this paper we introduce a family of automated refactorings aimed at restructuring functions and scripts, and calls to functions and scripts. We start with a refactoring for converting scripts into functions, which improves their reusability and modularity. Then we introduce the MATLAB version of the well-known EXTRACT FUNCTION refactoring that can be used to break up large functions into smaller parts. Finally, we briefly survey several other useful refactorings for inlining scripts and functions, and a refactoring to replace spurious uses of the dynamic `feval` feature with direct function calls.

We have implemented our refactoring transformations in our MCLAB compiler framework [5], and evaluated the refactorings on a collection of 3023 MATLAB programs. We found that the vast majority of refactoring opportunities could be handled with few spurious warnings.

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 refactorings for MATLAB functions and scripts.
- An implementation of these refactorings in MCLAB.

– An evaluation of the implementation 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 determining the kind of identifiers and the semantics of function lookup. Section 3 describes a refactoring for converting scripts to functions, Section 4 presents EXTRACT FUNCTION, and Section 5 briefly introduces several other refactorings. Section 6 evaluates the refactoring implementations on our benchmark set, Section 7 surveys related work, and Section 8 concludes.

2 Background and Motivating Example

In this section we introduce some key features of MATLAB, and we give a motivating example to demonstrate both a useful MATLAB refactoring and the sorts of MATLAB-specific issues that must be considered.

2.1 MATLAB scripts and functions

A MATLAB program consists of a collection of scripts and functions. A script is simply a sequence of MATLAB statements. For example, Figure 1(a) defines a script called `sumcos` which computes the sum of the cosine values of the numbers `i` to `n`. Although using scripts is not a good programming practice, they are very easy for MATLAB programmers to create. Typically, a programmer will experiment with a sequence of statements in the read-eval-print loop of the IDE and then copy and paste them into a file, which becomes the script.

A script is executed in the workspace from which it was called, either the main workspace, or the workspace of the calling function.[5] For example, Figure 1(b) shows function `ex1` calling script `sumcos`. When `sumcos` executes it reads the values of variables `i` and `n` from the workspace of function `ex1`, and writes the value of `s` into that same workspace. Clearly, scripts are highly non-modular, and do not have a well-defined interface. A programmer cannot easily determine the inputs and outputs of a script. Thus, a better programming practice would be to use functions.

Figure 1(d) shows the script `sumcos` refactored into an equivalent function. The body of the function is the same as the script, but now the output parameter `s` and the input parameters `i` and `n` are explicitly declared. As shown in Figure 1(c) and (f), in this case the refactored function produces the same result as the original script.[6]

In general, MATLAB functions may have multiple output and input arguments. However, not all input arguments need to be provided at a call, and not all returned

[5] Workspaces are MATLAB's version of lexical environments. 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 to the function call stack. A call to a function creates and pushes a new workspace, which becomes the current workspace.

[6] These results are snippets taken from an interactive session in the MATLAB read-eval-print loop. The "`>>`" prompt is followed by the expression to be evaluated. In Figure 1(c) this is a call to function `ex1`. The line after the prompt prints the result of the evaluation.

```
s = 0;
while i <= n
    s = s + cos(i);
    i = i + 1;
end
```

(a) script `sumcos.m`

```
function ex1( )
  i = 1;
  n = 5;
  sumcos;
  s
end
```

(b) calling `sumcos`

```
>> ex1
s = -1.2358
```

(c) result of call

```
function s = sumcosFN(i, n)
  s = 0;
  while i <= n
    s = s + cos(i);
    i = i + 1;
  end
end
```

(d) function `sumcosFN.m`

```
function ex1FN( )
  s = sumcosFN(1, 5)
end
```

(e) calling `sumcosFN`

```
>> ex1FN
s = -1.2358
```

(f) result of call

Fig. 1. Example script and function

values need to be used. Parameters obey call-by-value semantics where semantically a copy of each input and output parameter is made.[7]

2.2 Identifier kinds

MATLAB does not explicitly declare local variables, nor explicitly declare the types of any variables. Input and output arguments are explicitly declared as variables, whereas other variables are implicitly declared upon their first definition. For example, the assignment to `s` in the first line of Figure 1(d) implicitly also declares `s` to be a variable, and allocates space for that variable in the workspace of function `sumcosFN`.

It is important to note that it is not possible to syntactically distinguish between references to array elements and calls to functions. For example, so far we have assumed that the expression `cos(i)` is a call to function `cos`. However, it could equally well be an array reference referring to the ith element of array `cos`.

