# Kind Analysis for MATLAB

Jesse Doherty, Laurie Hendren and Soroush Radpour
McGill University
[jdoher1,hendren,sradpo]@cs.mcgill.ca

## Abstract

MATLAB is a popular dynamic programming language used for scientific and numerical programming. As a language, it has evolved from a small scripting language intended as an interactive interface to numerical libraries, to a very popular language supporting many language features and libraries. The overloaded syntax and dynamic nature of the language, plus the somewhat organic addition of language features over the years, makes static analysis of modern MATLAB quite challenging.

A fundamental problem in MATLAB is determining the *kind* of an identifier. Does an identifier refer to a variable, a named function or a prefix? Although this is a trivial problem for most programming languages, it was not clear how to do this properly in MATLAB. Furthermore, there was no simple explanation of kind analysis suitable for MATLAB programmers, nor a publicly-available implementation suitable for compiler researchers.

This paper explains the required background of MATLAB, clarifies the kind assignment program, and proposes some general guidelines for developing good kind analyses. Based on these foundations we present our design and implementation of a variety of kind analyses, including an approach that matches the intended behaviour of modern MATLAB 7 and two potentially better alternatives.

We have implemented all the variations of the kind analysis in MCLAB, our extensible compiler framework, and we present an empirical evaluation of the various analyses on a large set of benchmark programs.

***Categories and Subject Descriptors*** D.3.4 [*Processors*]: Compilers

***General Terms*** Experimentation, Languages, Performance

***Keywords*** MATLAB, Name Resolution, Kind Analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*OOPSLA'11,* October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10. . . $10.00

## 1. Introduction

MATLAB is a popular dynamic programming language used for scientific and numerical programming with a very large and increasing user base. The most recent data from MathWorks shows that the number of users of MATLAB was 1 million in 2004, with the number of users doubling every 1.5 to 2 years.[1] Certainly it is one of the key languages used in education, research and development for scientific and engineering applications. There are currently over 1200 books based on MATLAB and its companion software, Simulink (`http://www.mathworks.com/support/books`). This large and diverse collection of books illustrates the many scientific areas which rely on computational approaches and use MATLAB.

Given the importance of MATLAB there is a real lack of publicly-available compiler toolkits for analyzing MATLAB programs, thus hindering development in the research community for new optimization, program understanding, refactoring and verification tools. All of these tools need a good framework for program analysis. However, the dynamic nature of the language, the overloaded syntax, plus the somewhat organic addition of language features over the years, makes static analysis of modern MATLAB quite challenging.

This paper tackles the foundational problem of determining whether an identifier refers to a prefix, a variable, a named function. We call this *kind analysis*. It might seem surprising that this is a problem worth investigating, but while building our MCLAB [7] system for analyzing MATLAB we found that this was a crucial concept to understand correctly, and a crucial first phase of our analysis framework.

Without knowing the correct kinds of identifiers one cannot even build a very specific intermediate representation (IR). For example, an expression of the form "`a(i)`" could correspond to four different meanings depending on whether "`a`" is a variable or a named function, and whether "`i`" is variable or named function. If both are variables, then this is a simple array access of array "`a`", indexed by variable "`i`". If both are named functions, then this is a call to function "`a`", with an argument which is the result of a call to function "`i`". The two other cases correspond to when one

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

identifier is a variable and the other is a function. Clearly, to build an IR that is suitable for further analyses one would like to explicitly represent these different cases.

Since MATLAB does not have type declarations, and since it does not syntactically distinguish between array/variable accesses and function calls, older versions of MATLAB and other interpreter-based implementations such as Octave[8] actually determine the meaning of identifiers completely at runtime. In such systems an expression such as "`a(i)`" is stored as a very unspecific fashion such as a *parameterized expression*, and the meaning of that expression is determined, at runtime, by the interpreter. Each identifier is looked up first in the current workspace, and if it is found then the identifier refers to that variable. If it is not in the current workspace, then it is looked up in the current path of function definitions. If it is found, then the identifier refers to that named function, and if it is still not found then it is a runtime error.

Having these dynamic semantics to determine the meaning of identifiers is very easy to implement in an interpreter, but it has two main disadvantages. First, it is quite difficult for programmers and IDEs to determine the meaning of identifiers. Secondly, with such completely dynamic semantics it is hard for a JIT compiler to produce very efficient code. Thus, in more modern versions of MATLAB the semantics have been changed to include a static kind assignment to identifiers. In systems such as MATLAB 7, when a function is first loaded (JIT-compiled), a static kind analysis is applied which assigns a kind to each identifier in the body of the function. The kind analysis also raises compile-time errors for situations in which an identifier is used as both a function and variable within the same function body.

It is important to note that the static kind analysis in MATLAB 7 is not a static approximation of the old dynamic semantics, but effectively defines a new semantics for MATLAB. The new semantics will reject some programs that would previously execute in the old semantics, and it changes the meaning of some programs, particularly those that make use of other dynamic features such as "`eval`" and "`assignin`". Thus, to properly understand the meaning of a MATLAB program, and to correctly implement compilers for modern versions of MATLAB such as MATLAB 7, it is important to have a clear definition and implementation of kind analysis. We were unable to find either, and so our definition and implementation of kind analysis is the topic of this paper.

We first give a motivating example in Section 2. This illustrates the difference between the dynamic and static semantics and also shows the confusion caused when an IDE does not understand the analysis, and when a programmer does not understand the definition of the kind analysis (or even knows that such an analysis is performed by the system).

As many compiler writers are not familiar with MATLAB, and since there is no accessible formal definition of the language, we provide the essence of MATLAB semantics in Section 3. This information is needed to completely understand the problem and solution in this paper, and may also be useful for those interested in working on other compiler problems for MATLAB.

Given there was no documentation for kind analysis, we developed a large set of test programs designed to expose the intended semantics of MATLAB 7 kind analysis, and we designed and implemented a flow analysis which matches those semantics. This analysis is neither flow-insensitive nor fully flow-sensitive, but is defined by a depth-first traversal over the AST.[2] This analysis is presented in Section 4.

By defining and implementing a kind analysis to match MATLAB 7, we noted several bugs which we reported to MathWorks[3], and we also found some issues which caused us to rethink kind analysis and to suggest some alternatives. In Section 5 we present our alternatives. The first alternative is a flow-sensitive analysis very similar in spirit to the MATLAB 7 analysis, and the second alternative is a flow-insensitive approach.

We have implemented both the MATLAB 7 approach and our alternatives in our MCLAB extensible compiler framework, where it now forms the basis for all subsequent static analyses. We examined the results of the analyses on a large suite of MATLAB programs collected from a variety of sources and the results of those results are presented and evaluated in Section 6. We found that our alternative flow-sensitive analysis provided a cleaner specification while at the same time matching the original MATLAB 7 approach except in cases where our approach detected more static errors.

The main contributions of this paper are:

- We identify static kind analysis as an important concept that must be clearly defined and understood by MATLAB programmers and compiler developers. We also provide a summary of MATLAB semantics so other researchers can understand both the kind analyses in this paper and other analyses they may wish to develop.
- We present an algorithm to compute kinds which matches the intended semantics of MATLAB 7.
- We point out weaknesses of the current kind semantics/analyses and suggest two alternatives, a flow-sensitive approach and a flow-insensitive approach.
- We implemented all variations of the kind analysis and we present a comparison of the results, and we discuss the pros and cons of each approach. The implementation is part of a publicly-available toolkit.[4].

[2] It took us some time to discover this, as we were expecting either a flow-sensitive or flow-insensitive approach.

[3] These were acknowledged as bugs by MathWorks and the algorithm presented in the paper fixes those bugs.

[4] `http://www.sable.mcgill.ca/mclab/download_mclab.html`

## 2. Motivating Example

To demonstrate the problem of not having a clear definition of kind analysis, consider the toy example for the MATLAB function "`iassigni`" given in Figure 1(a). This is the display produced with MATLAB 7.11 IDE. The IDE signals that there are potential problems with this function via the red bar on the right. When the user clicks on the red bar, the warnings and explanation given in Figure 1(b) are displayed. These warnings would make sense if all occurrences of the identifier "`i`" refer to variables. However, the IDE appears to be oblivious of kind analysis, and in fact the warnings are not correct, and when the user attempts to call the function, a completely different error occurs, as listed in Figure 1(c). What is going on?

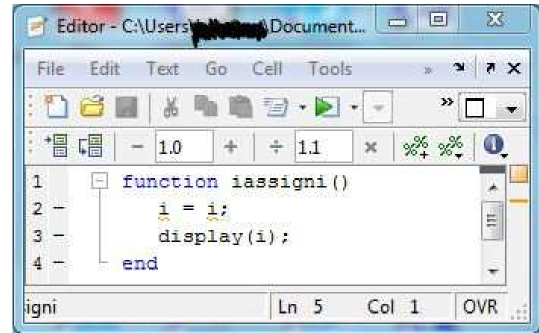
To try and understand the problem, the user might type the statements one by one into the read-eval-print loop. Figure 1(d) shows such an interaction. Each line beginning with "`>>`" is a user input. The user starts by clearing the workspace. The user then checks the binding of identifier "`i`" using the command "`which i`". This returns the fact that "`i`" refers to a built-in named function in one of the standard toolboxes (returning the mathematical value for $i$). The user then types in the previously troublesome statement "`i = i`". This actually has a well defined meaning; it first evaluates the right-hand-side, which is a call to the built-in function called "`i`", and then assigns the value to a variable called "`i`". Since variables are not declared in MATLAB, they are created upon their first definition. The user then verifies that "`i`" is now a variable using another call to "`which i`". Finally, the call to display is done, which displays the value of the variable "`i`".

The root of all this confusion is that the semantics of name lookups depends on whether or not a static kind analysis is being used. When the function "`iassigni`" is called, the JIT compiler is invoked and a kind analysis is applied. This kind analysis will determine that the first use of "`i`" corresponds to a call of the named library function "`i`", whereas the assignment to "`i`" corresponds to a variable. This kind conflict results in the error message.

