Overview – PART I

• Why MATLAB?
• Introduction to MATLAB – challenges
• Overview of the McLAB tools
• Resolving names in MATLAB


• 38% of scientists spend at least 1/5th of their time programming.
• Codes often buggy, sometimes leading to papers being retracted. Self-taught programmers.
• Monster codes, poorly documented, poorly tested, and often used inappropriately.
• 45% say scientists spend more time programming than 5 years ago.

A lot of MATLAB programmers!

• Started as an interface to standard FORTRAN libraries for use by students.... but now
  – 1 million MATLAB programmers in 2004, number doubling every 1.5 to 2 years.
  – over 1200 MATLAB/Simulink books
  – used in many sciences and engineering disciplines
• Even more “unofficial” MATLAB programmers including those using free systems such as Octave or SciLab.
Why do Scientists choose MATLAB?

MATLAB

FORTRAN

Implications of choosing a dynamic, “scripting” language like MATLAB....

No types and “flexible” syntax

http://imgs.xkcd.com/comics/fourier.jpg

Many run-time decisions ...

Potentially large runtime overhead in both time and space

Most semantic (syntactic) checks made at runtime ... No static guarantees

No formal standards for MATLAB
Culture Gap

Scientists / Engineers
- Comfortable with informal descriptions and "how to" documentation.
- Don't really care about types and scoping mechanisms, at least when developing small prototypes.
- Appreciate libraries, convenient syntax, simple tool support, and interactive development tools.

Programming Language / Compiler Researchers
- Prefer more formal language specifications.
- Prefer well-defined types (even if dynamic) and well-defined scoping and modularization mechanisms.
- Appreciate "harder/deeper/more beautiful" programming language/compiler research problems.

Goals of the McLab Project
- Improve the understanding and documentation of the semantics of MATLAB.
- Provide front-end compiler tools suitable for MATLAB and language extensions of MATLAB.
- Provide a flow-analysis framework and a suite of analyses suitable for a wide range of compiler/soft. eng. applications.
- Provide back-ends that enable experimentation with JIT and ahead-of-time compilation.

Enable PL, Compiler and SE Researchers to work on MATLAB

Brief Introduction to MATLAB

Functions and Scripts in MATLAB

Basic Structure of a MATLAB function

```
function [ prod, sum ] = NestedSubEx( a, n )
    function [ z ] = MyTimes( x, y )
        z = x * y;
    end
    prod = 1;
    sum = 0;
    for i = 1:n
        prod = MyTimes( prod, a(i) );
        sum = MySum( sum, a(i) ) ;
    end;
end
```

```
>> [a,b] = ProdSum([10,20,30],3)
    a = 6000
    b = 60
>> ProdSum( [10,20,30],2 )
    ans = 200
>> ProdSum( 'abc',3 )
    ans = 941094
>> ProdSum( [97 98 99], 3 )
    ans = 941084
```

Basic Structure of a MATLAB script

```
% stored in file ProdSumScript.m
prod = 1;
sum = 0;
for i = 1:n
    prod = prod * a(i);
    sum = sum + a(i);
end;
```

```
>> clear
>> a = [10, 20, 30];
>> >>
>> whos
    Name      Size         Bytes  Class      Attributes
    a        1x3              24  double
    >> ProdSumScript()
>> whos
    Name      Size         Bytes  Class      Attributes
    a        1x3              24  double
    i         1x1              8  double
    n         1x1              8  double
    prod     1x1              8  double
    sum      1x1              8  double
```
Directory Structure and Path

- Each directory can contain:
  - .m files (which can contain a script or functions)
  - a private/ directory
  - a package directory of the form +pkg/
  - a type-specialized directory of the form @int32/

- At run-time:
  - current directory (implicit 1st element of path)
  - directory of last called function
  - path of directories
  - both the current directory and path can be changed at runtime (cd and setpath functions)

Function/Script Lookup Order (call in the body of a function f)

- Nested function (in scope of f)
- Sub-function (in same file as f)
- Function in /private sub-directory of directory containing f.
- 1st matching function, based on function name and type of first argument, looking in type-specialized directories, looking first in current directory and then along path.
- 1st matching function/script, based on function name only, looking first in current directory and then along path.

