*Mc*LAB: A toolkit for static and dynamic compilers for MATLAB

Amina Aslam
Toheed Aslam
Andrew Casey
Maxime Chevalier- Boisvert
Jesse Doherty
Anton Dubrau
Rahul Garg
Maja Frydrychowicz
Nurudeen Lameed
Jun Li
Soroush Radpour
Olivier Savary

Laurie Hendren
McGill University

Leverhulme Visiting Professor
Department of Computer Science
University of Oxford

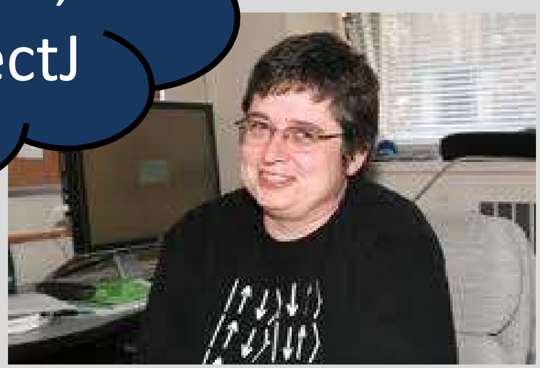
Overview

- Why MATLAB?
- Overview of the McLAB tools
- McVM – a Virtual Machine and Just-In-Time (JIT) compiler
- McFOR – translating MATLAB to FORTRAN95

FORTRAN
C/C++

MATLAB
PERL
Python
Domain-specific

C, Parallel C,
Java, AspectJ



Culture Gap

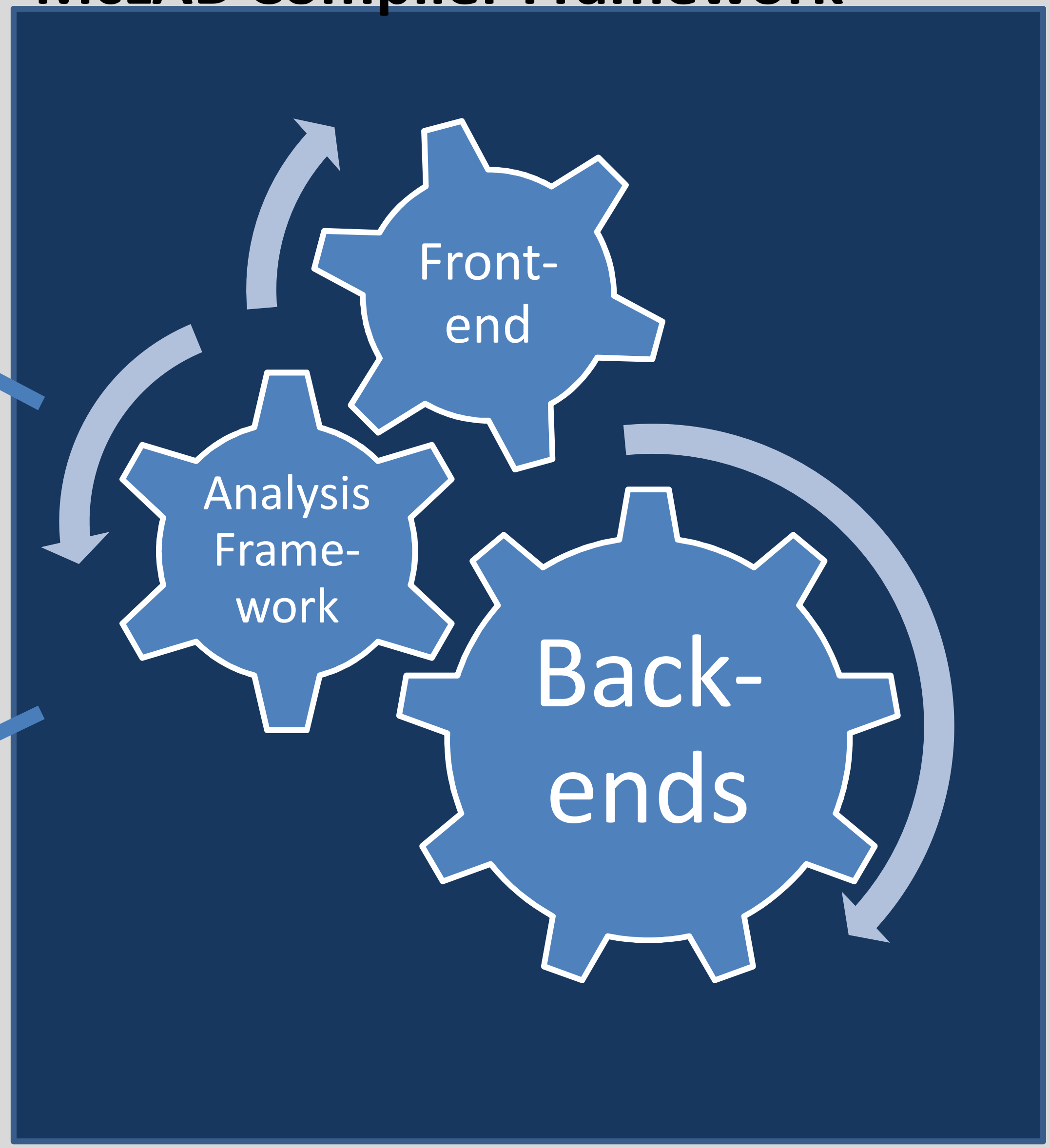
Scientists / Engineers

- 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.

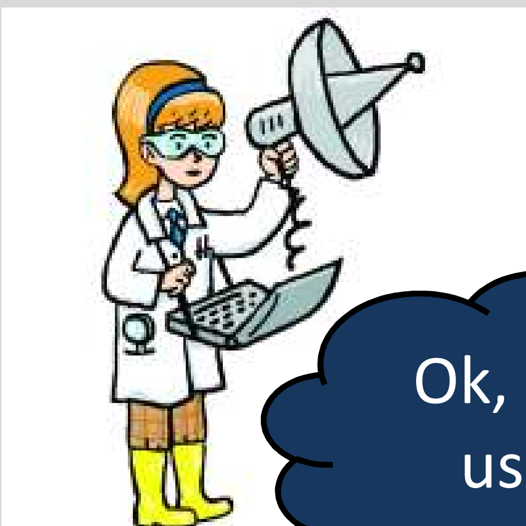
Programming Language / Compiler Researchers

- 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.

McLAB Compiler Framework



Ok, I can deal with this!