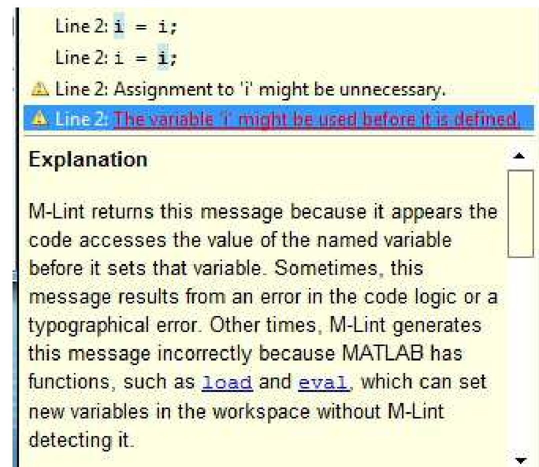
However, when the user enters the same computation line-by-line into the read-eval-print loop the kind analysis is not applied to the whole computation, which results in dynamic lookup semantics.[5]

There are two important points demonstrated by this example. First, it shows that all tools, including optimizing compilers, program-understanding tools and refactoring tools need to correctly implement kind analysis. In this case, the IDE apparently does not perform kind analysis (or does not integrate it correctly). Thus, it assumes that all occurrences of "`i`" are variables, does not correctly identify the kind conflict, and gives misleading warning messages. In the case of optimizing compilers the kinds of identifiers are needed for building correct call graphs and for correctly applying many transformations such as inlining. In the case of tools like refactoring tools, refactoring transformations must

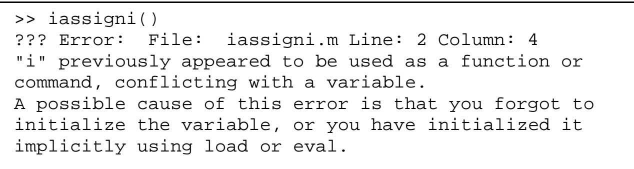
[5] Older versions of MATLAB and interpreter-based systems like Octave do not have a kind analysis, in these systems the function body would be interpreted using the dynamic lookup semantics.




(a) Function definition

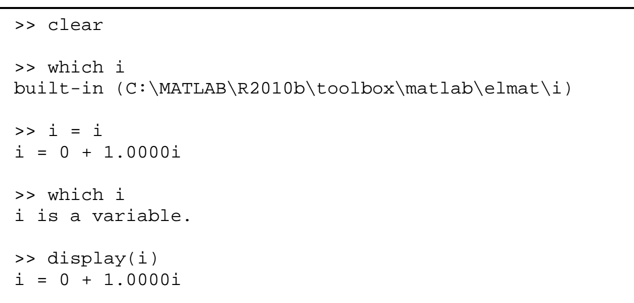



(b) IDE warnings



```
>> iassigni()
??? Error:  File:  iassigni.m Line: 2 Column: 4
"i" previously appeared to be used as a function or
command, conflicting with a variable.
A possible cause of this error is that you forgot to
initialize the variable, or you have initialized it
implicitly using load or eval.
```

(c) error when executed



```
>> clear

>> which i
built-in (C:\MATLAB\R2010b\toolbox\matlab\elmat\i)

>> i = i
i = 0 + 1.0000i

>> which i
i is a variable.

>> display(i)
i = 0 + 1.0000i
```

(d) read-eval-print execution

**Figure 1.** Motivating toy example

ensure that the refactoring does not change the kind of an identifier and does not introduce a kind conflict.

Second, this example shows that the dynamic semantics for identifier lookup that programmers may expect from having performed a computation in the read-eval-print loop are not valid within function and script definitions. Thus, programmers must be given some simple rules so they can determine the kind of identifiers in function/script definitions. This will make it easier for them to ensure they are using identifiers consistently within a function/script and will make the programs easier to understand.

Although this is just one toy example, it does demonstrate that it is important for both MATLAB programmers and the compiler/toolkit developers to have a clear understanding and clean implementation of kind analysis. In Section 4 we present a kind analysis that matches the current MATLAB 7 semantics and in the subsequent section we present some alternative kind analyses.

## 3. The Essence of MATLAB

In this section we provide a brief overview of the semantics of MATLAB, concentrating on the rules for resolving identifiers.[6] For the purposes of this paper we are concentrating on the non object-oriented part of MATLAB and we assume that all functions are either defined as MATLAB source or are well-defined built-in functions.

At a high level, a MATLAB computation executes relative to a pair <*library, env*> where *library* is a collection of named function/script definitions, and *env* is a mapping of variable names to values. We define the *library* and *env* in more detail in the next two subsections.

### 3.1 MATLAB library of function/script definitions

MATLAB functions and scripts are stored in directories with a specific format. A directory *d*, may contain:

**.m source 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'`.

**Private directories:** A directory may contain a directory named `private/`.

**Package directories:** 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:** It is possible to overload function declarations using type-specialized directories. These 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`.

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

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

The current directory and path do not normally change during the execution of the program. This is particularly true if the application has been written in a way that makes it somewhat portable. However, there do exist run-time functions which allow *dir* and *path* to change at run-time.

### 3.2 Environment and Values

The environment, *env*, corresponds to the MATLAB notion of a workspace. The environment maps variable names to values. We can think of the environment as consisting of the following four parts:

**Main workspace:** This is the initial workspace, a mapping of variable names to values, which is acted upon by commands entered into the main read-eval-print loop.

**Function call workspace stack:** A call to a function creates and pushes a new workspace, which becomes the current workspace. A return from a function pops the workspace, restoring back the new top of the stack as the current workspace. In the case of a stack frame corresponding to a nested function, there are the usual display pointers to the stack frame workspaces corresponding to the outer scopes. A call to a script does not create a

[6] Since we were unable to find any official language specification documents, these semantics have been determined from reading user-level documentation [6], and observing the behaviour of MATLAB 7.11.

new workspace, but rather uses the workspace of the last called function (i.e. the topmost stack frame).

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

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

A MATLAB value can be one of the following types: array, function handle, struct or cell array.

We can think of the basic types as being arrays and function handles. Arrays are homogeneous (i.e. all elements have the same type) and the elements must have some numeric type (double, int32, char, ...). In particular, the elements of arrays cannot be handles, structs or cell arrays. Arrays have a shape and contents, and arrays are mutable. A scalar is simply a 1x1 array. A function handle refers to a closure, where the closure consists of a reference to the function and a reference to a workspace that maps free variables to values. A function handle is created by either taking the handle of a named function (for example, "h = @sin;") or by creating a handle to an anonymous function (for example, "h = @(x)(x+1);").

Structs and cell arrays are heterogeneous and provide a way of aggregating data. Structs do not have explicit types but are constructed using calls like "a = struct('x',exp1,'y',exp2)", which would create a structure with two fields, "x" initialized to the value denoted by "exp1" and "y" initialized to the value denoted by "exp2". Each field can contain any type (array, handle, struct or cell array). Cell arrays have the same rectangular structure as arrays, but their elements are cells instead of numeric values, where each cell can contain any type. Thus cell arrays allow one to create heterogeneous and nested arrays. Cell arrays are accessed using "a(...)" which denotes the cells or "a{...}" which denotes the contents of the cells.

### 3.3 General Identifier Lookup Semantics

In the read-eval-print loop of MATLAB, the meaning of an identifier is determined at run-time, which is potentially quite expensive. The basic idea is quite simple, as outlined in "generalLookup" below. First a lookup in the current environment is performed, and if the identifier is found then the lookup returns a variable. If the identifier is not found in the environment, then a lookup in the library is performed and if an entry is found, then the lookup returns a named function/script.

```
generalLookup(id,env, lib )
 { if ( existsInEnv (id,env))
     return(lookupInEnv(id,env))
   else if ( existsInLib (id, lib ))
     return(lookupInLib(id, lib ))
   else
     error ("Undefined variable or function")
 }
```

## 4. Kind Analysis

A kind analysis for MATLAB must perform two functions: (1) it must assign a kind to each identifier occurrence in the body of functions/scripts; and (2) it must detect clashes in kind assignments within a function/script body and produce appropriate errors/warnings.

Since we were not able to find any formal description of the semantics, nor any description of the kind analysis (or even the existence of such an analysis), we developed an extensive set of test cases and observed the behaviour of MATLAB 7.11. Based on those observations we have defined the following kind analysis, which appears to replicate the semantics of MATLAB 7. By defining this analysis precisely we can both implement it in our own MATLAB system and use it as the basis for subsequent static analyses. In fact, any system attempting to match the semantics of modern MATLAB systems must implement this kind analysis (or an equivalent one), otherwise the meaning of identifiers at run-time may not be correct.

Our kind analysis assigns to each identifier one of the following abstract values:

**ID:** It is not known if the identifier refers to a named function/script or a variable. At run-time, an identifier with this kind must be looked up with the general lookup function, first looking in the environment and then in the library.

**FN:** The identifier refers to a named function/script. At run-time, an identifier with this kind is looked up directly in the library. Thus, even if a variable with this name exists at run-time, the named function/script is used.[7] If at run-time a function/script with this name does not exist in the current library, then it is an undefined function error.

**VAR:** The identifier refers to a variable. At run-time a variable with this kind is looked up in the environment only.

[7] A variable may exist due to the use of dynamic features available in MATLAB, such as "eval" and "assignin". In older versions of MATLAB, which did not have a kind analysis, such dynamically-created variables would shadow the function of the same name.

If at run-time a variable with this name is not in the environment, then it is an undefined variable error.

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

At the end of the analysis each identifier will be assigned one of the values ID, FN, VAR, or PREFIX or a compile-time error signaling a kind clash will be raised. During the analysis we use two further abstract values:

UNDEF**:** This is a special value used when analyzing functions and is used to denote the fact that the identifier has not yet occurred in any statements already analyzed. If the analysis ends without a compile-time error, then there will be no UNDEF values since every identifier will be visited at least once.

MAYVAR**:** This is a special value to indicate that the identifier might be a VAR. It is used when an identifier is explicitly mentioned by a "`load`" command. In this case the identifier may or may not refer to a variable, depending on whether or not the variable exists in the loaded file. The MAYVAR value is also used as an initial approximation when analyzing script bodies. At the end of the analysis if any identifier remains mapped to MAYVAR, its final kind is set to ID because a general lookup should be used.

These abstract values are ordered as shown in Figure 2, and based on this ordering the $\bowtie$ operation is defined, which is the join of the two values where it exists, and **error** otherwise. For example, there is no join for VAR and FN, so the result is **error**.

Although discovering the structure and details of the analysis was quite challenging, in the end the analysis itself is quite straight-forward, and consists of three steps:

**Initialize:** The initial kinds are set such that there is an initial value, $kind[id]$ for each identifier occurring in the function/script.

**Traverse function/script updating the kind abstraction:** This is a simple traversal of the AST representation of the function/script based on the program structure, updating the kind abstractions, and detecting compile-time kind errors.