To illustrate, consider Figure 2(a), where `cos` is defined to be a five-element vector. The call to `sumcos` in this context actually just sums the elements of the vector, returning `15`. This is because the MATLAB semantics give a *kind* of ID (identifier) to most identifiers in scripts. The rule for looking up identifiers with kind ID at runtime is to first look in the current workspace to see if a variable of that name exists, and if so the identifier denotes that variable. If no such variable exists then the identifier is looked up as a function. Since the script `sumcos` is being executed in the workspace of function `ex2`, and there does exist a variable called `cos` in that workspace, the reference to `cos` refers to that variable, and not the library function for computing the cosine.

The identifier lookup semantics within functions is different. In the case of functions, each identifier is given a static kind at JIT compilation time; for details of this

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

```
function ex2( )
  cos = [1,2,3,4,5];
  i = 1;
  n = 5;
  sumcos;
  s
end
```

(a) calling script `sumcos`

```
>> ex2
s = 15
```

(b) result of call

```
function ex2FN( )
  cos = [1,2,3,4,5];
  s = sumcosFN(1, 5)
end
```

(c) calling function `sumcosFN`

```
>> ex2FN
s = -1.2358
```

(d) result of call

Fig. 2. Calling `sumcos` in a context where `cos` is a variable

process we refer to the literature [4]. In the case of the refactored function `sumcosFN`, identifiers `i`, `n` and `s` would be determined to have kind VAR (variables), and identifier `cos` would be given the kind FN (function). Thus, the reference to `cos` will always be to the function, and our transformed function `sumcosFN` may have a different meaning than the original script `sumcos`, as demonstrated by the different results in Figure 2(b) and (d).

From this example, it is clear that any MATLAB refactoring of scripts must take care not to change the meaning of identifiers, and in order to do this all of the calling contexts of the script must be taken into consideration.

2.3 MATLAB programs and function lookup

MATLAB programs are defined as directories of files. Each file `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'`.

Figure 3(a) shows an example of a file containing two functions. The primary function is `ex3` and will be visible to all functions and scripts defined in other files. This file also has a secondary function `cos`, which is an implementation of the cosine function using a Taylor's approximation. The important question in this example is which `cos` will be called from the script `sumcos`: the library implementation of `cos` or the Taylor's version of `cos` defined as a subfunction for `ex3`? The answer is that the lookup of a function call from within a script is done with respect to the calling function's environment. In this case the call to `cos` in script `sumcos` refers to the environment of

function `ex3`, which was the last called function. Thus, `cos` binds to the subfunction in `ex3`.

The transformed function `sumcosFN`, however, will not call the Taylor's version of `cos` since subfunctions are not visible to functions defined outside of the file. Thus, the results of running the original script and the transformed function are different. Clearly any MATLAB refactoring must take care that it does not change the binding of functions.

```
function ex3( )
  i = 1;
  n = 5;
  sumcos;
  s
end

function r = cos(x)
  r = 0;
  xsq = x*x;
  term = 1;
  for i = 1:1:10
    r = r + term;
    term = -term*xsq/((2*i-1)*(2*i));
  end
end
```

(a) `ex3.m` with primary and subfunction

```
>> ex3
s = -1.2359
```

(b) result of call

```
function ex3FN( )
  s = sumcosfn(1,5)
end

function r = cos(x)
  % same as above
  ...
end
```

(c) refactored `ex3.m`

```
>> ex3FN
s = -1.2358
```

(d) result of call

Fig. 3. Calling `sumcos` in a context where `cos` is defined as a subfunction.

In addition to subfunctions, MATLAB also uses the directory structure to organize functions, and this directory structure also impacts on function binding.

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 argument is `typename`.

Overall, the MATLAB 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.

In summary, a refactoring needs to ensure that identifier kinds do not change unexpectedly, and that function lookup remains the same.

3 Converting Scripts to Functions

In the previous section we have motivated the need for a refactoring that can convert scripts, which are non-modular, into equivalent functions which will help improve the overall structure of MATLAB programs. We also demonstrated that this refactoring is not as straightforward as one might think due to MATLAB's intricate kind assignment and function lookup rules.

In this section we provide an algorithm to refactor a script into a semantically equivalent function. 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.

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 the 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 on 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 liveness results so we look into the called scripts' body during the analyses. Global and persistent variables may be defined by function calls, so our analysis handles these conservatively by associating a special "`global_def`" or "`persistent_def`" with each variable declared as global or persistent. These definitions are never killed.

Our liveness analysis is intra-procedural, but also follows calls to scripts. The liveness analysis safely approximates global variables as always being live, and persistent variables as live at the end of the function they are associated with. Variables that are used in nested functions are also kept alive for simplicity.

Recall that the main difference between a script and a function is that a function has its own workspace and communicates with its caller via input and output arguments, while a script executes directly within the caller's workspace. Thus, 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) ensure that name binding will stay the same after conversion.