Function/Script Lookup Order (call in the body of a script s)

- Function in /private sub-directory of directory of last called function (not the /private sub-directory of the directory containing s).
- 1st matching function/script, based on function name, looking first in current directory and then along path.

Variables and Data in MATLAB

% in s.m
...
foo(a);
...

dir1/
   f.m
   g.m
   private/
   foo.m

dir2/
   s.m
   h.m
   private/
   foo.m

MATLAB types: high-level

any
  data fnhandle
  array cell array struct

Variables

- Variables are not explicitly declared.
- Local variables are allocated in the current workspace. Global and persistent variables in a special workspace.
- All input and output parameters are local.
- Local variables are allocated upon their first definition or via a load statement.
  - x = ...
  - x(i) = ...
  - load ('f.mat', 'x')
- Local variables can hold data with different types at different places in a function/script.
Variable Workspaces
- There is a workspace for global and persistent variables.
- There is a workspace associated with the read-eval-print loop.
- Each function call creates a new workspace (stack frame).
- A script uses the workspace of its caller (either a function workspace or the read-eval-print workspace).

Variable Lookup
- If the variable has been declared global or persistent in the function body, look it up in the global/persistent workspace.
- Otherwise, lookup in the current workspace (either the read-eval-print workspace or the top-most function call workspace).
- For nested functions, use the standard scoping mechanisms.

Variable Lookup
- If the variable has been declared global or persistent in the function body, look it up in the global/persistent workspace.
- Otherwise, lookup in the current workspace (either the read-eval-print workspace or the top-most function call workspace).
- For nested functions, use the standard scoping mechanisms.

Other Tricky "features" in MATLAB
- Keyword `end` not always required at the end of a function (often missing in files with only one function).
- Command syntax:
  - `length('x')` or `length x`
  - `cd('mydirname')` or `cd mydirname`
- Arrays can be defined with or without commas:
  - `[10, 20, 30]` or `[10 20 30]`
- Sometimes newlines have meaning:
  - `a = [ 10 20 30 40 50 60 ]; // defines a 2x3 matrix`
  - `a = [ 10 20 30; 40 50 60 ]; // defines a 2x3 matrix`
  - `a = [ 10 20 30; 40 50 60 ]; // defines a 2x3 matrix`

"Evil" Dynamic Features
- Not all input arguments required
- `(function) (prod, sum) = ProdSumNargs( a, n )`
- `if nargin == 1 n = 1; end;`
- `...`
- `end`
- Do not need to use all output arguments
- `eval, evalin, assignin`
- `cd, addpath`
- `load`

Evil Feature of the Day - Looking up an identifier
- First lookup as a variable.
- If a variable not found, then look up as a function.

Old style general lookup - interpreter
- When function/script first loaded, assign a "kind" to each identifier. VAR – only lookup as a variable, FN – only lookup as a function, ID – use the old style general lookup.
- How is the kind assignment done. What impact does it have on the semantics?
McLab – Overall Structure

McLab Extensible Front-end

Analysis Engine

How does MATLAB resolve Names?

Motivating example
Evil Feature of the Day - Recap

Old style general lookup - interpreter
- First lookup as a variable.
- If a variable not found, then look up as a function.

MATLAB 7 lookup - JIT
- When function/script first loaded, statically assign a "kind" to each identifier. VAR – only lookup as a variable, FN – only lookup as a function, ID – use the old style general lookup.
- Compile-time error if, within the body of a function or script, an identifier has kind VAR in one place and FN in another.

Does the kind analysis change the semantics?
Yes, in two ways!

1. New compile-time errors, so programs that would previously execute will not.
2. Different binding at run-time for some identifiers which are assigned a kind of VAR or FN.