The traversal order is a simple depth-first traversal of the AST. Sequences of statements are visited in the order found in the sequence; in assignment statements first the right-hand side is visited, followed by the left-hand side; in expressions the sub-expressions are visited first[8], left-to-right; in "`if`" statements the condition is visited first, followed by the "`then`" part, followed by the "`else`" part; and in loops first the condition/header is visited and then the body. In the case of functions with inner functions, these are processed recursively, first the body of the outer function, followed by a recursive analysis of all functions which are immediately nested. As the AST is traversed an abstract kind is assigned to each identifier.[9]

**Finalize kind assignments:** The final step is to finalize the kind assignments for identifier occurrences based on either the final kind abstraction (functions) or the kinds computed during the traversal (scripts).

The analysis for functions and scripts differ from each other in terms of the initial approximation and the finalization step (the first and third steps). The actual traversal and kind approximation rules (i.e. the second step) are the same in both cases. We first present the approach for both functions (Section 4.1) and scripts (Section 4.2), and then give an example to illustrate the rules, and the differences between functions and scripts (Section 4.3).

### 4.1 Kind analysis for functions

For functions, the initial kind approximation starts by assigning VAR to each input and output parameter, and UNDEF to all other identifiers (indicating no occurrence of identifier has been visited yet). Then, each statement is processed using a depth-first traversal of the AST, updating the kind approximation using the following rules and the merge ($\bowtie$) operation defined in Figure 2.

**Variable Definition:** If identifier "`x`" is being defined (ie. "`x`" is being used as an lvalue on the left-hand side of an assignment statement), or "`x`" is an argument to "`global(x)`" or "`persistent(x)`", then "`x`" must be a variable in this statement. Hence the following rule is used:

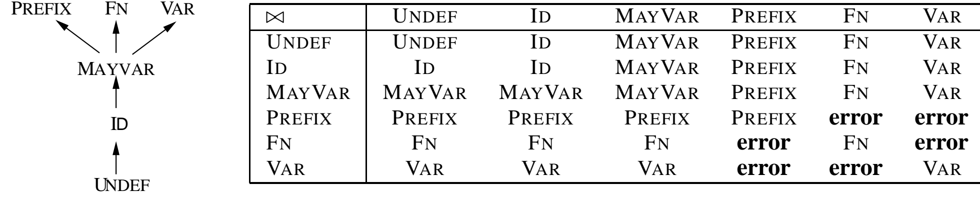
$$kind[x] \leftarrow kind[x] \bowtie \text{VAR}$$

Note that if the previous approximation for $kind[x]$ was FN, then this indicates a compile-time kind error because there must be another place where the identifier was bound to a named function and the current MATLAB semantics disallow an identifier being used both as a variable and a named function/script. Thus, in this case the analysis terminates with an error.

**Cell Array Access:** If identifier "`x`" is being used in a cell array access (either as an rvalue or lvalue), then "`x`" must be a variable. This is one case where the syntactic

[8] We should note that is important to always visit the sub-expressions first, even when processing the left-hand side of an assignment. In developing our algorithm, we uncovered a bug in the MATLAB 7.11 implementation of kind analysis. Consider the example, "`size(size(i))=1`". In this case the sub-expression should first be evaluated. Since "`size`" is a function in the library, it has kind FN. Then the outer use of "`size`" should be processed, which is a definition of a variable, and so the kind should be VAR, which is a kind clash. This clash is not reported by the current MATLAB 7.11 implementation and MathWorks has acknowledged that this is a bug.

[9] This traversal is made more explicit when we compare this approach with our proposed flow-sensitive analysis in the next section.




| ⋈ | UNDEF | ID | MAYVAR | PREFIX | FN | VAR |
|---|---|---|---|---|---|---|
| UNDEF | UNDEF | ID | MAYVAR | PREFIX | FN | VAR |
| ID | ID | ID | MAYVAR | PREFIX | FN | VAR |
| MAYVAR | MAYVAR | MAYVAR | MAYVAR | PREFIX | FN | VAR |
| PREFIX | PREFIX | PREFIX | PREFIX | PREFIX | **error** | **error** |
| FN | FN | FN | FN | **error** | FN | **error** |
| VAR | VAR | VAR | VAR | **error** | **error** | VAR |

**Figure 2.** Merge (⋈) operation for kind analysis

structure clearly disambiguates between a variable and a named function. The same analysis rule as for *Variable Definition* is used:

$$kind[x] \leftarrow kind[x] \bowtie \text{VAR}$$

**Handle Expression or Command Statement:** If identifier "x" occurs in a handle expression (i.e. an expression of the form "@x") or as the name of a command (i.e. in a statement of the form x arg), then "x" must be a named function/script.[10] Hence the following rule is used:

$$kind[x] \leftarrow kind[x] \bowtie \text{FN}$$

Note that if the previous approximation for $kind[x]$ is VAR, a compile-time error is raised (VAR ⋈ FN evaluates to **error**).

**Variable Use:** If the identifier is a use of "x" (i.e. occurs as an rvalue), then we first check to see if the $kind[x]$ is one of UNDEF or ID, in which case no previous statement has given this identifier a specific kind. In particular, this means no previously analyzed statement has made this identifier a VAR. If the identifier is in the library, we know that this occurrence of the identifier must refer to a named function. If the identifier is not in the library as a function, then we check to see if it is the name of a class/package. Hence given identifier "x", and the current library "lib", the following rule is used:

if $((kind[x] \in \{\text{ID}, \text{UNDEF}\})\&(\text{exists_function(x,lib)}))$

  $kind[x] \leftarrow \text{FN}$

elseif $((kind[x] \in \{\text{ID}, \text{UNDEF}\})\&(\text{exists_package(x,lib)}))$

  $kind[x] \leftarrow \text{PREFIX}$

else

  $kind[x] \leftarrow kind[x] \bowtie \text{ID}$

**Explicit Load:** MATLAB allows loading variables from saved .mat files. Thus, a statement of the form "load('mydata','x')" will attempt to load the value of variable "x" from the file "mydata". If the load succeeds (i.e. "x" is defined in the file) then the variable will be defined in the workspace, but otherwise it will not. Thus, we have to represent the situation where a variable may or may not exist. For this situation we use the special abstract value MAYVAR. In particular, note that if a subsequent statement finds the kind to be FN, then this is not considered to be an error, and the kind will updated to FN.

$$kind[x] \leftarrow kind[x] \bowtie \text{MAYVAR}$$

**Variable Binding an "end":** There is one remaining corner case, which is not at all obvious. In MATLAB, one can use the keyword "end" to denote the last index of an array. For example, if "a" is an array and "f" is a named function, then the value denoted by "end" in the expression "a(f(end))" is the index of the last element of "a". However, in order to bind "end" to the correct identifier, we need to know which is the closest enclosing variable name.

We have followed the intended MATLAB 7 semantics. Assume we are analyzing identifier "x" in an expression of the form "id_1(id_2 ...(id_k (x (id_m... (id_n(... end)...)", and that all of id_m to id_n have already been processed. If any of the more closely nested identifiers ("id_m" to "id_n") have the kind VAR, then the "end" has been bound.

If the "end" has not already been bound by a VAR, then there are three cases:

FN : If the kind of "x" is FN, then "x" is not binding the "end", so the kind of "x" does not change and the analysis proceeds to the next outer level. If this was the outermost level, then an error is produced indicating that there is no variable binding "end".

VAR : If the kind of "x" is VAR, then "x" is the binding var, and the kind remains as VAR and the binding process terminates.

**MAYVAR, ID or UNDEF:** In this case the analysis needs to determine if "x" is the only remaining possibility to bind "end" or not. If there are any outer identifiers ( "id_1" to "id_k") that have kind VAR, ID, MAYVAR or UNDEF, then there is no way to determine which identifier binds "end", and a kind error is generated to

[10] In MATLAB a function call of the form foo('mystring'), where the argument is a string, may also be written foo mystring, where the second alternative implicitly treats mystring as a string and not a variable. This is a very natural syntax for some commands, especially those used in the read-eval-print loop, for example one can use cd mydata instead of cd('mydata'). When used as a command, it clearly references a named function and not a variable.

indicate that the binding of "end" is ambiguous.[11] If there are no such outer identifiers, then "x" is the only remaining possibility to bind the "end", so its kind is changed to VAR, even though it is not certain that the identifier is a variable (i.e. it has not been explicitly assigned to). In the MATLAB 7 implementation the kind is changed without generating any warning, but in our suggested alternative analyses we generate a warning for this case.

After traversing the complete AST of a function, the final kind assignments are made to the identifiers. In the case of functions, all occurrences of an identifier "x" are given the same kind, based on the final values computed by the kind analysis (this is not the case for scripts). The exact assignment is as follows, assuming that the final kind assignment is stored in the mapping "kind".

```
for each id occurrence in f do
  if kind[id] in {ID, MAYVAR}
    id.kind = ID
  else    /* kind[id] in {VAR, FN, PREFIX} */
    id.kind = kind[id]
```

## 4.2 Kind analysis for scripts

Scripts in MATLAB are simply a sequence of MATLAB statements. A script is called either from the main read-eval-print loop, or from a function or another script. In the first case the script executes in the main workspace, and in the other cases it executes using the topmost stack workspace. The static kind analysis of a script must take into consideration that, unlike the case of functions, the initial set of variables is not known. Thus the initial set approximation for all identifiers is MAYVAR, representing the situation that the variable may or may not exist. Given this initial approximation the same rules are applied as with the function case.

There is one additional twist in handling the "end" expression for scripts. The rule used for functions may generate too many errors because many identifiers in scripts have the kind MAYVAR, and thus there may be many ambiguities in binding "end". For example, a script may have a statement of the form "size(a(end))" where both "a" and "size" have kind MAYVAR. Since "size" is a library function and "a" is not a library function, it is very likely (but not certain) that the programmer intended "end" to bind to "a". Thus, within scripts, the current MATLAB 7 implementation checks if one of the identifiers is in the library and the other is not. If this is the case, it binds the "end" to the identifier not in the library. If both identifiers are in the library, then it somewhat arbitrarily chooses the outermost one. If neither identifier is in the library, then it issues an ambiguity error.[12]

The script case also differs from the function case in how the final kinds are assigned to identifier occurrences. Unlike in the function case, where the final kind values are used, in the script case the values that were computed during the analysis are used. Thus, when analyzing scripts the kind analysis decorates each identifier occurrence with the current value of $kind[id]$. The final kind assignment makes a final pass through the AST adjusting the kind values as follows. Any identifier occurrence which had a kind of VAR or MAYVAR is set to be ID.[13] Note, however, that the analysis rules will still give compile-time errors for obvious mismatching of kinds.

```
for each id occurrence in s do
  if id.kind in {VAR, MAYVAR}
    id.kind = ID
  else /* id.kind must be FN, it can't be ID or UNDEF */
    id.kind = FN
```

## 4.3 Illustrative Examples

In order to illustrate the kind analysis, and also to demonstrate the differences between functions and scripts, consider the examples in Figure 3.