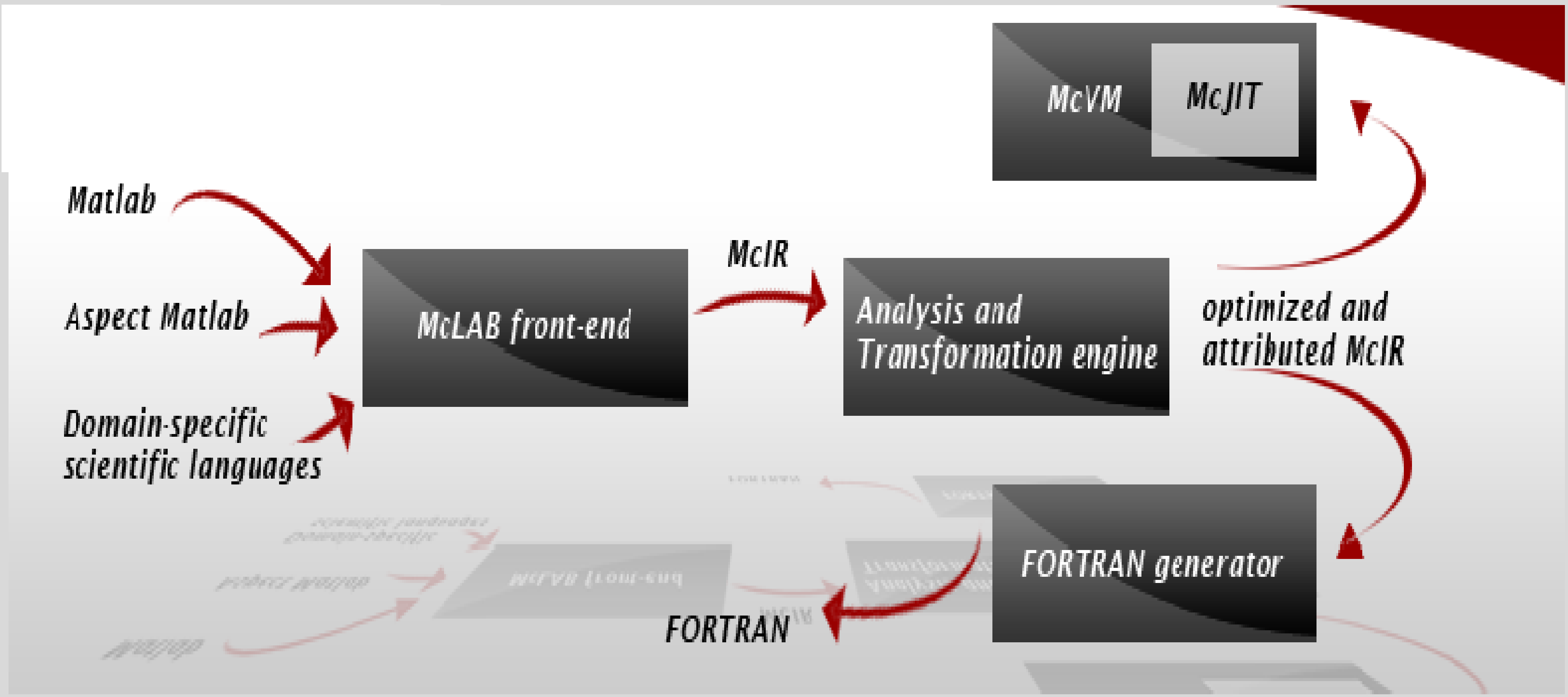
Ok, this is useful!

Goals of the McLab Project

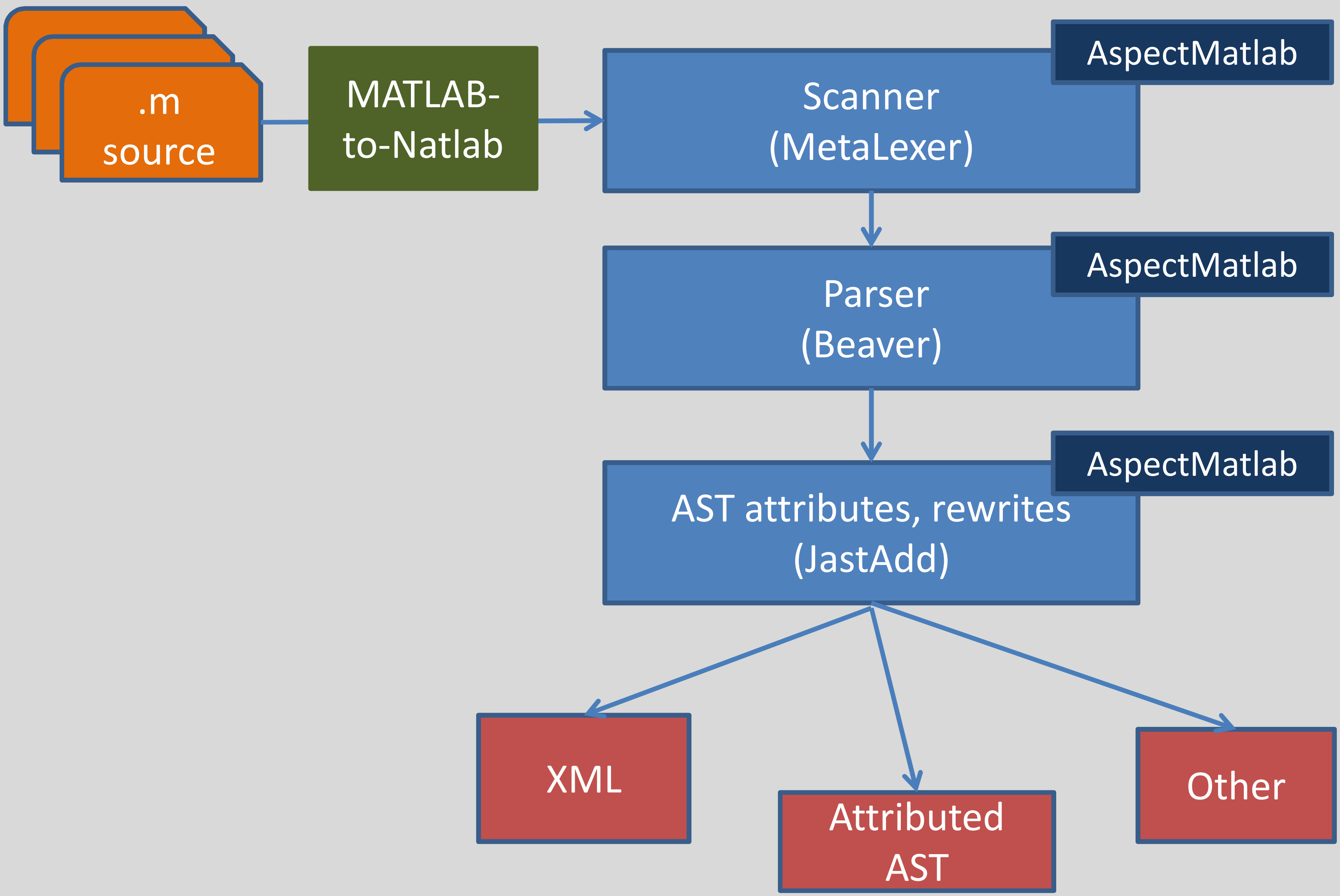
- Improve the understanding and documentation of the semantics of MATLAB.
- Provide front-end compiler tools suitable for MATLAB and language extensions of MATLAB.
- Provide a flow-analysis framework and a suite of analyses suitable for a wide range of compiler/soft. eng. applications.
- Provide back-ends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