To determine arguments, the basic idea is that a variable needs to be made an input argument if it is live within the script and assigned at every call site; conversely, it needs to be an output argument if it is assigned within the script and live at some call site.

Notation	Meaning
DA_s	variables definitely assigned on every path through s
PA_s	variables possibly assigned on some path through s
$L_{<s}$	variables live immediately before s
$L_{>s}$	variables live immediately after s
$RD_s(x)$	reaching definitions for x immediately before s
$K_f(x)$	kind assigned to x inside script or function f; $K_f(x) = \lightning$ for inconsistent kind
$Lookup_f(x)$	look up identifier x in function f (subscript omitted where clear from context)

Fig. 4. Notation for auxiliary analysis results; s may be a sequence of statements, or the body of a function or script.

This intuition is made more precise in Algorithm 1, which uses the notations defined in Figure 4. To convert script s into a function, we first compute the set L of identifiers that are used before being defined in s, and that may refer to a variable (as opposed to a function); these are candidates for becoming input arguments.

Now we examine every call c to s. If the call occurs in a script s', we abort the refactoring: the lack of structure in scripts makes it all but impossible to determine appropriate sets of input and output arguments; the user can first convert s' into a function, and then attempt the refactoring again.

If c is in a function, we consider the set $DA_{<c}$ of variables definitely assigned at c. As far as call site c is concerned, the set I_c of input arguments should simply be the intersection of this set with L, the set of live variables at the beginning of s. Similarly, the set O_c of output arguments should contain all variables that are possibly assigned in s and that are live immediately after c. We also need to ensure that every output argument is definitely assigned somewhere in the script; otherwise the refactoring cannot go ahead. Finally, we compute a set $lookup_c$ capturing name binding information for functions at c, whose purpose will be explained below.

Next, we need to check that the set of input arguments I is consistent between call sites: if different call sites provide different input arguments, the refactoring cannot go ahead (line 13). For output arguments, on the other hand, no such precaution is required: if an output argument is unused at a particular call site, it can be ignored by binding it to the dummy "$\sim$" identifier. Thus the set of output parameters O is simply the union of output arguments at every call site.

We are now ready to build the function f using input arguments I, output arguments O, and the body of s.

Algorithm 1 SCRIPT TO FUNCTION

Require: script s
Ensure: s converted to function; all calls to s replaced with function calls

1: // preliminary definitions
2: $L \leftarrow \{x \mid x \in L_{<s} \wedge K_s(x) \in \{\text{VAR}, \text{ID}\}\}$
3: $C_s \leftarrow$ calls to s
4: // compute input and output arguments
5: **for all** calls $c \in C_s$ **do**
6: **if** c is in another script s' **then**
7: abort refactoring
8: $I_c \leftarrow DA_{<c} \cap L$ // input arguments
9: $O_c \leftarrow PA_s \cap L_{>c}$ // output arguments
10: **if** $O_c \not\subseteq DA_s$ **then**
11: abort refactoring
12: $lookup_c \leftarrow \{\langle n, Lookup(n)\rangle \mid n$ occurs in $s, K_s(n) \in \{\text{ID}, \text{FN}\}\}$ // binding information
13: **if** $\neg\forall c, c' \in C_s . I_c = I_{c'}$ **then**
14: abort refactoring
15: **else**
16: $I \leftarrow I_c$ for some call $c \in C_s$
17: $O \leftarrow \bigcup_{c \in C_s} O_c$
18: // construct new function
19: construct new function f with input arguments I and output arguments O
20: // check name binding and kinds
21: $lookup_f \leftarrow \{\langle n, Lookup(n)\rangle \mid n$ is identifier in f of kind ID or FN$\}$
22: **if** $\neg\forall c \in C_s . lookup_c = lookup_f$ **then**
23: abort refactoring
24: **for all** identifiers x in f **do**
25: **if** $K_f(x) = \text{ID}$ **then**
26: abort refactoring
27: **else if** $K_s(x) = \text{ID}$ and $K_f(x) = \text{FN}$ **then**
28: emit warning
29: **else if** $K_s(x) = \text{VAR}$ and $K_f(x) = \not\downarrow$ **then**
30: abort refactoring
31: replace calls to s with calls to f

As a final step, we need to check that name resolution and kind assignments have not changed.

The former is easy to do: we simply compute pairs $\langle n, Lookup(n)\rangle$ determining the binding of every identifier n with kind ID or FN in f, and check that these bindings agree with the bindings $lookup_c$ observed at the call sites.

To check kind preservation, we compare the kind $K_s(x)$ an identifier x had in s, with its kind $K_f(x)$ in the new function f. In general, identifiers of kind ID can remain so or turn into FN, and identifiers with kind VAR can cause a kind conflict.