### 4.3.1 Kind analysis for "myfunc"

In MATLAB the declaration of a function starts with the keyword "function" followed by an optional list of output parameters, followed by the function name, an optional list input parameters, and then the function body which is terminated with the keyword "end". The function "myfunc" in Figure 3(a) has one output parameter ("r"), and two input parameters "size" and "i". Parameters have no types, and variables within the function body are declared implicitly upon first definition.

The kind analysis for "myfunc" is summarized in Figure 3(c). The table lists all the identifiers occurring in the function definition. The initialization sets the parameters ("r", "size" and "i")) to VAR, and all other identifiers to UNDEF.

The body of the function is then traversed, applying the analysis rules. For example, at line 8, identifier "s" is defined, so its kind is set to VAR. At line 9 there are two identifier uses, "s" and "magic". Following the rule for uses, "s" is already a VAR, so nothing needs to be done. At this point "magic" is still UNDEF, so a lookup is made in the library. Since "magic" is in the standard MATLAB library, its kind is set to FN. The analysis continues until the complete func-

[11] Note that we have implemented the intended MATLAB 7 semantics for the ambiguous case and not the current implementation in MATLAB 7.11, which has an acknowledged bug. The current MATLAB 7 implementation only finds an ambiguous binding if the ambiguous identifier is the immediately nesting one and thus misses some ambiguous bindings.

[12] In our alternative approaches we are slightly more rigorous. We issue a warning in the case where one identifier is in the library and the other is not, and we issue an error when both identifiers are in the library and we also issue an error when neither identifier is in the library.

[13] This seems like a strange decision to us, and we think this may just be an artifact of how scripts interact with the current JIT compiler in MATLAB 7. We suggest an alternative approach in the flow-sensitive analysis we present in the next section.

```
1  function [r] = myfunc(size, i)
2  % Returns sin or cos of magic
3  % square with dim. size(i)
4  % If size(i) is odd,
5  %    return sin of magic square
6  % else
7  %    return cos of magic square
8    s = size(i);       % s:VAR, size:VAR, i:VAR
9    a = magic(s);      % s:VAR, magic:FN, a:VAR
10   if (mod(s,2)==1)   % mod:FN, s:VAR
11     fp = @sin;       % fp:VAR, sin:FN
12     r = fp(a);       % r:VAR, fp:VAR, a:VAR
13   else
14     r = cos(a);      % r:VAR, cos:FN, a:VAR
15   end
16   display(r);        % display:FN, r:VAR
17   fp2 = @display;    % fp2:VAR, display:FN
18   display(r);        % display:FN, r:VAR
19 end
```

(a) myfunc.m

```
1  % Assumes prev defn of size and i
2  % Returns sin or cos of magic
3  % square with dim. size(i)
4  % If size(i) is odd,
5  %    return sin of magic square
6  % else
7  %    return cos of magic square
8    s = size(i);       % s:ID, size:ID, i:ID
9    a = magic(s);      % a:ID, magic:ID, s:ID
10   if (mod(s,2)==1)   % mod:ID, s:ID
11     fp = @sin;       % fp:ID, sin:FN
12     r = fp(a);       % r:ID, fp:ID, a:ID
13   else
14     r = cos(a);      % r:ID, cos:ID, a:ID
15   end
16   display(r);        % display:ID, r:ID
17   fp2 = @display;    % fp2:ID, display:FN
18   display(r);        % display:FN, r:ID
19 % end of script
```

(b) myscript.m

| Stmt # | r | size | i | s | magic | a | mod | sin | cos | fp | display | fp2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| init | V | V | V | U | U | U | U | U | U | U | U | U |
| 8 | V | V | V | V | U | U | U | U | U | U | U | U |
| 9 | V | V | V | V | F | V | U | U | U | U | U | U |
| 10 | V | V | V | V | F | V | F | U | U | U | U | U |
| 11 | V | V | V | V | F | V | F | F | U | V | U | U |
| 12 | V | V | V | V | F | V | F | F | U | V | U | U |
| 14 | V | V | V | V | F | V | F | F | F | V | U | U |
| 16 | V | V | V | V | F | V | F | F | F | V | F | U |
| 17 | V | V | V | V | F | V | F | F | F | V | F | V |
| 18 | V | V | V | V | F | V | F | F | F | V | F | V |
| final | V | V | V | V | F | V | F | F | F | V | F | V |

(c) Kind analysis for `myfunc`

| Stmt # | r | size | i | s | magic | a | mod | sin | cos | fp | display | fp2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| init | M | M | M | M | M | M | M | M | M | M | M | M |
| 8 | M | M | M | V | M | M | M | M | M | M | M | M |
| 9 | M | M | M | V | M | V | M | M | M | M | M | M |
| 10 | M | M | M | V | M | V | M | M | M | M | M | M |
| 11 | M | M | M | V | M | V | M | F | M | V | M | M |
| 12 | V | M | M | V | M | V | M | F | M | V | M | M |
| 14 | V | M | M | V | M | V | M | F | M | V | M | M |
| 16 | V | M | M | V | M | V | M | F | M | V | M | M |
| 17 | V | M | M | V | M | V | M | F | M | V | F | M |
| 18 | V | M | M | V | M | V | M | F | M | V | F | V |

(d) Kind analysis for `myscript`

**Figure 3.** Kind analysis for a function and script (Note that to save space the kinds in the table use only the first letter of the kind.)

tion body has been traversed, producing the final kind approximation shown at the bottom of the table. Identifiers "`r`", "`size`", "`i`", "`s`", "`a`", "`fp`" and "`fp2`" are VAR, and the remaining identifiers are FN.

The finalization phase traverses through the AST annotating each identifier occurrence with the kind found in the final approximation. In Figure 3(a) we have put the final kind values for identifier occurrence.

### 4.3.2 Kind analysis for "`myscript`"

MATLAB scripts are simply a sequence of MATLAB statements. The run-time meaning of identifiers in the script body can depend on the context in which the script is called. Most importantly, the free identifiers may or may not refer to existing variables.

The analysis for "`myscript`" is summarized in Figure 3(d). In the case of scripts we do not know the context in which the script will be called, and thus we have to assume that every identifier may or may not be a variable. This corresponds precisely to the MAYVAR abstraction, so all identifiers are initialized to MAYVAR.

The body of the script is then analyzed. Because all of the identifiers have an initial value of MAYVAR, the analysis can detect many fewer cases where an identifier must refer to a FN. For example, at line 9 in "`myfunc`" we could determine that since "`magic`" was not a variable, and since it was in the library, its kind became FN. However, at line 9 in "`myscript`", "`magic`" **may** be a variable, as denoted by MAYVAR. Hence, we cannot sharpen the kind estimation. In fact, it is only at lines 11 and 17, where there are explicit uses of the "`@`" operator, where we can determine that an identifier has kind FN.

The finalization phase traverses through the AST finalizing the kind assignment of each identifier occurrence using the information computed at the statement. All VAR and MAYVAR values are coarsened to ID, and all FN values remain. Note that unlike the case of functions, different occurrences of the same identifier in a script can be given different kinds. For example, at line 16 "`display`" is an ID, whereas at line 18 "`display`" is a FN. This means that if the "`myscript`" script were to be executed in an environment where "`display`" was a variable, the statement at line 16 would refer to the variable, whereas the statement at line 18 would refer to the named function.[14]

### 4.3.3 Implications of the differences in kind analysis for scripts and functions

There are two important implications of the differences between the kind analysis for functions and scripts. Firstly, the kind analysis in scripts is likely to produce much less precise (and hence less useful) kind information. This has nega-

[14] This is another "feature" of the existing MATLAB 7 semantics that we think is problematic, and which we address in our upcoming flow-sensitive analysis.

tive impact on our ability to effectively analyze and optimize code in scripts. Secondly, one cannot easily inline scripts (at the source code level), because the kind assignment (and hence the identifier lookup results) could be quite different when the code is inlined into a function. This means that any inlining in a compiler are most easily done after the kind assignment has been done, and the kinds that were computed in the script body must be retained in the inlined code.

#### 4.3.4 How kind analysis changes the semantics of MATLAB

The introduction of the kind analysis in modern versions of MATLAB has changed the semantics of the language in two ways. The first change is that some programs that previously computed a value will cause a compile-time kind error when first load/compiled by the JIT compiler. This was demonstrated by our motivating example in Figure 1. This example would run under the old dynamic name lookup (as we demonstrated using the eval-print-loop), but triggers a kind error when compiled using modern MATLAB 7.11.

The second change is due to the fact that the kind analysis assigns a specific kind to identifiers, and then at runtime performs a lookup of only that kind. Thus, programs may compute different results as compared to the dynamic lookup. For example, consider the program in Figure 4.

With a purely dynamic lookup "`sum`" denotes the function "`sum`" at line 2 and the variable "`sum`" at line 4. This is because line 3 indirectly assigns to "`sum`". Such indirect assignments may be via calls to dynamic functions like `eval` or via calls to scripts.

With a kind analysis, a static decision must be taken about whether "`sum`" denotes a VAR or FN. Since there is no direct assignment to "`sum`", and "`sum`" is in the library, "`sum`" will denote the function "`sum`" at both line 2 and line 4, even though a variable "`sum`" exists at runtime at line 4.

```
1 function [r] = KindEx(a)
2   x = a + sum(j);
3   eval('sum = ones(10); ');
4   r = sum(x);
5 end
```

**Figure 4.** Example of dynamic lookup versus lookup with kind analysis

Since kind analysis changes the semantics, both programmers and compiler-based tools must be aware of kind analysis and must implement it correctly in order to match the semantics of MATLAB 7.11.

## 5. Alternative Kind Analyses

Our original goal for this research was to clearly understand the MATLAB 7 kind analysis and to provide an algorithm and implementation for it. In the previous section we have accomplished that. However, as we specified and implemented the algorithm we were struck by ways in which it could be improved, and in this section we describe the features of the MATLAB 7 algorithm that we think are problematic, and the two different approaches we suggest.

As we discuss our approaches, it is important to keep in mind the requirements for a good kind analysis. Firstly, the kind analysis should be very simple to understand for the programmers, so that it is trivial for a programmer (or an IDE) to determine the kind of each identifier. Likewise, the kind analysis should be simple and efficient to implement. Ideally the IDE should compute the kinds as the user edits, and perhaps colour code the identifiers to distinguish variables from functions.