Enable PL, Compiler and SE Researchers to work on MATLAB

McLAB – Overall Structure

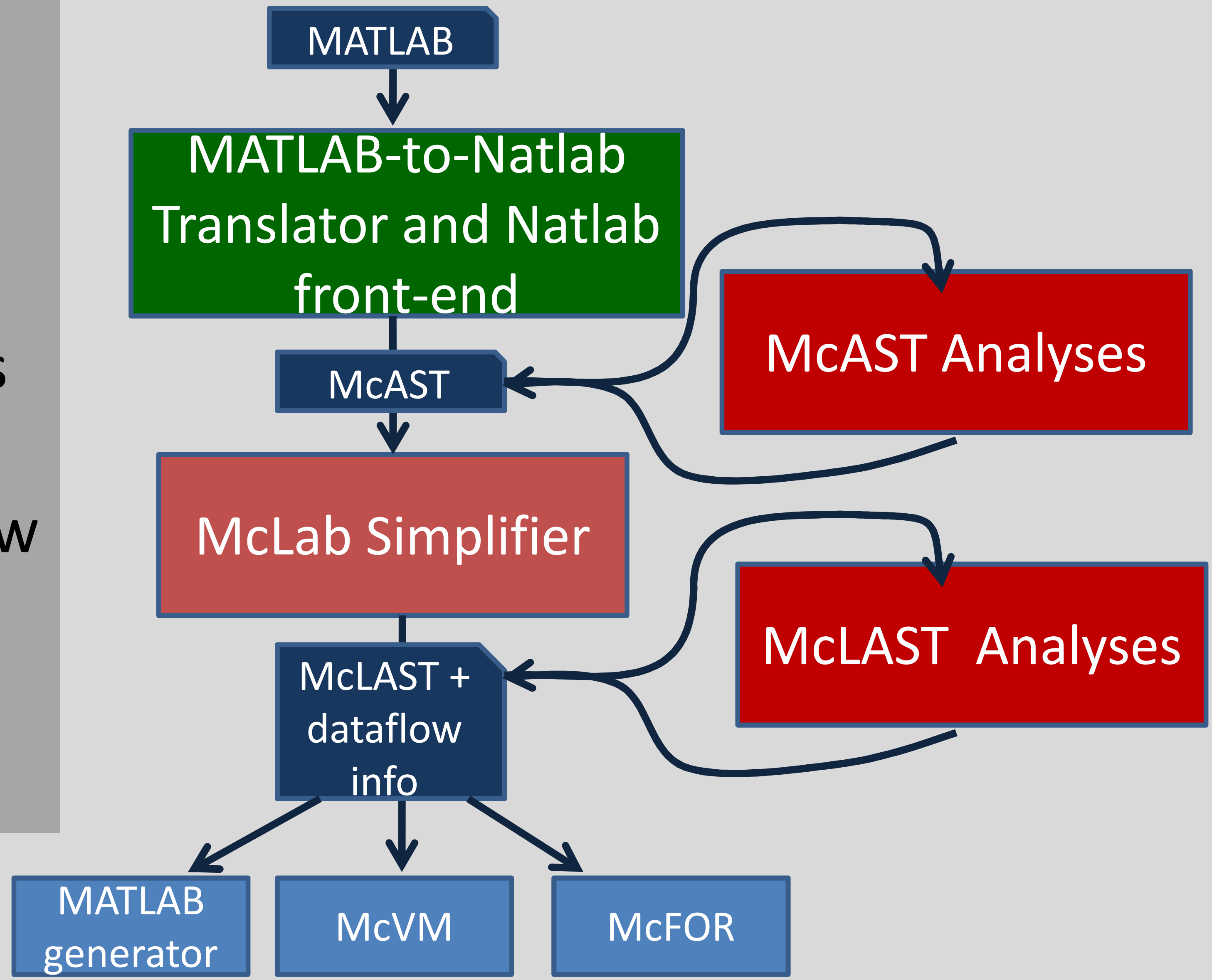


McLAB Extensible Front-end

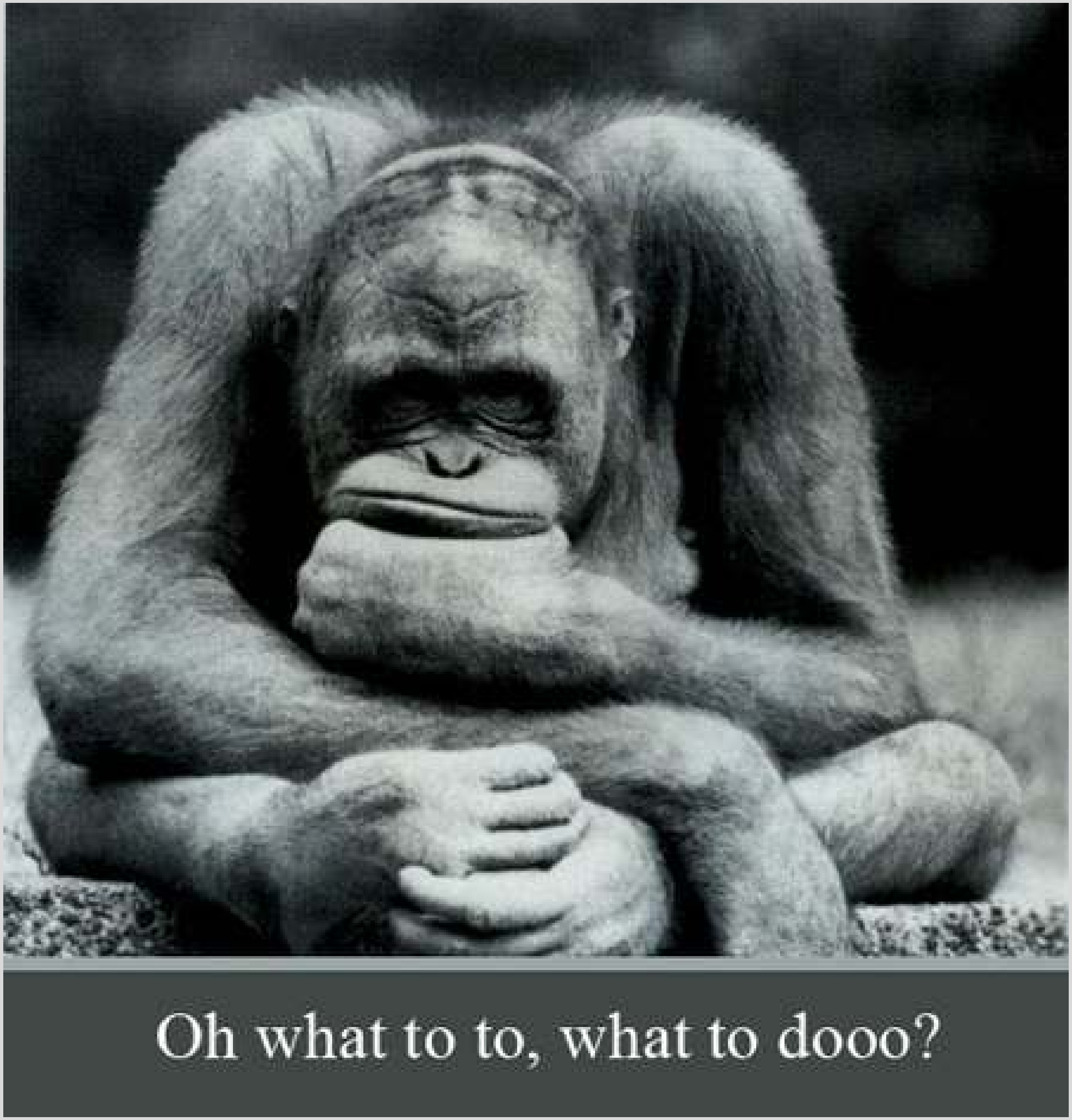


Analysis Engine

Analyses are written using an Analysis Framework that supports forward and backward flow analysis over McAST and McLAST.



How to build the back-ends?



- No official language specification.
- Closed-source implementations, including the library.

Back-end #1: MATLAB generator

- McLAB can be used as a source-to-source translator:
  - source-level optimizations like loop unrolling
  - source-level refactoring tool
- McLAB can generate:
  - xml files (used to communicate with McVM)
  - text .m files