If $K_f(x) = \text{ID}$, x may originally have been referring to a variable created dynamically in the calling function. Since functions do not share their caller's workspace, this cannot be achieved in a function, and the refactoring has to be aborted.

If x's kind changed from ID to FN, we emit a warning informing the user that the refactoring assumes x refers to a function, which is always the case unless a variable of the same name is created dynamically by `eval` or code loading.

Finally, if x was originally of kind VAR, but provokes a kind conflict in f, we need to abort the refactoring, since it is not clear which uses of the identifier were meant to refer to a function, and which to a variable.

If all checks pass, calls to s can be rewritten to function calls, passing in all input arguments in I and extracting output arguments from the result, discarding any output arguments not needed at a particular call site.

4 Extracting Functions

The EXTRACT FUNCTION refactoring makes it possible to split large functions into smaller ones to improve understandability and reusability. Across all our MATLAB benchmarks, we found that the average number of lines of code per function is 22.7; for comparison, this number is 5.4 for Java and 10.5 for C++ [7], which suggests that MATLAB functions tend to be fairly long and could benefit from extraction.

We first introduce the refactoring on an example before giving a precise specification of the extraction algorithm.

```
1  function printBest(names,
2                     grades)
3   bestGrade=-1; bestIdx=-1;
4   for i=1:length(grades)
5    if grades(i) > bestGrade
6     bestGrade=grades(i);
7     bestIdx=i;
8    end
9   end
10  if bestGrade == -1
11   return
12  end
13  disp(names{bestIdx})
14 end
```

(a) Original Function; extract lines 3–12

```
1  function printBest(names,
2                     grades)
3   RET=false;
4   bestGrade=-1; bestIdx=-1;
5   for i=1:length(grades)
6    if grades(i) > bestGrade
7     bestGrade=grades(i);
8     bestIdx=i;
9    end
10  end
11  if bestGrade == -1
12   RET=true;
13  end
14  if (~RET)
15   disp(names{bestIdx})
16  end
17 end
```

(b) After Return Elimination

Fig. 5. An example for EXTRACT FUNCTION

Figure 5(a) shows an example function that takes an array `names` containing the names of students, and an array `grades` containing their grades. On lines 3–12, it searches through `grades` to find the best grade, storing its index in local variable

`bestIdx`. If no best grade was found (because `grades` was empty or contained invalid data), the function returns to its caller; otherwise, the name of the student with the best grade is printed.

Assume that we want to extract the code for finding the best grade (lines 3–12) into a new function `findBest`. Note that the extraction region contains the return statement on line 11; if this statement were extracted into `findBest` unchanged, program semantics would change, since it would now only return from `findBest`, not from `printBest` any more. To avoid this, we first eliminate the return as shown in Figure 5(b) by introducing a flag `RET`. In general, return elimination requires a slightly more elaborate transformation than this, but it is still fairly straightforward and will not be described in detail here; the reader is referred to the first author's thesis for details [8].

Next, we need to determine which input and output arguments the extracted function should have. Reasoning similar to the previous section, we determine that `grades` should become an input argument, since it is live at the beginning of the extracted region and definitely assigned beforehand. Conversely, `bestIdx` should become an output argument, since it is assigned in the extracted region and live afterwards.

```
1  function [RET, bestIdx] =
2           findBest(grades)
3   bestGrade=-1; bestIdx=-1;
4   for i=1:length(grades)
5    if grades(i) > bestGrade
6     bestGrade=grades(i);
7     bestIdx=i;
8    end
9   end
10  if bestGrade == -1
11   RET=true;
12  end
13 end
14
15 function printBest(names,
16                     grades)
17  RET=false;
18  [RET, bestIdx] = ...
19    findBest(grades);
20  if (~RET)
21   disp(names{bestIdx})
22  end
23 end
```

(a) After extracting function, `RET` may be undefined after the call

```
1  function [RET, bestIdx] =
2           findBest(grades, RET)
3   bestGrade=-1; bestIdx=-1;
4   for i=1:length(grades)
5    if grades(i) > bestGrade
6     bestGrade=grades(i);
7     bestIdx=i;
8    end
9   end
10  if bestGrade == -1
11   RET=true;
12  end
13 end
14
15 function printBest(names,
16                     grades)
17  RET=false;
18  [RET, bestIdx] = ...
19    findBest(grades, RET);
20  if (~RET)
21   disp(names{bestIdx})
22  end
23 end
```

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

Fig. 6. Example for EXTRACT FUNCTION, continued

Similarly, RET should also become an output argument. Figure 6(a) shows the new function with these arguments. Note, however, that RET is not assigned on all code paths, so it may be undefined at the point where the extracted function returns, resulting in a runtime error. To avoid this, we have to also add RET to the list of input arguments, ensuring that it always has a value. This finally yields the correct extraction result, shown in Figure 6(b).