Related to this first point, we think the kind analysis for functions and scripts should be as similar to each other as possible. We found it very confusing that the kind analysis for functions and scripts for MATLAB 7 produce quite different final results.

Secondly, we think that the kind analysis should be explicitly flow-sensitive or explicitly flow-insensitive. The MATLAB 7 kind analysis gives the illusion of being flow-sensitive, since it is computed by a depth-first traversal of the AST. However, it is not completely flow-sensitive as it does not handle control-flow merge points. For example, consider the program snippets in Figure 5. We can see from this example that the MATLAB 7 kind analysis is *traversal-sensitive*. In the case of conditionals, it first analyzes the "`then`" branch and then analyzes the "`else`" branch. For example, in Figure 5(a), on the "`then`" branch "`sum`" gets the kind FN, and then there is a kind mismatch when the "`else`" branch is evaluated, because now "`sum`" is being assigned-to and is therefore a VAR. However, the semantically equivalent snippet in Figure 5(b) gives a different answer. In this case the "`then`" branch determines that "`sum`" is a VAR, and then the else branch also determines it is a VAR. We think that whatever analysis you use should give the same result for both the (a) and (b) variations. The analysis should either give an error for both variations (as with our flow-sensitive analysis), or it should assign VAR for both cases (as with our flow-insensitive analysis).

```
1 if (exp)
2   ... = sum(10);  % sum:FN
3 else
4   sum(10) = ...;  % *error*
5 endif
```
(a)

```
1 if (!exp)
2   sum(10) = ...;  % sum:VAR
3 else
4   ... = sum(10);  % sum:VAR
5 endif
```
(b)

**Figure 5.** Anomaly due to traversal-sensitive analysis

Thirdly, we don't think that the kind analysis should silently make assumptions about the binding of "`end`" for the cases where there is some uncertainty, but where there is a common case where an assumption is usually correct. We agree that it is practical to handle to common cases without resorting to errors, but we believe that warnings should be issued in cases where such assumptions are being made. For example, in the case for "`size(a(end))`", where "`size`" is

FN and “a” is either MAYVAR or ID, it makes sense to bind “end” to “a” and to therefore assume “a” is VAR, since this is the common case. However, in such situations, the user should be warned that this assumption is being made.

Finally, we are mindful that there are a lot of existing MATLAB programs and MATLAB programmers. In our alternate approaches we wish to keep the same spirit of the MATLAB 7 kind analysis. We aim provide alternative analyses which are cleaner and easier to understand than the MATLAB 7 approach, while at the same time produce the same results for almost all real MATLAB programs.

In Section 5.1 we outline a flow-sensitive approach, in Section 5.2 we outline flow-insensitive approach, and in Section 5.3 we provide a summary of how the MATLAB 7 approach and our two approaches compare.

### 5.1 Flow-Sensitive Kind Analysis

We have implemented both the MATLAB 7 and our alternative analyses using our structure-based flow analysis framework in McLAB. These analyses are structured as traversals over the AST. Listing 1 gives the top-level structure of our implementation of the MATLAB 7 analysis.

The procedure AnalyzeFile is applied to a file, which in turn applies the appropriate analyses for scripts or functions. Note that since the MATLAB 7 kind analysis for scripts has special requirements (for example, the analysis needs to store results for each identifier occurrence), there is a global variable to indicate whether a script of function is being analyzed.

Both AnaylzeScript and AnalyzeFunction have three phases, first the kind abstractions are initialized, then the body of the script/function is analyzed, and lastly the final mapping of the identifiers is made. AnalyzeFunction also handles the case of nested functions.

Listing 2 gives the top-level rules for our flow-sensitive semantics. The overall structure of the analysis is very similar to the MATLAB 7 version. However, there are two notable differences in these top-level rules.

First, we have decided that there is no need for the analysis of scripts to keep their kind results at each program point and thus there is no need for a global variable to differentiate between analyzing scripts and functions.

Second, we see no reason why the final mapping of kinds for scripts needs to be different than for functions, as it is in the MATLAB 7 semantics. Thus, in our flow-sensitive version we use the same final mapping of kinds for functions and scripts (i.e. we use the final result after analyzing the body, except that MAYVAR must be mapped to ID because the general lookup should be used for these identifiers).

We can see the impact of these changes in the analysis of scripts by revisiting the “myscript” example. Figure 6(a) shows the kind analysis. The MATLAB 7 semantics uses the kind analysis results associated with each statement, and maps that information to either ID of FN. If the kind was FN at that program point, then the result is FN. However, if

```
1  AnalyzeFile( fileAst )
2    switch on type of fileAst
3      case Script
4        global inScript = true
5        AnalyzeScript( fileAst )
6
7      case Function
8        global inScript = false
9        for each f in fileAst . functionList
10         AnalyzeFunction( f, {} )
11
12 function AnalyzeScript( script )
13   # initialize all names to MAYVAR
14   names = all name uses in script
15   initial = { ⟨n:MAYVAR⟩ | n ∈ names }
16   # analyze body
17   out = Analyze( script .body, initial )
18   # map each name corresponding to VAR , MAYVAR to ID
19   for each node in script
20     for each name in node.kind
21       if out[name]∈{VAR, MAYVAR}
22         set out[name] = ID
23   return out
24
25 function AnalyzeFunction( fun, outer )
26   # initialize all params to VAR
27   in = copy(outer)
28   remove all ⟨n:FUN⟩ from in
29   p1 = { ⟨n:VAR⟩ | n ∈ node.inArgs }
30   p2 = { ⟨n:VAR⟩ | n ∈ node.outArgs }
31   initial = (p1 ⋈ p2) ⋈ in
32   # analyze the function body
33   out = Analyze( node.body, initial )
34   # analyze nested functions
35   for each n in node.nested
36     outN = AnalyzeFunction( nested , out )
37     for each ⟨name:kind⟩ in outN
38       if outN[name]==VAR and out[N] in {ID, MAYVAR}
39         set out[name] = VAR
40   # map MAYVAR to ID in output
41   for each name in out
42     if out[name] == MAYVAR
43       set out[name] = ID
44   return out
```

**Listing 1.** Top-level Analysis Rules - MATLAB 7

the kind was anything else at that program point (including VAR), then it is mapped to ID. As you can see in Figure 6(b), this results is almost all identifiers being mapped to ID.

In contrast, our proposed analysis takes the kind results as collected by analyzing the whole script body, and maps it to create one final kind mapping that is used for all program points. These values are shown on the last row of the table in Figure 6(a). There appears to be no good reason to throw away the information that an identifier is a VAR, so we retain that information, and we only map MAYVAR to ID (the same rule as used for functions). It is clear that the kind information, as shown in Figure 6(c), is much more useful. Many more identifier uses can be determined to be VAR. Furthermore, the kind assignment is consistent across the whole script body. In particular, “display” has a kind of FN at both lines 16 and 18, whereas it had different kinds using the the MATLAB 7 approach.

| Stmt # | r | size | i | s | magic | a | mod | sin | cos | fp | display | fp2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| init | M | M | M | M | M | M | M | M | M | M | M | M |
| 8 | M | M | M | V | M | M | M | M | M | M | M | M |
| 9 | M | M | M | V | M | V | M | M | M | M | M | M |
| 10 | M | M | M | V | M | V | M | M | M | M | M | M |
| 11 | M | M | M | V | M | V | M | F | M | V | M | M |
| 12 | V | M | M | V | M | V | M | F | M | V | M | M |
| 14 | V | M | M | V | M | V | M | F | M | V | M | M |
| 16 | V | M | M | V | M | V | M | F | M | V | M | M |
| 17 | V | M | M | V | M | V | M | F | M | V | F | M |
| 18 | V | M | M | V | M | V | M | F | M | V | F | V |
| final | V | I | I | V | I | V | I | F | I | V | F | V |

(a) kind analysis

```
1  % Assumes prev defn of size and i
2  % Returns sin or cos of magic
3  % square with dim. size(i)
4  % If size(i) is odd,
5  %   return sin of magic square
6  % else
7  %   return cos of magic square
8    s = size(i);      % s:ID, size:ID, i:ID
9    a = magic(s);     % a:ID, magic:ID, s:ID
10   if (mod(s,2)==1)  % mod:ID, s:ID
11     fp = @sin;      % fp:ID, sin:FN
12     r = fp(a);      % r:ID, fp:ID, a:ID
13   else
14     r = cos(a);     % r:ID, cos:ID, a:ID
15   end
16   display(r);       % display:ID, r:ID
17   fp2 = @display;   % fp2:ID, display:FN
18   display(r);       % display:FN, r:ID
19 % end of script
```

(b) final mapping: MATLAB 7

```
1  % Assumes prev defn of size and i
2  % Returns sin or cos of magic
3  % square with dim. size(i)
4  % If size(i) is odd,
5  %   return sin of magic square
6  % else
7  %   return cos of magic square
8    s = size(i);      % s:VAR, size:ID, i:ID
9    a = magic(s);     % a:VAR, magic:ID, s:VAR
10   if (mod(s,2)==1)  % mod:ID, s:VAR
11     fp = @sin;      % fp:VAR, sin:FN
12     r = fp(a);      % r:VAR, fp:VAR, a:VAR
13   else
14     r = cos(a);     % r:VAR, cos:ID, a:VAR
15   end
16   display(r);       % display:FN, r:VAR
17   fp2 = @display;   % fp2:VAR, display:FN
18   display(r);       % display:FN, r:VAR
19 % end of script
```

(c) final mapping: flow-sensitive

**Figure 6.** Kind analysis for script, MATLAB 7 approach vs. flow-sensitive

```
1  AnalyzeFile( fileAst )
2    switch on type of fileAst
3      case Script
4        AnalyzeScript( fileAst )
5
6      case Functions
7        for each f in fileAst.functionList
8          AnalyzeFunction( f, {} )
9
10 function AnalyzeScript( script )
11   # initialize all names to MAYVAR
12   names = all name uses in script
13   initial = { ⟨n:MAYVAR⟩ | n ∈ names }
14   # analyze body
15   out = Analyze( script.body, initial )
16   # map MAYVAR to ID in output
17   for each name in out
18     if out[name] == MAYVAR
19       set out[name] = ID
20   return out
21
22 function AnalyzeFunction( fun, outer )
23   # ... same as before ...
```

**Listing 2.** Top-level Analysis Rules - Flow-sensitive

Listing 3 gives the core analysis rules for the MATLAB 7 approach, and Listing 4 gives the rules for our flow-sensitive approach. [15] We have already explained the basic MATLAB 7 rules in some detail in Section 4. The interesting differences here are the way in the control-flow constructs are handled.