- Comments are maintained to enable readable output.

Back-end #2: McVM with JIT compiler

- Based on Maxime Chevalier-Boisvert's M.Sc. thesis, McGill, published in CC 2010.
- Currently being extended by Nurudeen Lameed and Rahul Garg, Ph.D. candidates, McGill.

McVM-McJIT

- The dynamic nature of MATLAB makes it very suitable for a VM/JIT.
- MathWorks' implementation does have a JIT, although technical details are not known.
- McVM/McJIT is an open implementation aimed at supporting research into dynamic optimization techniques for MATLAB.

Design Choices for JITs

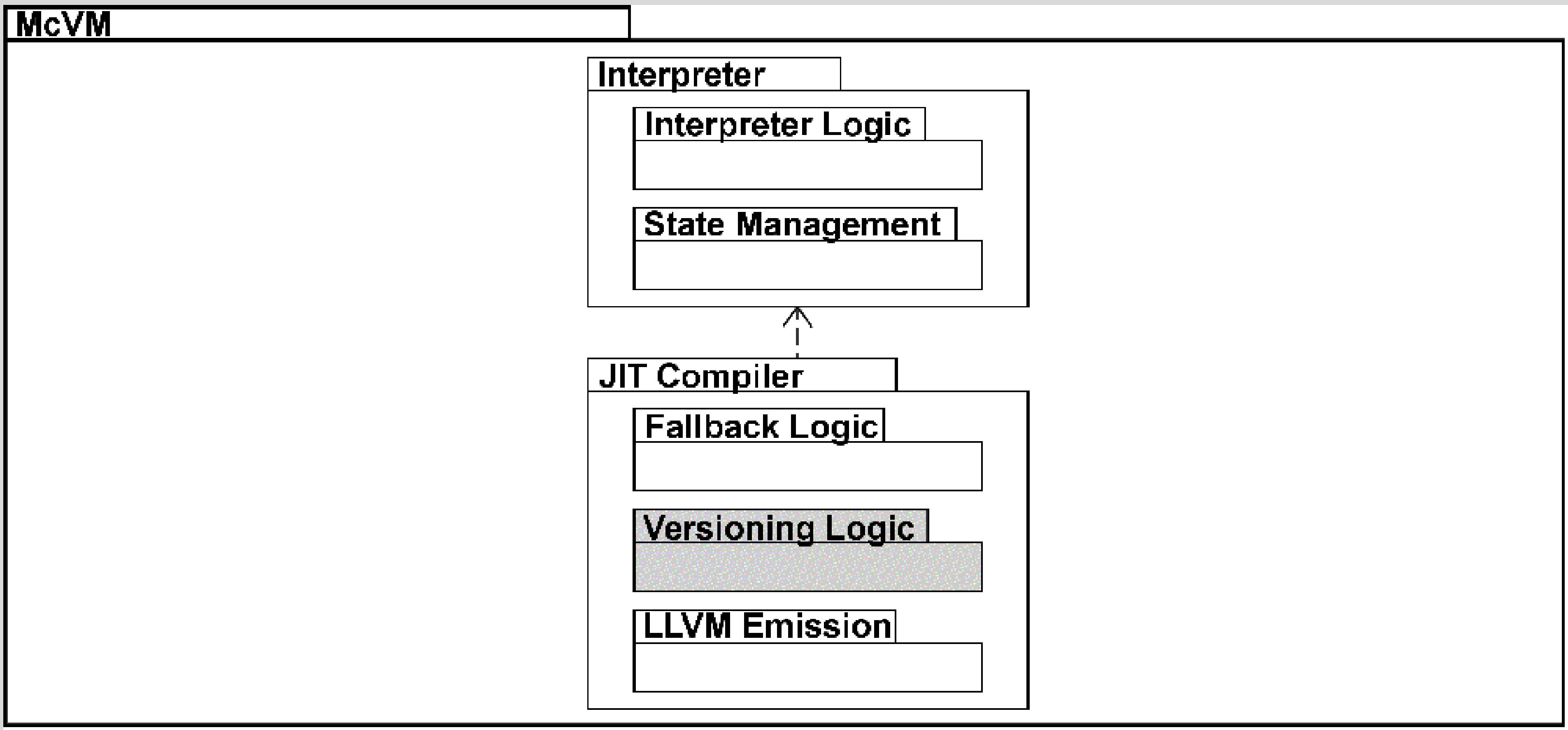
1. Interpreter + JIT compiler with various levels of optimizations.
2. Fast JIT for naïve code generation + optimizing JIT with various levels of optimizations.