Algorithm 2 EXTRACT FUNCTION

Require: sequence s of contiguous statements in function f, name n
Ensure: s extracted into new function g with name n

if s contains top-level break or uses vararg syntax **then**
    abort refactoring
if s contains return statement **then**
    eliminate return statements in f
$I \leftarrow \{x \mid x \in L_{<s} \wedge RD_s(x) \not\subseteq \{\texttt{undef}, \texttt{global}\}\}$
$O \leftarrow PA_s \cap L_{>s}$
for all $x \in O \setminus DA_s$ **do**
    if $\texttt{undef} \notin RD_s(x)$ **then**
        $I \leftarrow I \cup \{x\}$
    else
        abort refactoring
if function with name n exists in same folder as f **then**
    abort refactoring
create new function g with name n, input arguments I, output arguments O
declare any globals used in s as globals in g
for all identifier x in g **do**
    if $K_g(x) \neq K_f(x)$ **then**
        abort refactoring
    else if $K_g(x) = \text{FN}$ and $Lookup_f(x) \neq Lookup_g(x)$ **then**
        abort refactoring
    else if $K_g(x) = \text{ID}$ **then**
        abort refactoring
replace s by call to g

Algorithm 2 shows how to extract a sequence s of contiguous statements in a function f into a new function named n, again using the notations from Figure 4.

We first check whether s contains a break statement that refers to a surrounding loop that is not part of the extraction region; if so, the refactoring is aborted. Similarly, if s refers to a variable argument list of f using "varargin" or "varargout", the refactoring is also aborted. Both of these cases would require quite extensive transformations, which we do not believe to be justified.

After eliminating return statements if necessary, we compute the set I of input arguments, and the set O of output arguments for the new function: every variable that is live immediately before the extraction region s and that has a non-trivial reaching

definition becomes an input argument; every variable that is potentially assigned in s and is live afterwards becomes an output argument.

Additionally, any output arguments that are not definitely assigned in s but are definitely assigned before (like `RET` in our example above) also become input arguments. We also check for the corner case of an output argument that is neither definitely assigned in s nor before s, which results in the refactoring being aborted.

Having established the sets of input and output arguments, we can now create the extracted function g, but we need to ensure that no function of this name exists already. We also need to declare any global variables used in s as global in g, as they would otherwise become local variables of g.

Finally, we need to check that name binding and kind assignments work out. First, we check that the kind of all identifiers in g is the same as before the extraction. Additionally, if there is any function reference that refers to a different declaration in the new function g than it did before, we need to abort the refactoring. Lastly, we ensure that no identifier has kind ID, as this may again lead to different name lookup results.

If all these checks pass, we can replace s by a call to the extracted function.

5 Other Refactorings

In addition to the SCRIPT TO FUNCTION and EXTRACT FUNCTION refactorings described in the previous sections, we have implemented several other refactorings that we briefly outline in this section.

Corresponding to EXTRACT FUNCTION, there are two inlining refactorings for inlining scripts and functions. While this does not usually improve code quality, inlining refactorings can play an important role as intermediate steps in larger refactorings.

When inlining a call to script or function g in function f, return statements in g first have to be eliminated in the same way as for EXTRACT FUNCTION. If g is a function, its local variables have to be renamed to avoid name clashes with like-named variables in f. After copying the body of g into f, we then have to verify that name bindings stay the same, and kind assignments either stay the same or at least do not affect name lookup. For details we refer to the first author's thesis [8].

Finally, we briefly discuss a very simple but surprisingly useful MATLAB-specific refactoring, ELIMINATE 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. Replacing `feval` by direct function calls where possible leads to cleaner and more efficient code.

Somewhat to our surprise, we found numerous cases where programmers used a constant function name in `feval`. For example, the code in Listing 1.1, which is extracted from one of our benchmarks, uses `feval` for all invocations of user defined functions (lines 2, 5 and 6), even though there is no apparent reason for doing so; all uses of `feval` can be replaced by direct function calls.

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

```
%%% matrix of Fourier coefficients
eps1 = feval ('epsgg',r,na,nb,b1,b2,N1,N2);
...
for j=1:length(BZx)
    [kGx, kGy] = feval('kvect2',BZx(j),BZy(j),b1,b2,N1,N2);
    [P, beta]=feval('oblic_eigs',omega,kGx,kGy,eps1,N);
    ...
end
```

Listing 1.1. Extracts from a script which uses `feval`

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

We now evaluate our implementation on a large set of MATLAB programs. While it would be desirable to evaluate correctness (i.e., behavior preservation) of our implementation, this is infeasible to do by hand due to the large number of subject programs. Automated testing of refactoring implementations is itself still a topic of research [9] and relies on automated test generation, which is not yet available for MATLAB. Instead, we aim to assess the usefulness of our refactorings and their implementation.