For example, consider the IfStmt case. The MATLAB 7 analysis visits the condition, the then part and the else part, propagating kind information as it goes. In contrast, the flow-sensitive approach takes control-flow into account. It analyzes the condition and uses that output as the input to both the then and else parts, and then merges the outputs. Similarly, the flow-sensitive analysis of the loops implements a proper fixed-point computation.[16]

We have made one other small improvement in the flow-sensitive analysis. We feel that when a programmer explicitly loads a variable using the "`load`" function with the name of a variable as a string constant (i.e. "`load('mydata','x')`"), we should treat this the same as a direct assignment to the variable, and assign it a kind of VAR. Even though the "`load`" may not create the variable, the programmer's obvious intent is that the identifier should be a variable. This is also consistent with the fact that a variable only needs to be defined on one control-flow path to be assigned the kind VAR.

### 5.2 Flow-Insensitive Kind Analysis

The flow-sensitive kind analysis provides a simpler definition of the kind semantics than the MATLAB 7 version. However, to understand the analysis a programmer must reason about control-flow and merge points in their programs. One question that arises is: is the added complexity of flow-sensitivity necessary? An alternative way of looking at the kind analysis is in a flow-insensitive way. Simply look at every occurrence of an identifier and determine its kind based on how it was used, or give an error if it was used in an inconsistent way. This definition is simple to understand, it should be simple to implement and it should be efficient to execute.

To formalize this definition we need to define how different uses of an identifier are used to determine its kind. This definition is exactly the same as how different uses determine kind in the flow-sensitivity analysis. If an identifier is assigned to, it should be a VAR; if a handle is taken of it, it

[15] Note that statements assigning to flow sets do not denote copies of the flow set, one flow set is propagated, and copies are created only by using an explicitly call to the copy function.

[16] For this particular analysis the fixed-point computation only needs one iteration, but we have expressed it using our standard loop analysis framework.

```
1  Analyze( node, in )
2    switch on type of node
3      case WhileStmt
4        out1 = Analyze( node.cond, in )
5        out2 = Analyze( node.body, out1 )
6        out = out2
7
8      case IfStmt
9        out1 = Analyze( node.cond, in )
10       out2 = Analyze( node.then_body, out1 )
11       out3 = Analyze( node.else_body, out2 )
12       out =  out3
13
14     case List
15       out = in
16       for n in node. list
17         out = Analyze( n, out )
18
19     case AssignmentStmt
20       out = Analyze( node.RHS, in )
21       for each lvalue in node.LHS
22         out = Analyze( lvalue .subExpressions, out )
23         out = out ⋈ {⟨lvalue:VAR⟩}
24
25     case ParameterizedExpr
26       if( !containsEnds( node ) )
27         out1 = Analyze( node.arguments, in )
28         out2 = Analyze( node. target , out1 )
29         out = out2
30       else
31         (out, hasEnd) = AnalyzeExprWithEnds( node, in, null )
32
33     case CellIndexExpr
34       if( !containsEnds( node ) )
35         out1 = in ⋈ {⟨node.target:VAR⟩}
36         out = Analyze( node.arguments, out1 )
37       else
38         (out,boundToID) = AnalyzeExprWithEnds( arg, in, node )
39
40     case NameExpr
41       if in[node.name] ∈ {UNDEF, ID}
42          if  exists_function (node.name, library )
43             out = in ⋈ {⟨node.name:FN⟩}
44          else if exists_package (node.name, library )
45             out = in ⋈ {⟨node.name:PREFIX⟩}
46       else
47         out = in ⋈ {⟨node.name:ID⟩}
48
49     case HandleExpr
50       out = in ⋈ {⟨node.name:FN⟩}
51     case CommandStmt
52       out = in ⋈ {⟨node.cmdName:FN⟩}
53
54     case GlobalStmt or PersistentStmt
55       out = in
56       for each name in node.names
57         out = out ⋈ {⟨name:VAR⟩}
58
59     case Load
60       for i = 2: size (load . arg)
61         if load . arg [ i ] is a string
62           if load . arg [ i ] does not start with '−'
63             out = in ⋈ {⟨load.arg[i]:MAYVAR⟩}
64
65   if( inScript ) node.kind = copy(out) else node.kind = out
66   return out
```

**Listing 3.** Main Analysis Rules: MATLAB 7

```
1  Analyze( node, in )
2    switch on type of node
3      case WhileStmt
4        out = Analyze( node.cond, in )
5        previousIn = in
6        do
7          previousOut = out
8          outBody = Analyze( node.body, previousOut )
9          newIn = out ⋈ previousIn
10         out = Analyze( node.cond, newIn )
11       while( out != previousOut )
12
13     case IfStmt
14       condOut = Analyze( node.cond, in )
15       thenIn = copy(condOut)
16       elseIn = copy(condOut)
17       thenOut = Analyze( node.then_body, thenIn )
18       elseOut = Analyze( node.else_body, elseIn )
19       out = thenOut ⋈ elseOut
20
21     # ... other cases are the same ...
22
23     case Load
24       for i = 2: size (load . arg)
25         if load . arg [ i ] is a string
26           if load . arg [ i ] does not start with '−'
27             out = in ⋈ {⟨load.arg[i]:VAR⟩}
28
29   node.kind = out
30   return out
```

**Listing 4.** Main Analysis Rules - modified flow-sensitive

should be a FN; if it's explicitly loaded, it should be a VAR; if it is in the library, it should be a FN; if it's cell indexed or has an "`end`" bound to it, it should be a VAR. The difference is in the ordering of when these cases are applied.

Even though this analysis will be flow-insensitive, some care is needed when ordering how identifiers get assigned a kind. To demonstrate this, we present the following simple program:

```
1  i = 3;
2  x = i;
```

A programmer would expect this program to execute without issue. It's reasonable to expect this even though "`i`" is a function defined in the MATLAB library. However, if the flow-insensitive analysis were to look at the occurrence of "`i`" on line 2 before the one on line 1, this would be an error. It's an error because first we determined that "`i`" has kind FN, then we saw an occurrence of it being assigned to. This causes us to try to make `i`'s kind VAR, but as was seen in Section 4, this would cause a kind conflict error. Alternatively, if the analysis saw line 1 before line 2, it would cause no error. The behavior of the kind analysis should be deterministic, so an ordering needs to be defined. This ordering should not cause reasonable programs, such as our example, to be rejected. Finally, the ordering should be defined in a simple and clear way.

Rather than define an ordering to how nodes are visited, the flow-insensitive analysis defines five phases. Each phase conceptually represents a traversal of all nodes in the AST,

and each phase performs one case for determining the kind of identifiers. These phases performed in the following order.

1. Assign VAR to all identifiers on the left-hand side of assignments; identifiers mentioned in a "`load`", "`global`", "`persistent`"; or identifiers indexed with cell indexing.
2. Assign FN to all identifiers that have a handle taken of them or used for command syntax.
3. Assign FN to all identifiers that are not VAR and are in the library as a function or PREFIX if it is in the library as a package.
4. Check that all "`end`" expressions can be bound without ambiguity and issue an error for ambiguous cases.
5. For each unambiguous "`end`" expression select the correct identifier to bind to the "`end`". If the selected identifier has kind ID, change it to kind VAR and issue a warning. If there is no selected identifier, issue an error.

The first three phases can be thought of in the following way. First all obvious variables are found, then all obvious functions, then the less obvious functions. Each of these phases is run completely, and if any phase tries to assign a kind to an identifiers that already has a conflicting kind, then a kind conflict error occurs.

The fourth and fifth phases handle the case for binding "`end`" expressions. The fourth phase processes all expressions containing "`end`" to ensure that there is no ambiguity in choosing the binding identifier. The fifth phase then revisits the unambiguous expressions containing "`end`", determines which identifier binds the "`end`". If the binding identifier has kind ID, it assigns it the kind VAR and issues a warning. If there is no binding identifier, then it issues an error.

The fourth and fifth stages must be conceptually separate passes because we do not want traversal order to matter. If we were to change the kind of some identifier `x` from ID to VAR during the fourth pass, this could affect a subsequent ambiguity check involving `x`. This would mean that different traversal orders could give different results. To prevent this, all of the ambiguity checks are completed in the fourth pass and then all unambiguous bindings and changes of kinds from ID to VAR are made in the fifth pass.

Pseudo-code that demonstrates this ordering is given in Listing 5. This pseudo-code gives the top-level rule for analyzing functions and scripts. Listing 6 presents the main analysis rules for the flow-insensitive analysis. It's important to note that there are no rules for nodes like "`if`" and "`while`". Traversal over these nodes is captured in the top-level rules.

It's important to note that this ordering has a preference for assigning VAR. This causes some differences over the flow-sensitive analysis when it comes to error reporting. For example, the following code:

```
1   x = i;
2   i = 3;
```

```
1  function AnalyzeFunction( fun, outer )
2    in = copy(outer)
3    remove all ⟨n:FUN⟩ from in
4    out1 = { ⟨n:VAR⟩ | n ∈ node.inArgs }
5    out2 = { ⟨n:VAR⟩ | n ∈ node.outArgs }
6    out = (out1 ⋈ out2) ⋈ in
7    AnalyzeBody(fun.body)
8
9    for each n in node.nested
10     outN = AnalyzeFunction( nested, out )
11     for each ⟨name:kind⟩ in outN
12       if outN[name]==VAR and out[N] in {ID, MAYVAR}
13         set out[name] = VAR
14
15 function AnalyzeScript( script )
16   # initialize all names to MAYVAR
17   names = all name uses in script
18    initial = { ⟨n:MAYVAR⟩ | n ∈ names }
19   AnayzeBody(script.body)
20
21 function AnalyzeBody( body )
22   out1 = out
23   for each AssignmentStmt,Load,Global, Persistent ,CellIndex node in body
24     out1 = Analyze( node, out1 )
25
26   out2 = out1
27   for each HandleExpr handle in body
28     out2 = Analyze( handle, out2 )
29
30   out3 = out2
31   for each NameExpr name in body
32     out3 = Analyze( name, out3 )
33
34   out4 = out3
35   for each ParameterizedExpr exp in body
36     if hasEnd(exp)
37       out4 = AmbiguityCheck( name, out4 )
38
39   out5 = out4
40   for each ParameterizedExpr exp in body
41     if hasEnd(exp)
42       out5 = Analyze( name, out5 )
43
44   out = out5
```

**Listing 5.** Ordering of Flow-Insensitive Analysis

In the flow-sensitive analysis, this code would have been flagged as causing an error. This is because line 1 is visited before line 2 and on line 1 the occurrence of "`i`" is treated as a function call. When it reaches line 2, the flow-sensitive analysis tries to make "`i`" a VAR, which causes the error. In the flow-insensitive version, this code would not be flagged. It simply treats "`i`" as a variable. Presumably, at runtime, if this variable had no value, it would cause a variable-not-defined error. This means the flow-insensitive analysis provides fewer static guarantees. However, if an error occurred at runtime, the kind information could be used to give a more precise error since it knew that it was expecting a variable to be defined.