- McVM uses the 1st option because it simplifies adding new features, if a feature is not yet supported by the JIT it can back-up to the interpreter implementation, which is easy to provide.

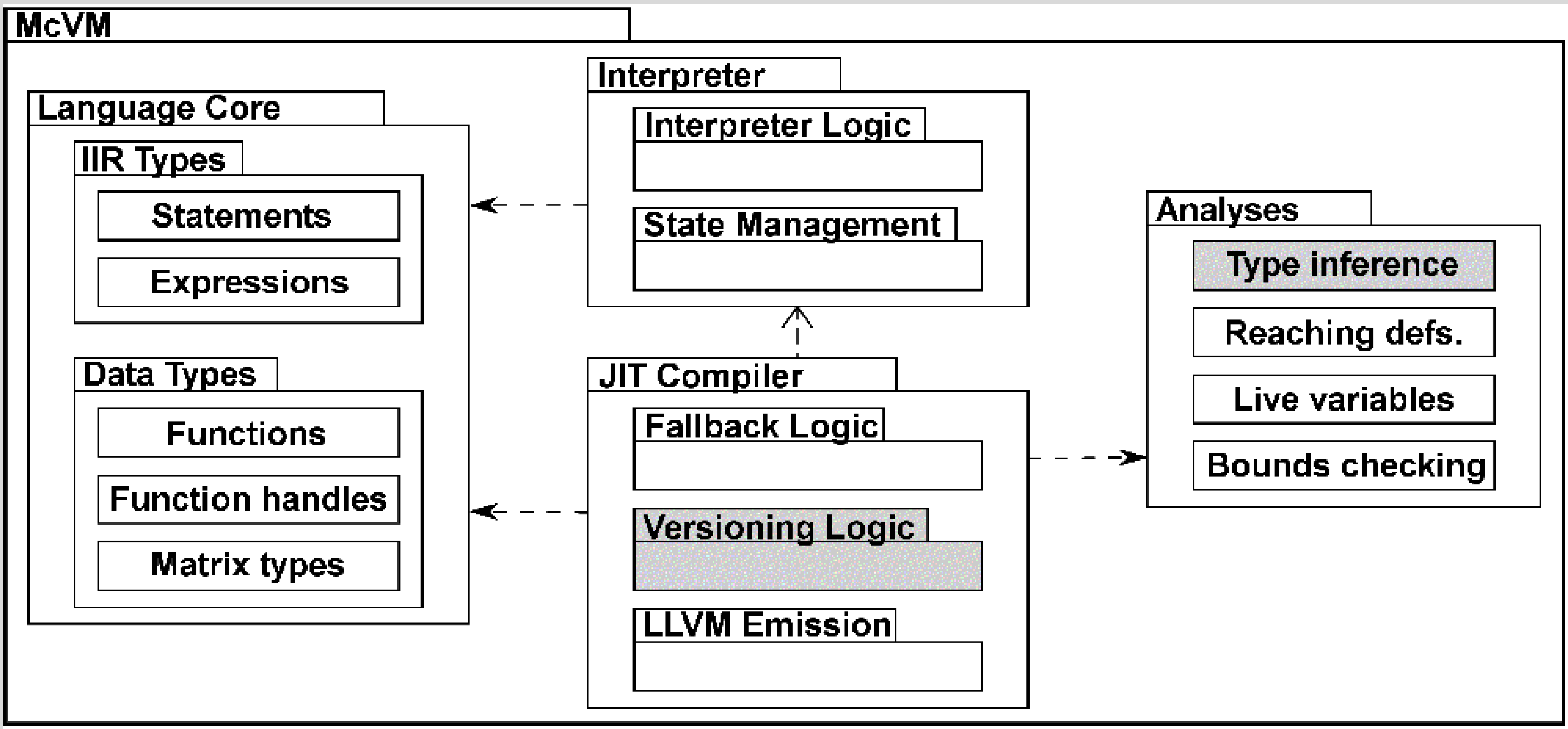
McVM Design

- A basic, but fast, interpreter for the MATLAB language.
- A garbage-collected JIT Compiler as an extension to the interpreter.
- Easy to add new data types and statements by modifying only the interpreter.
- Supported by the LLVM compiler framework and some numerical computing libraries.
- Written entirely in C++; interface with the McLAB front-end via a network port.