6.1 Evaluation Criteria

We evaluate every refactoring according to the following criteria:

EC1 How many refactoring opportunities are there?

EC2 Among all opportunities, how often can McLAB perform the refactoring without warnings or errors? How often is the user warned of possible behavior changes? How often is McLAB unable to complete the refactoring?

EC3 How invasive are the code changes?

6.2 Experimental Setup and Benchmarks

In order to experiment with our analyses we gathered a large number of MATLAB projects.[8] The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing. We analyzed 3023 projects composed of

[8] 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].

11698 function files, some with multiple functions, and 2380 scripts. The projects vary in size between 283 files in one project, and a single file in other cases. 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 for less popular projects.

Benchmark Category	Number of 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	3023

Table 1. Distribution of size of the benchmarks

6.3 Converting Scripts to Functions

We start by evaluating the SCRIPT TO FUNCTION refactoring presented in Section 3.

We consider every script a candidate for the refactoring (**EC1**), thus there are 2380 refactoring opportunities overall (note that some benchmarks only define functions). For criterion **EC2**, Table 2 summarizes the result of using MCLAB to convert all these scripts to functions: in 204 cases, the refactoring completed without warnings or errors; in 1312 cases, the refactoring succeeds with a warning about an identifier changing from kind ID to the more specialized kind FN. This is only a problem if the program defines a variable reflectively through `eval` or code loading, and the identifier in question is also a function on the path. This is unlikely and should be easy for the programmer to check. Finally, for 864 scripts the refactoring aborted because behavior preservation could not be guaranteed.

Further breaking down the causes of rejection, we see that in most cases the problem is an identifier of kind ID that cannot statically be resolved to a variable or a function. In all these cases, the script is the only script in a single file project; thus it arguably is not a very good target for conversion anyway. In some cases, the script was itself called from a script, which also leads to rejection (as mentioned in Section 3, this could be resolved by first converting the calling script to a function). Finally, in one case different invocations of the script lead to different input argument assignments.

To assess the invasiveness of the code changes (**EC3**), we measured the number of input and output arguments of the newly created functions. A large number of input and output parameters can clutter up the code, so it is important that the refactoring creates no spurious parameters. For those scripts that were called at least once, the number of inputs range between 0 and 5 with an average of 1, and the number of outputs range

Refactoring Outcome	Number of Scripts
Success	204
Success with Warning about ID changed to FNs	1312
Unresolved IDs	712
Call from script	151
Input arguments mismatch	1

Table 2. Results from converting scripts to functions

between 0 and 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

For function extraction, the number of refactoring opportunities is hard to measure, since it is not clear how to identify blocks of code for which function extraction makes sense.

In order to nevertheless be able to automatically evaluate a large number of function extraction refactorings, we employ a heuristic for identifying regions that are more or less independent in terms of control and data flow from the rest of a function.

We concentrate on regions starting at the beginning of a function, and comprising a sequence of top-level statements. We only consider functions with at least seven top-level statements; smaller functions are unlikely to benefit from extraction. Since we want the region to contain some reasonable amount of code, we include at least as many statements as it takes for the region to contain 30 AST nodes. We don't want to move all the body of the original function to the new function either, so we never extract the last 30 AST nodes in the function either. In between, we find the choice that will need the minimum number of input and output arguments, but only if that minimum number is less than 15. Figure 7 shows these constraints.

Out of 13438 functions overall, 6469 contain at least seven top-level statements and thus are interesting for extraction (**EC1**). Among these, we can successfully break 6214 functions (i.e., 96%) into smaller ones (**EC2**). The average number of arguments to the newly created functions was 2.8 (**EC3**), with most extracted functions having between one and three arguments. This means that the selection algorithm was effective in selecting regions with minimal inter-dependency. Figure 8 shows the distribution of the number of arguments among these 6214 functions.

In 48 cases the refactoring was rejected because a possibly undefined input argument was detected, and in 21 cases a possibly undefined output argument prevented the refactoring from going ahead.

6.5 Replacing `feval`

Finally, we evaluated the ELIMINATE FEVAL refactoring for converting trivial uses of `feval` into direct function calls. Of the 200 calls to `feval`, there were 23 uses of it with a string literal argument (**EC1**), and all of them could be eliminated successfully (**EC2**).