Even though the analysis is described as five passes through the code, our implementation optimizes this by introducing some new abstract values and implements one pass through the code that handles the first three phases,

```
1 function Analyze( node, in )
2   switch on type of node
3     case AssignmentStmt
4       for each lvalue in node.LHS
5         out = out ⋈ {⟨lvalue:VAR⟩}
6
7     case NameExpr
8       if in[node.name] ∈ {UNDEF, ID} and node.name ∈ library
9         out = in ⋈ {⟨node.name:FN⟩}
10      else
11        out = in ⋈ {⟨node.name:ID⟩}
12
13    case HandleExpr
14      out = in ⋈ {⟨node.name:FN⟩}
15
16    case Load
17      for i = 2: size (load . arg)
18        if load . arg [ i ] is a string
19          if load . arg [ i ] does not start with '−'
20            out = in ⋈ {⟨load.arg[i]:VAR⟩}
21
22    # ... ParameterizedExpr, GlobalStmt, PersistentStmt , CellIndex same ...
23
24  node.kind = out
```

**Listing 6.** Main Analysis Rules - modified flow-insensitive

and a second pass through only the expressions that contain "`end`" expression.

We believe that the flow-insensitive analysis is more suitable for use in an IDE because the analysis does not depend on the order of visiting the nodes, and it is quite easy to update the results if the user adds a new statement.

### 5.3 Summary of Analysis Differences

We have presented three algorithms for kind analysis and discussed their differences. Table 1 gives an overall summary of the key differences. The leftmost column gives a name to each situation, where **D** stands for a definition, **U** stands for a use, and **H** stands for a function handle. There are also two specializations of uses, $\mathbf{U}_l$ stands for an identifier which corresponds to a function in the library and $\mathbf{U}_n$ stands for an identifier that does not correspond to a function in the library.

For each situation we give a small snippet of code which corresponds to the situation, and the analysis results for that piece of code for each of the three analyses. For the MATLAB 7 and flow-sensitive analyses we give the analysis results that would be computed during the propogation phase of the analysis, and then the final kind assignment. For the flow-insensitive analysis we give only the final result.

The first block of five situations represents cases when there is only one statement involving identifier "`i`". All three analyses compute exactly the same result for the first four situations. However, the fifth situation, $\mathbf{E}_u$, does show a difference. This is the case where the variable "`i`" is binding and "`end`", and "`i`" has kind UNDEF or ID. The MATLAB 7 approach silently determines that "`i`" is a VAR, whereas our two approaches warn that this kind assignment is being made.

The second block of five situations represents cases where there are two statements involving identifier "`i`" where there is sequential control flow between the two statements. Again the first four situations give identical results for all three analyses, but the fifth one, $\mathbf{U}_l\mathbf{D}$, gives a different result for the flow-insensitive analysis (as discussed in the previous subsection).

The final block of six situations corresponds to cases where there are two statements involving identifier "`i`", but there is no control flow between them. These exhibit more differences between the analyses. The two cases $\mathbf{D}\|\mathbf{U}_l$ and $\mathbf{U}_l\|\mathbf{D}$ are particularly interesting. In both of these cases "`i`" should be a FN on one branch and a VAR on the other branch, and thus there should be a kind conflict. The strange traversal strategy of the MATLAB 7 approach finds the error in second case, but not in the first, whereas the flow-sensitive approach correctly finds errors in both cases. The flow-insensitive approach ignores any control flow and does not signal an error in either case, but gives a kind of VAR because there exists an assignment to "`i`".

## 6. Empirical Study

In order to experiment with our analyses we gathered a large number of MATLAB projects.[17] The benchmarks come from a wide variety of application areas including Computational Physics, Statistics, Computational Biology, Geometry, Linear Algebra, Signal Processing and Image Processing. We analyzed 3057 projects composed of 11692 functions and 2307 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 2 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.[18]

| Benchmark Category | # Benchmarks |
|---|---|
| Single (1 file) | 2067 |
| Small (2-9 files) | 859 |
| Medium (10-49 files) | 120 |
| Large (50-99 files) | 9 |
| Very Large ($\geq$ 100 files) | 2 |
| Total | 3057 |

**Table 2.** Distribution of size of the benchmarks

[17] 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/`.

[18] We continue to add to the benchmark set, further contributions are most welcome.

| Name | Code | MATLAB 7 | Flow-Sensitive | Flow-Insensitive |
|---|---|---|---|---|
| **D** | `i =    // S1` | Prop, Final<br>S1: VAR, VAR | Prop, Final<br>S1: VAR, VAR | Final<br>S*: VAR |
| $\mathbf{U}_n$ | `  = i // S1` | Prop, Final<br>S1: ID, ID | Prop, Final<br>S1: ID, ID | Final<br>S*: ID |
| $\mathbf{U}_l$ | `  = i // S1` | Prop, Final<br>S1: FN, FN | Prop, Final<br>S1: FN, FN | Final<br>S*: FN |
| **H** | `  = @i // S1` | Prop, Final<br>S1: FN, FN | Prop, Final<br>S1: FN, FN | Final<br>S*: FN |
| $\mathbf{E}_u$ | `  = i(end) // S1` | Prop, Final<br>S1: VAR, VAR | Prop, Final<br>S1: ID, VAR,**warn** | Final<br>S*: VAR,**warn** |
| **DU** | `i =    // S1`<br>`...`<br>`  = i // S2` | Prop, Final<br>S1: VAR, VAR<br>S2: VAR, VAR | Prop, Final<br>S1: VAR, VAR<br>S2: VAR, VAR | Final<br>S*: VAR |
| **DH** | `i =    // S1`<br>`...`<br>`  = @i // S2` | Prop, Final<br>S1: VAR, **error**<br>S2: **error**, **error** | Prop, Final<br>S1: VAR, **error**<br>S2: **error**, **error** | Final<br>S*: **error** |
| **HD** | `  = @i // S1`<br>`...`<br>`i =    // S2` | Prop, Final<br>S1: FN, **error**<br>S2: **error**, **error** | Prop, Final<br>S1: FN, **error**<br>S2: **error**, **error** | Final<br>S*: **error** |
| $\mathbf{U}_n\mathbf{D}$ | `  = i // S1`<br>`...`<br>`i =    // S2` | Prop, Final<br>S1: ID, VAR<br>S2: VAR, VAR | Prop, Final<br>S1: ID, VAR<br>S2: VAR, VAR | Final<br>S*: VAR |
| $\mathbf{U}_l\mathbf{D}$ | `  = i // S1`<br>`...`<br>`i =    // S2` | Prop, Final<br>S1: FN, **error**<br>S2: **error**, **error** | Prop, Final<br>S1: FN, **error**<br>S2: **error**, **error** | Final<br>S*: VAR |
| $\mathbf{D}\Vert\mathbf{U}_n$ | `if (e)`<br>`  i =    // S1`<br>`else`<br>`    = i // S2`<br>`end // S3` | Prop, Final<br>S1: VAR, VAR<br>S2: VAR, VAR | Prop, Final<br>S1: VAR, VAR<br>S2: ID, VAR<br>S3: VAR, VAR | Final<br>S*: VAR |
| $\mathbf{U}_n\Vert\mathbf{D}$ | `if (e)`<br>`    = i // S1`<br>`else`<br>`  i =    // S2`<br>`end // S3` | Prop, Final<br>S1: ID, VAR<br>S2: VAR, VAR | Prop, Final<br>S1: ID, VAR<br>S2: VAR, VAR<br>S3: VAR, VAR | Final<br>S*: VAR |
| $\mathbf{D}\Vert\mathbf{U}_l$ | `if (e)`<br>`  i =    // S1`<br>`else`<br>`    = i // S2`<br>`end // S3` | Prop, Final<br>S1: VAR, VAR<br>S2: VAR, VAR | Prop, Final<br>S1: VAR, **error**<br>S2: FN, **error**<br>S3: **error**, **error** | Final<br>S*: VAR |
| $\mathbf{U}_l\Vert\mathbf{D}$ | `if (e)`<br>`    = i // S1`<br>`else`<br>`  i =    // S2`<br>`end // S3` | Prop, Final<br>S1: FN, **error**<br>S2: **error**, **error** | Prop, Final<br>S1: FN, **error**<br>S2: VAR, **error**<br>S3: **error**, **error** | Final<br>S*: VAR |
| $\mathbf{D}\Vert\mathbf{H}$ | `if (e)`<br>`  i =     // S1`<br>`else`<br>`    = @i // S2`<br>`end // S3` | Prop, Final<br>S1: VAR, **error**<br>S2: **error**, **error** | Prop, Final<br>S1: VAR, **error**<br>S2: FN, **error**<br>S3: **error**, **error** | Final<br>S*: **error** |
| $\mathbf{H}\Vert\mathbf{D}$ | `if (e)`<br>`    = @i // S1`<br>`else`<br>`  i =    // S2`<br>`end // S3` | Prop, Final<br>S1: FN, **error**<br>S2: **error**, **error** | Prop, Final<br>S1: FN, **error**<br>S2: VAR, **error**<br>S3: **error**, **error** | Final<br>S*: **error** |

**Table 1.** Comparative Summary of the three *kind analyses*

We ran the three kind analyses on all files in the benchmarks, and we categorized all identifiers based on their kind assignment. The cumulative results for both functions in Table 3 and scripts in Table 4.