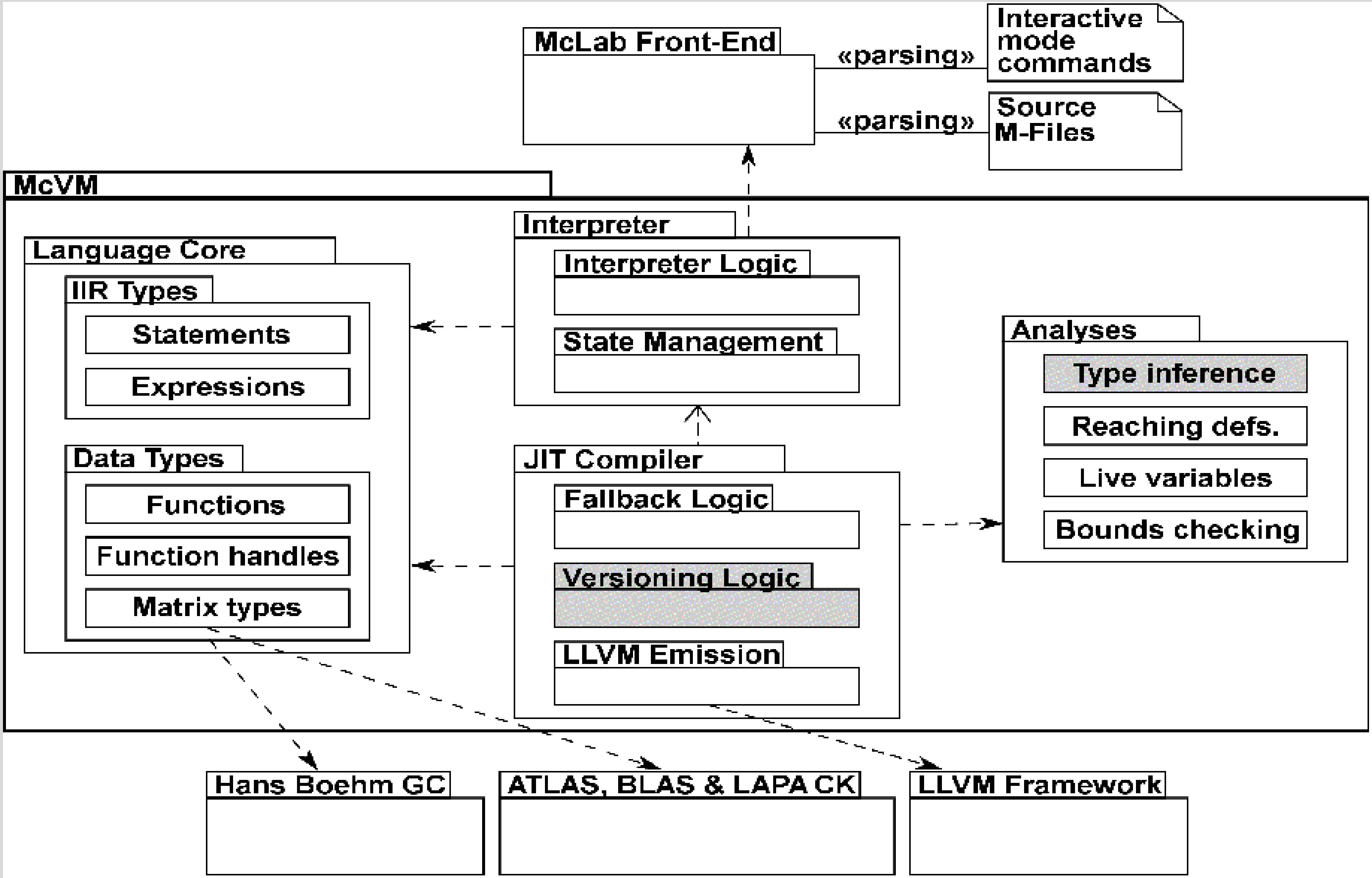
McVM Organization



McVM Organization



McVM Organization



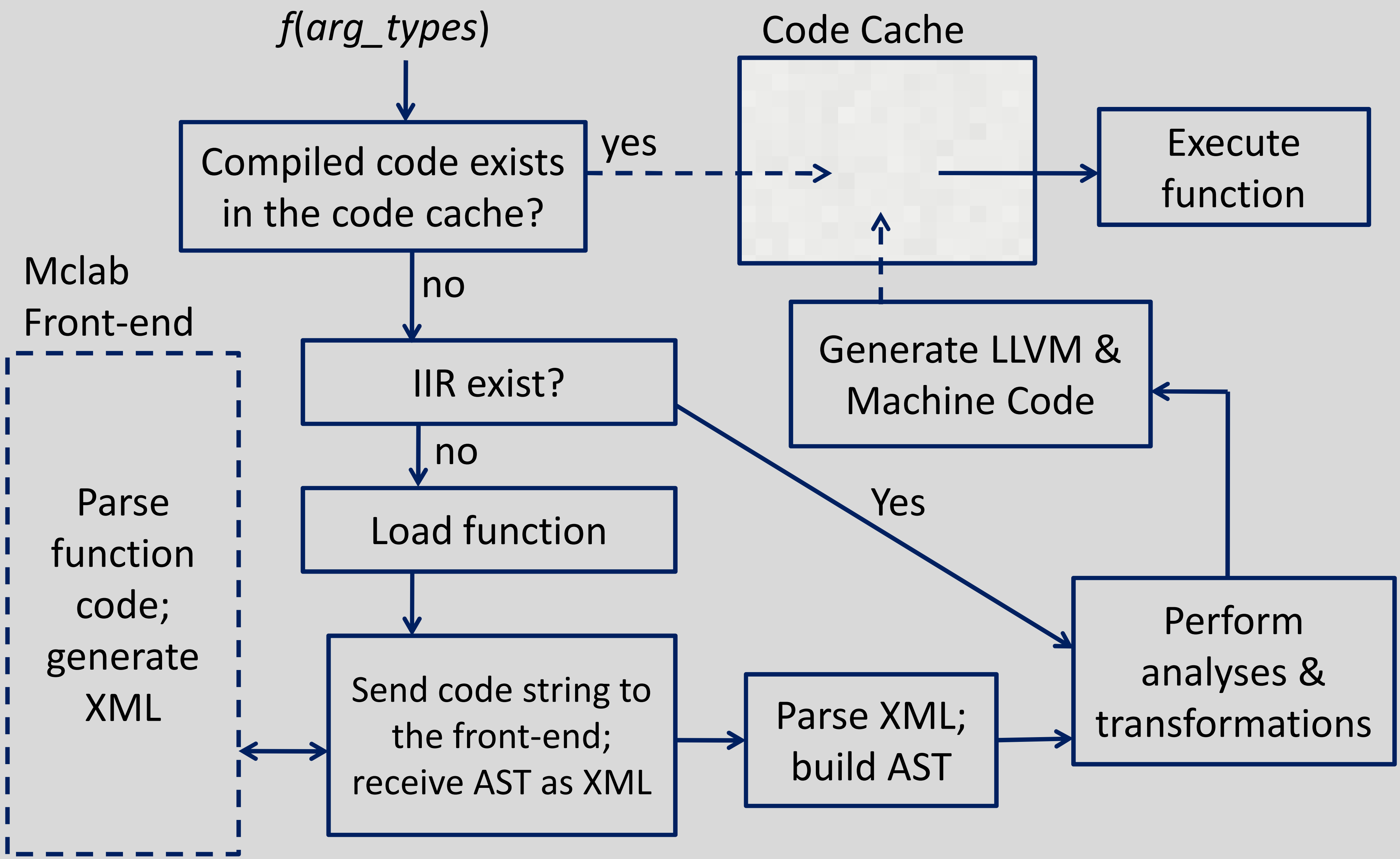
MATLAB Optimization Challenges

```
float sumvals(float start, float step, float
stop)
{
      float i = start;
      float s = i;

      while (i < stop)
      {
            i = i + step;
            s = s + i;
      }

      return s;
}
```