Compile-time kind error

Different lookup with old vs MATLAB 7 semantics

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

- Old interpreter semantics: 
  - sum, line 2, named function
  - sum, line 4, local variable
- MATLAB 7 semantics gives a static kind of FN to sum 
  - sum, line 2, named function
  - sum, line 4, named function

Our approach to the Kind Analysis Problem

- Identify that a kind analysis is needed to match MATLAB 7 semantics.
- Specify and implement a kind assignment algorithm that matches the observed behaviour of MATLAB 7. (both for functions and for scripts)
- Identify any weaknesses in the MATLAB 7 approach and suggest two more clearly defined alternatives, one flow-sensitive and one flow-insensitive.
- Determine if the alternatives could be used without significant change to the behaviour of existing MATLAB programs.
Collect all identifiers used in function/script and set initial kind approximations for each identifier.
2. Traverse AST applying analysis rules to identifiers.
3. Traverse AST making final kind assignment.

Step 2: Kind Analysis Rules

Definition of identifier $x$:

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

Use of identifier $x$:

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

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

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

Kind Analysis for Functions

- **Initial values**: input and output parameters are initialized to VAR, all other identifiers are initialized as UNDEF.

- **Final values**:  
  for each id occurrence in \( \{ \text{for}, \text{while}, \text{if} \} \)
    if \( \text{kind}[\text{id}] \in \{\text{ID}, \text{VAR}, \text{VF} \} \)
      \( \text{id} \cdot \text{kind} = \text{ID} \)
    else for \( \text{kind}[\text{id}] \in \{\text{VAR}, \text{FN} \} \)
      \( \text{id} \cdot \text{kind} = \text{VAR} \)

Kind Analysis for Scripts

- **Initial values**: all identifiers are initialized to MAYVAR

- **Final values**:  
  for each id occurrence in a do
  if \( \text{kind}[\text{id}] \in \{\text{VAR}, \text{MAYVAR} \} \)
    \( \text{id} \cdot \text{kind} = \text{ID} \)
  else for \( \text{kind}[\text{id}] \in \{\text{VAR}, \text{VF} \} \)
    if \( \text{id} \cdot \text{kind} = \text{FN} \), it can't be ID or UNDEF
    \( \text{id} \cdot \text{kind} = \text{FN} \)

- **Note**: most identifiers will be mapped to ID
Problems with MATLAB 7 kind analysis

- apparently not clearly documented, in some ways just a side-effect of a JIT implementation decision
- without a clear specification, confusing for the programmer and compiler/tool developer
- loses almost all information about variables in scripts
- some strange anomalies due to a "traversal-sensitive" analysis

Examples of Anomalies

```
if ( exp )
   ... = sum(10); (sum,FN)
else
   sum(10) = ...; *error*
```

```
if (~exp)
   sum(10) = ...; (sum,VAR)
else
   ... = sum(10); (sum,VAR)
```

```
size(size(10)) = ...
(size,VAR)
```

Flow-sensitive Analysis

```
if ( exp )
   ... = sum(10); (sum,FN)
else
   sum(10) = ...; (sum,VAR)
// merge. *error*
size(size(10)) = (size,FN)
```

- Apply a flow-sensitive analysis that merges at control-flow points.
- Consider explicit loads to be definitions -
  ```matlab
  load ('f.mat', 'x')
  ```
- Map final kinds for scripts using the same algorithm as for functions.