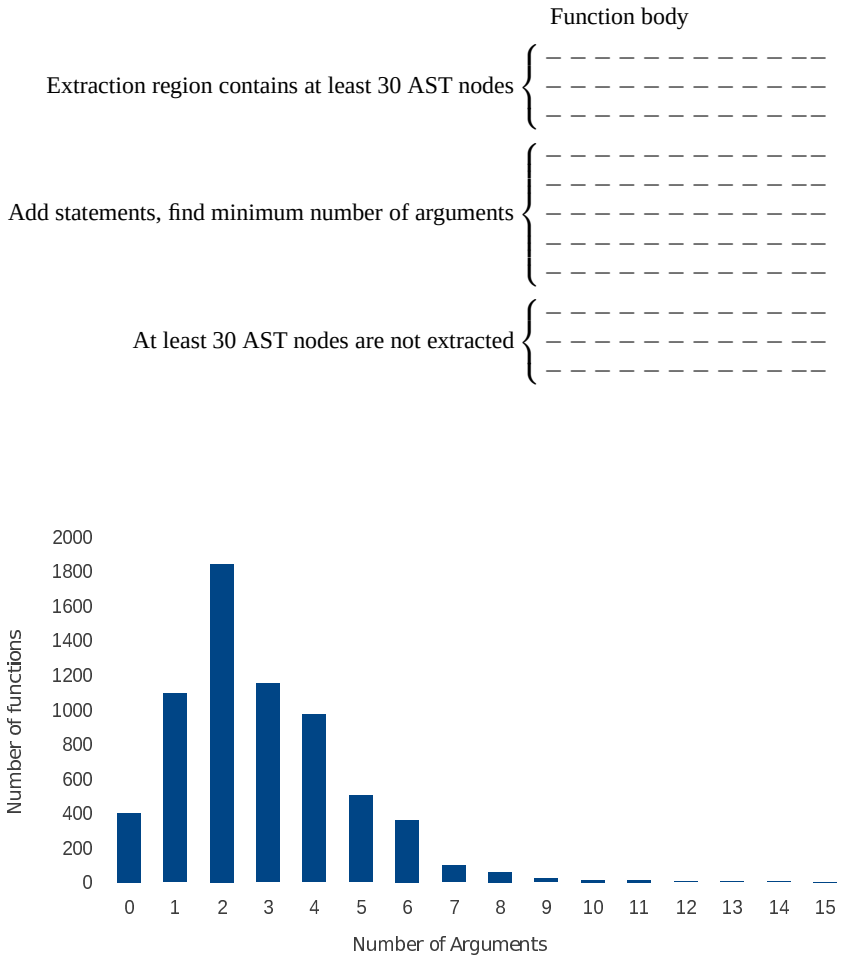

Fig. 8. Distribution of number of arguments for the new functions.

The transformation performed by this refactoring is very local, and in fact makes the code simpler (**EC3**).

6.6 Threats to Validity

There are several threats to validity for our evaluation.

First, our collection of benchmarks is extensive, but it may not be representative of other real-world MATLAB code. In particular, the percentage of rejected refactorings may be higher on code that makes heavy use of language features that are hard to analyze.

Second, our selection of refactoring opportunities is based on heuristics and may not be representative of actual refactorings that programmers may attempt. This is a

general problem with automatically evaluating refactoring implementations. Still, the low number of rejections gives some confidence that the implementation should be able to handle real-world use cases. A more realistic user study will have to wait until our implementation has been integrated with an IDE.

Finally, we have not checked whether the refactorings performed by our implementation are actually behavior preserving in every case; the large number of successful refactorings makes this impossible. We know of two edge cases where behavior may not be preserved: the kind and name analyses do not handle dynamic calls to `cd`, and `eval` is not handled by the liveness or reaching definition analysis. This is similar to how refactorings for Java do not handle reflection. One possibility would be for the refactoring engine to emit a warning to the user if a use of one of these features is detected, but we have not implemented this yet.

7 Related Work

There is a wide variety of work on refactoring covering a large number of programming languages. In particular, there is a considerable body of work on automated refactoring for statically typed languages such as Java with quite well developed and rigorous approaches for specifying correct refactorings[10,11,12]. However, these approaches intrinsically rely on the availability of rich compile-time information in the form of static types and a static name binding structure; thus they are not easily applicable to MATLAB, which provides neither.

Refactoring for dynamically typed languages has, in fact, a long history: the first ever refactoring tool, the Refactoring Browser [13], targetted the dynamically typed object-oriented language Smalltalk. However, the Refactoring Browser mostly concentrated on automating program transformation and performed relatively few static checks to ensure behavior preservation.

More recently, Feldthaus et al. [14] have presented a refactoring tool for JavaScript. They employ a pointer analyis to infer static information about the refactored program, thus making up for the lack of static types and declarations. Most of their refactorings have the goal of improving encapsulation and modularity, thus they are similar in scope to our proposed refactorings for MATLAB.