MATLAB Optimization Challenges

```
function s = sumvals(start, step, stop)
      i = start;
      s = i;

      while i < stop
            i = i + step;
            s = s + i;
      end
end

function caller()
      a = sumvals(1, 1, 10^6);
      b = sumvals([1 2], [1.5 3], [20^5 20^5]);
      c = [a b];

      disp(c);
end
```

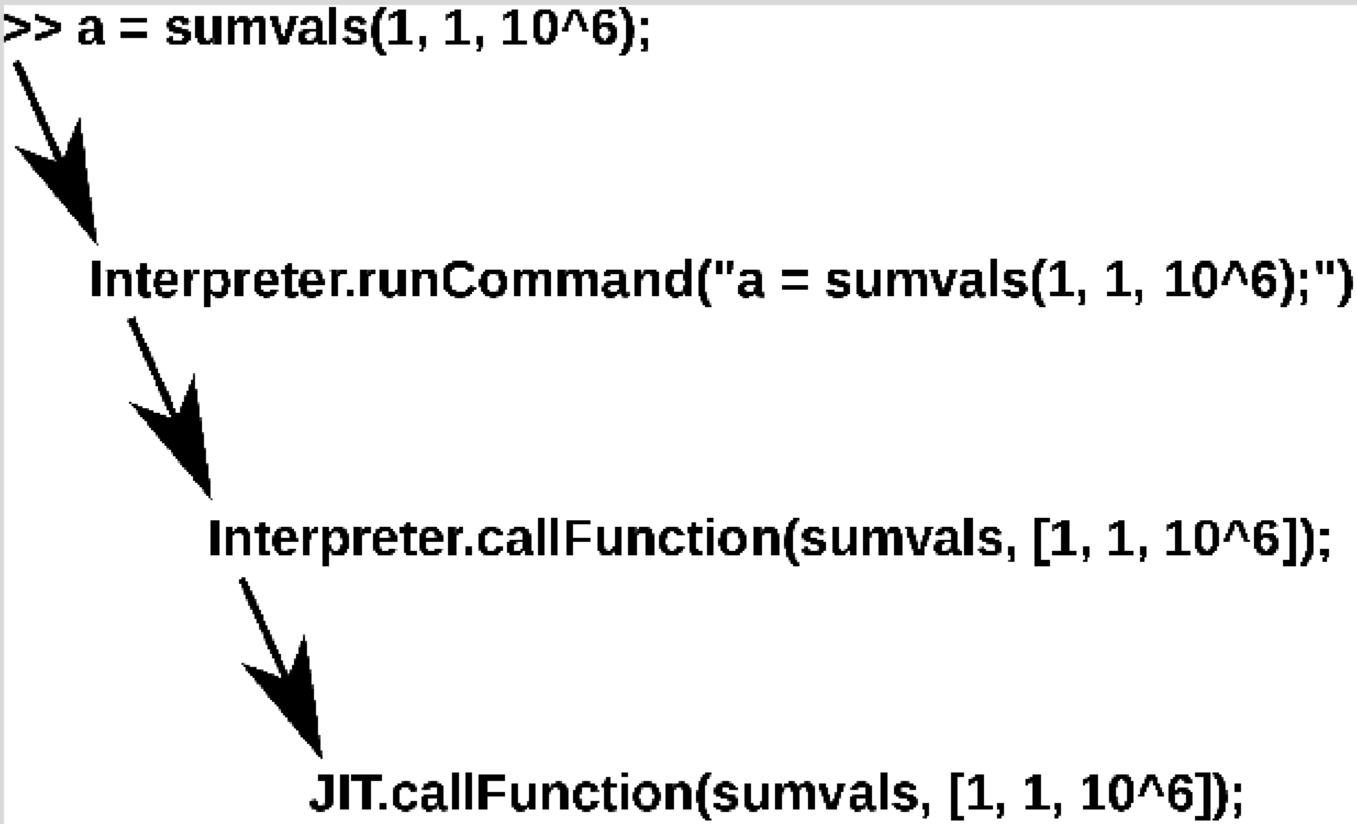
McJIT: Executing a Function



Just-In-Time Specialization (1)

```
>> a = sumvals(1, 1, 10^6);

>> b = sumvals([1 2], [1.5 3], [20^5 20^5]);

>> a = sumvals(1, 1, 500);

>> c = [a b];

>> disp(c);
```

Just-In-Time Specialization (2)



Just-In-Time Specialization (3)

```
JITCompiler.callFunction(sumvals, [1, 1, 10^6]);
```

```
sumvalsJIT = JITCompiler.compileFunction(sumvals, [<scalar int>, <scalar int>, <scalar int>]);
```

```
function s = sumvals(start <scalar int>, step <scalar int>,
    stop <scalar int>)

    i = start;
    s = i;

    while i < stop
        i = i + step;
        s = s + i;
    end
end
```

Just-In-Time Specialization (4)

```
JITCompiler.callFunction(sumvals, [1, 1, 10^6]);

    sumvalsJIT = JITCompiler.compileFunction(sumvals, [<scalar int>, <scalar int>, <scalar int>]);

        function s <scalar int> = sumvals(start <scalar int>, step <scalar int>,
            stop <scalar int>)

            i <scalar int> = start;
            s <scalar int> = i;

            while i < stop
                i <scalar int> = i + step;
                s <scalar int> = s + i;
            end
        end
```

Just-In-Time Specialization - 2nd example

```
>> a = sumvals(1, 1, 10^6);

>> b = sumvals([1 2], [1.5 3], [20^5 20^5]);

>> a = sumvals(1, 1, 500);

>> c = [a b];

>> disp(c);
```

JIT – second specialization (1)

```
JITCompiler.callFunction(sumvals, [[1 2], [1.5 3], [20^5 20^5]]);

  sumvalsJIT2 = JITCompiler.compileFunction(sumvals, [<1x2 int>, <1x2 real>, <1x2 int>]);

    function s = sumvals(start <1x2 int>, step <1x2 real>,
        stop <1x2 int>)

        i <1x2 int> = start;
        s <1x2 int> = i;

        while (i <1x2 int>) < (stop <1x2 int>)
            i <1x2 real> = i + step;
            s <1x2 real> = s + i;
        end
    end
```

JIT – second specialization (2)

```
JITCompiler.callFunction(sumvals, [[1 2], [1.5 3], [20^5 20^5]]);

   sumvalsJIT2 = JITCompiler.compileFunction(sumvals, [<1x2 int>, <1x2 real>, <1x2 int>]);

      function s <1x2 real> = sumvals(start <1x2 int>, step <1x2 real>,
          stop <1x2 int>)

          i <1x2 int> = start;
          s <1x2 int> = i;

          while (i <1x2 real>) < (stop <1x2 int>)
              i <1x2 real> = i + step;
              s <1x2 real> = s + i;
          end
      end
```