### 6.1 Kind results for functions

Table 3 gives the results for the functions. For functions we counted the number of identifiers (not identifier occurrences). For example, for "`myfunc`" in Figure 3(a) there are 12 identifiers and we would have counted 7 VAR and 5 FN, meaning that we found 7 variables and 5 named functions. Overall, we found that about 58% of the identifiers were variables and about 40% were named functions. Only 1.2% of the identifiers remained uncategorized (ID) after the kind analysis. The three different kind analyses had only small differences in their outputs, which is what we intended. We examined all benchmarks where the results differed in order to determine the cause of the difference.

| Kind | # Id. (Matlab) | # Id. (FS) | # Id. (FI) |
|---|---|---|---|
| VAR | 107327 | 107340 | 107345 |
| FN | 75486 | 75486 | 75486 |
| ID | 2357 | 2333 | 2333 |
| PREFIX | 12 | 12 | 12 |
| **error** | 1 | 3 | 0 |
| **warn** | 0 | 9 | 7 |
| Total | 185183 | 185183 | 185183 |

**Table 3.** Cumulative Results for Kind Analysis of 11698 functions

Let us first summarize the differences in the number of errors found. The flow-insensitive (FI) approach did not find any kind errors in any of the benchmarks. The flow-insensitive approach can only find kind clashes for very explicit situations such as **DH**, **HD**, $\mathbf{D}\|\mathbf{H}$, and $\mathbf{H}\|\mathbf{D}$. The fact that no such kind error occurs in any of the benchmarks may mean that programmers find those rules easy to follow and are unlikely to make this sort of error. The MATLAB 7 approach found only one more kind error than the flow-insensitive approach, and this corresponded to a $\mathbf{U}_l\|\mathbf{D}$ case. As we expected, the flow-sensitive (FS) approach caught more programming errors. The two extra errors that were only found using the flow-sensitive approach were of type $\mathbf{D}\|\mathbf{U}_l$.

Both our flow-sensitive and flow-insensitive approaches issue warnings when an "`end`" expression causes an identifier to be given the type VAR (the $\mathbf{E}_u$ situation). The flow-insensitive version finds fewer such warnings. The two warnings which are caught by the flow-sensitive approach, but not by the flow-insensitive approach, are in situations where there is an assignment to the identifier later in the program. In the flow-insensitive case all assignments are analyzed first, so when the end expression is analyzed the $kind[x]$ is already VAR, and no warning is issued.

The number of identifiers found to be FN was exactly the same for all three approaches. All of the remaining differences come from assigning VAR instead of MAYVAR (which is mapped to ID when analysis ends) to load arguments.

Perhaps the most interesting aspect of the results is the small differences between the results using the original MATLAB 7 semantics and our proposed approaches. Our goal was to design cleaner approaches, to make the results of kind analysis easier to understand, but to avoid breaking the working codes as much as possible. Based on our results we feel that we achieved that, and we can recommend the improved approaches to be used in future versions of the MATLAB language.

### 6.2 Kind results for scripts

Table 4 gives the results for the scripts. Recall that in MATLAB 7 approach, a variable inside a script can only have final kinds of ID, FN or **error**, and that one identifier can actually have different kinds at different program points. However, in our two alternative approaches we update the kind of all occurrences of an identifier based on the final kind analysis results, and we retain the VAR kind.

Because the MATLAB 7 results are program-point specific, for scripts we counted the kinds for each identifier occurrence. For example, for the script in Figure 6, we would count the kind for the 22 identifier occurrences as summarized in the comments. We have also reported, for the MATLAB 7 case, the kinds both before and after the final mapping (post-process) of kinds.

| Kind | #Id. Matlab raw | #Id. Matlab post-process | #Id. FS | #Id. FI |
|---|---|---|---|---|
| VAR | 153563 | 0 | 154065 | 154075 |
| FN | 1 | 1 | 3 | 3 |
| ID | 69027 | 222590 | 68413 | 68413 |
| **error** | 0 | 0 | 0 | 0 |
| **warn** | 0 | 0 | 110 | 100 |
| Total | 222591 | 222591 | 222591 | 222591 |

**Table 4.** Cumulative Results for Kind Analysis of 2305 scripts

The results for scripts are very different than for functions, with almost all of the identifiers being given the kind ID in the MATLAB 7 approach (even though before the post-process step many identifiers had been determined to be VAR). Since almost all identifiers will have kind ID, which have an expensive general lookup, the runtime overheads for resolving names in scripts will be higher than for functions. This lack of accurate static information about the identifiers in scripts also implies that any subsequent static analysis of the script will have limited precision. In both our flow-sensitive and flow-insensitive approaches, we find a significant number of VAR identifiers, which improves the situation. Furthermore, keeping the VAR kind makes the behavior

for scripts more similar to functions and as a result makes the language easier to understand. This will also help in subsequent static analysis and compilation of the code.

It is also interesting to note that the flow-sensitive and flow-insensitive analysis compute the same kinds for almost all identifier occurrences in the scripts. Again like in functions, the flow-insensitive approach found fewer warnings. Both of them manage to find slightly more VAR instances than the MATLAB 7 approach (before post-processing) because of our treatment of explicit loads to named variables (remember that we use VAR instead of MAYVAR). The two cases of FN is from a code that references an identifier twice and then takes a function handle of the identifier. In scripts MATLAB keeps copy of the kind at each program-point so the previous occurrences of the identifier don't see the change of kind to FN.

Even with our improved analysis for scripts it is inevitable that many identifiers will not be able to be assigned a kind more precise than ID. The overall numbers show a similar trend to what we observed in the "`myscript`" example from Figure 6. In that example, 5 of the 12 identifiers are given a final kind of ID (size,i,magic, mod and cos). In all of these cases there are no explicit statements within the body of the script which can determine if these have kind VAR or FN. Indeed, whether they are VAR or FN depends on the context from which they are called. If the identifier is a variable in the caller's workspace, then the identifier will refer to a variable, and if it is not in the caller's workspace, the identifier will refer to a named function. Thus, there remain overheads for using scripts, both in the extra lookups required and in the possible imprecision of subsequent analyses.

## 7. Related Work

This paper tackles a very basic problem, giving a meaning to identifiers in a program. With many languages these are trivial issues that are dealt with by standard front-end parsing and symbol table modules. Even reasonably complicated situations, such as properly disambiguating package names in Java, usually have quite clear specifications and static types to work with. However, as we discovered when building our MATLAB infrastructure, the kind analysis problem for MATLAB was not obvious and we could find no documentation or prior work on this problem.

There are other open MATLAB-like systems such as Octave[8] and Scilab[3]. Octave uses a syntax mostly compatible with MATLAB, whereas Scilab defines a somewhat different syntax. Both of these systems concentrate on providing an interpreter for a MATLAB-like language, rather than providing a static analysis framework. Thus, all of the complexities of deciding the meaning of an identifier are deferred until runtime and these systems do not use a kind analysis. For example, in Octave, the example of "`i = i`" from the motivating example of Figure 1 executes without raising a compile-time error. Octave uses the completely dynamic semantics - at run-time the right-hand-side "`i`" refers to a function and the left-hand-side "`i`" refers to a variable.

There have also been research systems which had impressive static type inference analyses for subsets of MATLAB, including the FALCON[9], MAGICA[4] and MaJIC[1] systems. More recently the McFor[5] and McVM[2] systems have implemented variations on type and shape analysis in the context of ahead-of-time and JIT compilers for subsets of MATLAB. This paper is really addressing a simpler, but fundamental, problem for a modern version of MATLAB. To match the semantics of MATLAB 7, the kind analysis must be first be run to assign kinds to identifiers and to create an appropriately specialized IR on which the more complex shape analyses can be applied. Our hope is that a wide variety of static analyses, including similar sophisticated type inference algorithms, can be implemented more easily and for a larger language subset starting with our infrastructure.

## 8. Conclusions

This paper has presented the problem of defining and implementing kind analyses for MATLAB. When we started our project for developing an open and extensible compiler and analysis framework for MATLAB we did not imagine that this was a potential paper topic - we expected it to be trivial to build a good intermediate representation and that a decent front-end could resolve all the identifiers. However, the more we learned about MATLAB, the more we realized that this is actually a foundational problem and that a good solution to this problem was imperative as a starting point for all other static analyses.

Our first objective was to specify the kind analysis as it is implemented in MATLAB 7. As there is no written language standard for MATLAB, we accomplished this by developing a large set of tests that were designed to expose the subtleties of the kind analysis. Based on these tests we developed a kind analysis which appears to match the intended MATLAB 7 semantics. In this process we discovered several bugs or inconsistencies in the MATLAB 7 implementation, which we reported to MathWorks.

While developing the MATLAB 7 kind analysis algorithm, we found aspects of the approach that we thought could be improved. Thus, we also designed two new variations of the kind analysis, one flow-sensitive analysis, and another flow-insensitive analysis. In both cases we incorporated improvements to the analysis, especially in the treatment of scripts.

Our objective was to keep the general intention of the MATLAB 7 kind analysis, but to have cleaner specifications which would be easier for programmers to understand and for tool implementers to implement.

We implemented all three kind analyses in our MCLAB compiler framework, and evaluated the three analyses on a large number of MATLAB 7 programs which come from a

wide variety of sources. We were quite pleased to see that our cleaner kind analyses did not differ in many cases from the original MATLAB 7 approach.

Although as compiler researchers we tend to prefer the flow-sensitive analysis because it gives more precise results and is able to detect a few more compile-time warnings and errors, we think that the flow-insensitive approach has important practical merits. It is almost as good as the flow-sensitive analysis in finding kind warnings and errors, it is probably simpler to explain to MATLAB programmers, and it is likely more suitable for implementing in an IDE. Thus, we would recommend that the flow-insensitive definition of kind analysis be adopted for MATLAB, and that a standard be developed based on that definition.

As compiler researchers more familiar with imperative and object-oriented languages, which normally have an official language specification, we had to spend considerable time and effort understanding the implicit semantics of MATLAB, and formulating those semantics more explicitly in a way that we could understand in the context of static analysis. Thus, we hope that another key contribution of this paper is explaining the essence of MATLAB so that other compiler researchers can benefit from our experience.

The techniques presented in this paper now form the foundation of our analysis framework[7]. Based on kind analysis we are able to build a good intermediate representation suitable for further analysis development, both in the field of optimizations and for other applications such as refactoring tools.

## Acknowledgements

This work was done with support from NSERC (Canada), McGill University (Canada), and The Leverhulme Trust (UK). Many thanks to the many McGill students who have worked on building the MCLAB framework, and the OOPSLA reviewers who had many useful suggestions.

## References

[1] G. Almási and D. Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 294–303, New York, NY, USA, 2002. ACM.

[2] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB through Just-In-Time Specialization. In *International Conference on Compiler Construction*, pages 46–65, March 2010.

[3] INRIA. Scilab, 2009. `http://www.scilab.org/platform/`.

[4] P. G. Joisha and P. Banerjee. Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In *APL '01: Proceedings of the 2001 conference on APL*, pages 7–21, New York, NY, USA, 2001. ACM.

[5] J. Li. McFOR: A MATLAB to FORTRAN 95 compiler. Master's thesis, McGill University, Montreal, Canada, 2009.

[6] MathWorks. MATLAB Documentation, 2010. `http://www.mathworks.com/`.

[7] McLAB. McLAB. `http://www.sable.mcgill.ca/mclab/`.

[8] Octave. GNU Octave. `http://www.gnu.org/software/octave/index.html`.

[9] L. D. Rose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.