Even more closely related is recent work on refactoring support for Erlang. Like MATLAB, Erlang has evolved over time, adding new constructs for more modular and concise programming, and refactorings have been proposed that can help with upgrading existing code to make use of these new features. For instance, the Wrangler refactoring tool provides assistance for data and process refactoring [15], clone detection and elimination [16] and modularity maintenance [17].

Most recently, Wrangler has been extended with a scripting language that makes it easy to implement domain specific refactorings [18]. Such scriptable refactorings could be interesting for MATLAB as well, either to implement one-off refactorings to be used for one particular code base, or to provide a refactoring tool with specific information about a program that enables otherwise unsafe transformations.

While Wrangler is an interactive tool, the tidier tool [19] performs fully automatic cleanup operations on Erlang code. The standards for behavior preservation are obvi-

ously much higher for a fully automated tool than for an interactive one, so tidier only performs small-scale refactorings, but a similar tool could certainly also be useful for MATLAB.

Refactoring legacy Fortran code has also been the subject of some research. Overbey et. al. [20,21] point out the benefits of refactoring for languages that have evolved over time. Although the specific refactorings are quite different, the motivation and the applicability of our approaches is very similar. Like MATLAB, Fortran is often used for computationally expensive tasks, hence there has been some interest in refactorings for improving program performance [22,23].

In a similar vein, Menon and Pingali have investigated source-level transformations for improving MATLAB performance [24]. The transformations they propose go beyond the typical loop transformations performed by compilers, and capture MATLAB-specific optimizations such as converting entire loops to library calls, 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 for helping to define the direction of this work.

References

1. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
2. Griswold, W.G.: Program Restructuring as an Aid to Software Maintenance. Ph.D. thesis, University of Washington (1991)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
4. Doherty, J., Hendren, L., Radpour, S.: Kind analysis for MATLAB. In: In Proceedings of OOPSLA 2011. (2011)
5. : McLab. `http://www.sable.mcgill.ca/mclab/`
6. Lameed, N., Hendren, L.J.: Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler. In: CC. (2011) 22–41
7. English, M., McCreanor, P.: Exploring the Differing Usages of Programming Language Features in Systems Developed in C++ and Java. In: PPIG. (2009)
8. Radpour, S.: Understanding and Refactoring the MATLAB Language. M.Sc. thesis, McGill University (2012)
9. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making Program Refactoring Safer. IEEE Software **27**(4) (2010) 52–57
10. Schäfer, M., de Moor, O.: Specifying and Implementing Refactorings. In: OOPSLA. (2010)
11. Tip, F., Fuhrer, R.M., Kieżun, A., Ernst, M.D., Balaban, I., Sutter, B.D.: Refactoring Using Type Constraints. TOPLAS **33** (May 2011) 9:1–9:47
12. Schäfer, M., Thies, A., Steimann, F., Tip, F.: A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. TSE (2012) To appear.
13. Roberts, D., Brant, J., Johnson, R.E.: A Refactoring Tool for Smalltalk. TAPOS **3**(4) (1997) 253–263
14. Feldthaus, A., Millstein, T., Møller, A., Schäfer, M., Tip, F.: Tool-supported Refactoring for JavaScript. In: OOPSLA. (2011)
15. Li, H., Thompson, S.J., Orösz, G., Tóth, M.: Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In: Erlang Workshop. (2008) 61–72
16. Li, H., Thompson, S.J.: Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In: PEPM. (2009) 169–178
17. Li, H., Thompson, S.J.: Refactoring Support for Modularity Maintenance in Erlang. In: SCAM. (2010) 157–166
18. Li, H., Thompson, S.J.: A Domain-Specific Language for Scripting Refactorings in Erlang. In: FASE. (2012) 501–515
19. Sagonas, K., Avgerinos, T.: Automatic Refactoring of Erlang Programs. In: PPDP. (2009) 13–24
20. Overbey, J.L., Negara, S., Johnson, R.E.: Refactoring and the Evolution of Fortran. In: SECSE. (2009) 28–34
21. Overbey, J.L., Johnson, R.E.: Regrowing a Language: Refactoring Tools Allow Programming Languages to Evolve. In: OOPSLA. (2009)
22. Overbey, J., Xanthos, S., Johnson, R., Foote, B.: Refactorings for Fortran and High-performance Computing. In: SE-HPCS. (2005) 37–39
23. Boniati, B.B., Charão, A.S., Stein, B.D.O., Rissetti, G., Piveta, E.K.: Automated Refactorings for High Performance Fortran Programmes. IJHPSA **3**(2/3) (2011) 98–109
24. Menon, V., Pingali, K.: A Case for Source-level Transformations in MATLAB. In: DSL. (1999) 53–65