Results: What is the distribution of kinds for functions/scripts in real MATLAB programs?
Various-sized benchmarks from a wide variety of 
application areas

<table>
<thead>
<tr>
<th>Benchmark Category</th>
<th># Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single (1 file)</td>
<td>2051</td>
</tr>
<tr>
<td>Small (2-9 files)</td>
<td>848</td>
</tr>
<tr>
<td>Medium (10-49 files)</td>
<td>113</td>
</tr>
<tr>
<td>Large (50-99 files)</td>
<td>9</td>
</tr>
<tr>
<td>Very Large (&gt; 100 files)</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>3024</td>
</tr>
</tbody>
</table>

Send benchmarks or links to hendren@cs.mcgill.ca

Results for Functions - number of identifiers with each Kind

<table>
<thead>
<tr>
<th>Kind</th>
<th>MATLAB 7 Flow-Sens.</th>
<th>Flow-In sens.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>107388</td>
<td>107401</td>
</tr>
<tr>
<td>FN</td>
<td>75543</td>
<td>75543</td>
</tr>
<tr>
<td>ID</td>
<td>2335</td>
<td>2335</td>
</tr>
<tr>
<td>error</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>warn</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>185291</td>
<td>185291</td>
</tr>
</tbody>
</table>

11698 functions

Results for Scripts – number of identifier instances with each Kind

<table>
<thead>
<tr>
<th>Kind</th>
<th>MATLAB 7 new</th>
<th>MATLAB 7 post-process</th>
<th>Flow-sens.</th>
<th>Flow-In sens.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>18445</td>
<td>6</td>
<td>68410</td>
<td>68410</td>
</tr>
<tr>
<td>FN</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>222467</td>
<td>222467</td>
<td>222467</td>
<td>222467</td>
</tr>
<tr>
<td>error</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>warn</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>total</td>
<td>2035 scripts</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Overview – PART II

- What is an Aspect
- AspectMatlab
- Typing Aspects

What is an Aspect?

- Pattern specifying events to match.
- Action to do before, after or around the matched events.
- Action can use context information from the matched event.

Example: Profiling Array Sparsity

- Capture the sparsity and size at each operation on the whole array.
- Capture the number of indexed references to each array.
- Print out a summary for each array, allowing the programmer to identify good candidates to implement as sparse arrays.
Background - MATLAB Class

classdef myClass
    properties
        count = 0;
    end
    methods
        helper functions
        function x = getCount(this)
            x = this.count;
        end
    end
end

Aspect Definition

aspect myAspect
    properties
        count = 0;
    end
    methods
        helper functions
        function x = getCount(this)
            x = this.count;
        end
    end
end

Function and Operator Patterns

patterns
    pCallFoo : call(foo);
    pExecBar : execution(bar);
    pCallFooArgs : call(foo(*,*,*));
    pExecutionMain : mainexecution();
end

patterns
    plusOp : op(+);
    timesOp : op(*); || op(*);
    matrixOps: op(matrix);
    allButMinus: op(all) & ~op(-);
end

Array Patterns

context info

a(i) = b(j,k)

patterns
    pSetX : set(a);
    pGetX : get(b);
    arraySet : set(*);
    arrayWholeGet : get(*);
    arrayIndexedGet : get(*);
end

Loop Patterns

patterns
    for i = 1:2:n
        ...
        t1 = [1,3,5,7,9,...,n];
    end

patterns
    pLoopI : loop(i);
    pLoopHeadI : loophead(i);
    pLoopBodyI : loopbody(i);
end

Scope Patterns

patterns
    pWithinFoo : within(function, foo);
    pWithinBar : within(script, bar);
    pWithinMyClass : within(class, myClass);
    pWithinLoops : within(loops, 'i');
    pWithinAllAbc : within(*, abc);
end
**Compound Patterns**

- Logical combinations of primitive patterns

```
pCallFoo : call(foo) & within(loops, *);
pGetOrSet : (get(*) | set(*)) & within(function, bar); end
```

**Before & After Actions**

```
actions
aCountCall : before pCall
    this.count = this.count + 1;
    disp('calling a function');
end

aExecution : after executionMain
    total = this.getCount();
    disp(['total calls: ', num2str(total)]);
end
```

**Context Exposure**

```
actions
aCountCall : before pCall : (name, args)
    this.count = this.count + 1;
    disp(['calling ', name,' with args(', args, ')']);
end

aExecution : after executionMain : (file)
    total = this.getCount();
    disp(['total calls in ', file,' : ',num2str(total)]);
end
```

**Around Actions**

```
actions
actcall : around pCallFoo : (args)
    varargout{1} = bar(args{1}, args{2});
end

actcall : around pCallFoo : (args)
    disp(['before foo call with args(', args, ')']);
    proceed();
    disp(['after foo call with args(', args, ')']);
end
```

**Actions Weaving Order**

```
actions
before1 : before pCallFoo
    before2();
around1 : around pCallFoo
    around2();
after1 : after pCallFoo
    after2();
before2 : before pCallFoo
    before2();
around2 : around pCallFoo
    around2();
after2 : after pCallFoo
    end
```

**Compiler Structure**

```
```

```
```
**Compiler Structure**

- **Front-end**
  - Matlab + Aspects
  - Resolved Name Set
- **Parser**
  - Weaved Base Matlab
  - Matlab AST
- **Matcher & Weaver**
  - Woven AST
  -Base Matlab
  - Matlab Impl.
of Aspects
- **Matcher**
  - Resolved Name Set
  - Transformations
  - Simplified AST

**Scientific Use Cases**

- **Domain-Specific Profiling of Programs**
  - Tracking array sparsity
  - Tracking array size-growing operations
  - Counting floating-point operations
- **Extending Functionality**
  - Interpreting loop iteration space
  - Adding units to computations

**Typing Aspects**

- Types for MATLAB, somewhat in the spirit of aspects.
- Designed by what programmers might want to say.
- Checked at run-time, but some static analysis could be done.

**Simple Example MATLAB function**

```matlab
function [ r ] = Ex1( n )
% Ex1(n) creates a vector of n values containing the values [sin(1), sin(2), ..., sin(n)]
for i=1:n
    r(i) = sin(i);
end
```

```matlab
>> Ex1(3)
an = 0.8415 0.9093 0.1411
>> Ex1(2.3)
an = 0.8415 0.9093
```
MATLAB programmers often expect certain types

```matlab
function y = sturm(X, BC, F, G, R)
% STURM Solve the Sturm–Liouville equation:
% d( F * dY/dX )/dX - G*Y = R using linear finite elements.
% INPUT:
% X - a one-dimensional grid-point array of length N.
% BC - is a 2 by 3 matrix [A1, B1, C1 ; An, Bn, Cn]
% ... % Alex Pletzer: pletzer@pppl.gov (Aug. 97/July 99).
% ... %

% type('n','scalar of Float');
for i=1:n
r(i) = sin(i);
end
% type('r','array [n.value] of n.basetype');
```

High-level types in MATLAB

- any
- data
- fnhandle
- array
- cellarray
- struct

MATLAB prorammers often expect certain types

```matlab
function [ r ] = Ex1( n )
% Ex1(n) creates a vector of n values containing
% the values [sin(1), sin(2), ..., sin(n)]
% atype('n','scalar of Float');
for i=1:n
r(i) = sin(i);
end
% atype('r','array [n.value] of n.basetype');
```
**Simple Example**

```matlab
function [ r ] = foo( a, b, c, d )
atype('a', 'array [...] of int');
atype('b', 'array[*,*]');
atype('c', 'array[*,*,...] of complex');
atype('d', 'scalar of uint32');
% ...
% body of foo
% ...
atype('r','array[a.dims] of int');
end
```

**Capturing reflective information**

```matlab
function [ r ] = foo( a )
atype('a','any');
% ...
% body of foo
% ...
atype('r','a.type');
end
```

**Capturing dimensions and basetype**

```matlab
function [ r ] = foo( a, b )
atype('a','array[<n>,<m>] of real');
atype('b','array[a.m,<p>] of a.basetype');
% ...
% body of foo
% ...
atype('r','array[a.m, b.p] of a.basetype');
end
```

**Conclusions**

- MATLAB is an important language, but presents challenges for compiler writers
- McLAB provides toolkits for analysis and transformation of MATLAB
- McLAB is extensible, AspectMatlab is one extension
- Typing Aspects is another possible extension.