JIT – third specialization same as first

```
>> a = sumvals(1, 1, 10^6);

>> b = sumvals([1 2], [1.5 3], [20^5 20^5]);

>> a = sumvals(1, 1, 500);

>> c = [a b];

>> disp(c);
```

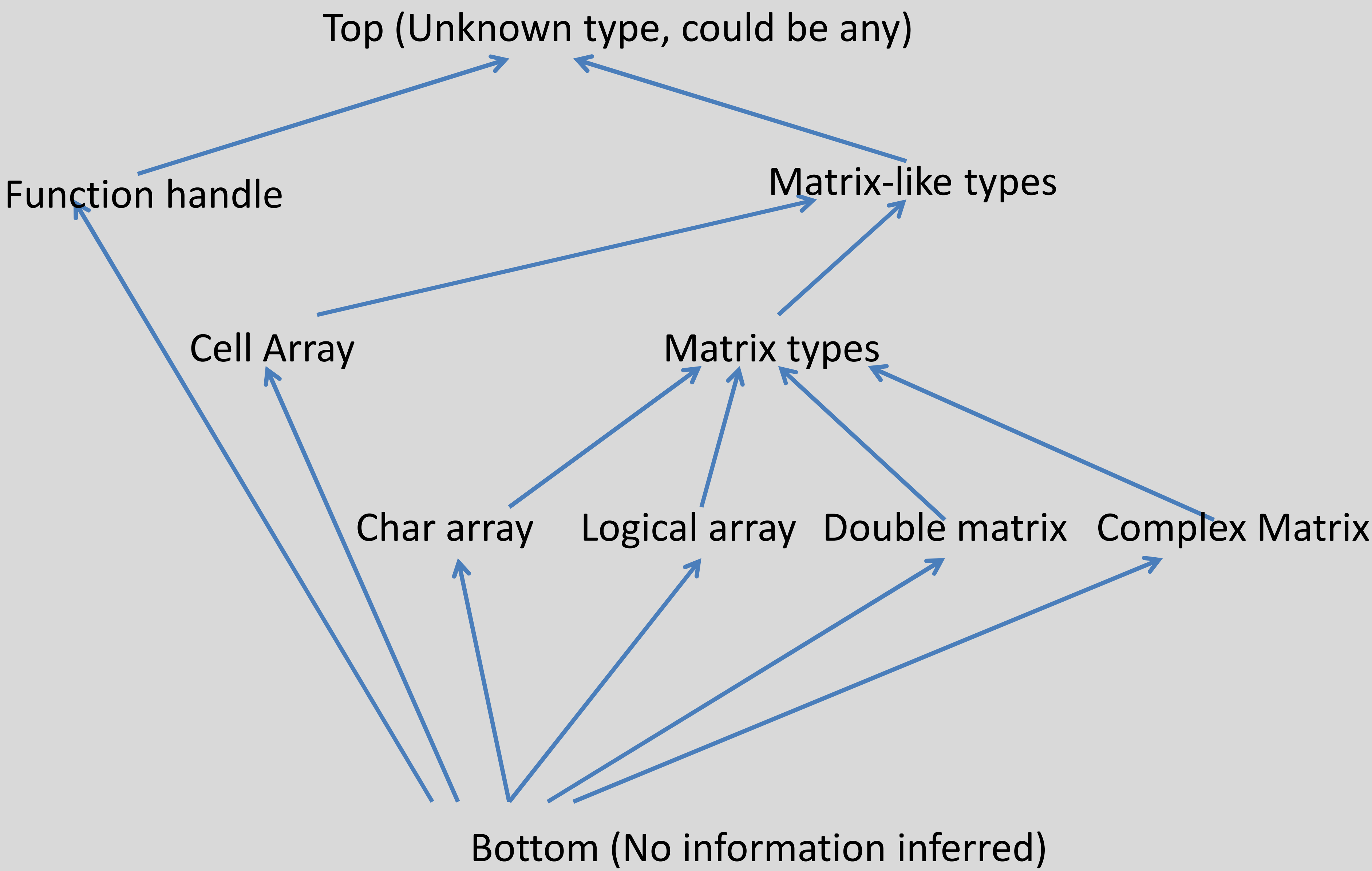
Type and Shape Inference

- In MATLAB, work with incomplete information
  - Dynamic loading: working with incomplete program
  - Dynamic typing : variables can change type
- Know argument types, what can we infer?
  - Propagate type info to deduce locals type and return type
- Forward dataflow analysis
  - Based on abstract interpretation
  - Structure-based fixed point
  - Annotates AST with type info

Flow Analysis Summary

- Start from known argument types
- Propagate type information forward
- Use a transfer function for each expression
  - Transfer functions provided for all primitive operators
  - Library functions provide their own transfer functions
  - Function calls resolved, recursively inferred
- Assignment statements can change var. types
- Merge operator
  - Union + filtering

Lattice of McVM types



Type Abstraction Properties

- Collection of simple abstractions
  - Specific features computed in parallel
- Represent variable types with 8-tuples:

```
<overallType, is2D, isScalar, isInteger,
     sizeKnown, size, handle, cellTypes>
```

Type Abstraction Properties

```
a = [1 2];
```

```
type(a) =
      <overallType = double, is2D = T, isScalar = F,
      isInteger = T, sizeKnown = T, size = (1,2),
      handle = null, cellTypes = {}>
```

Variable Types: Type Sets

```
if (c1)
    a = [1 2];
elseif (c2)
    a = 1.5;
else
    a = 'a';
end
```

```
type(a) = {
<overallType = double, is2D = T, isScalar = F,
  isInteger = T, sizeKnown = T,  size = (1,2),
  handle = null, cellTypes = {}>,
<overallType = double, is2D = T, isScalar = T,
  isInteger = F, sizeKnown = T,  size = (1,1),
  handle = null, cellTypes = {}>,
<overallType = char, is2D = T, isScalar = T,
  isInteger = T, sizeKnown = T,  size = (1,1),
  handle = null, cellTypes = {}>
}
```

Type Set Filtering

```
type(a) = {
  <overallType = double, is2D = T, isScalar = F,
   isInteger = T, sizeKnown = T, size = (1,2),
   handle = null, cellTypes = {}>,

  <overallType = double, is2D = T, isScalar = T,
   isInteger = F, sizeKnown = T, size = (1,1),
   handle = null, cellTypes = {}>
}
```

Becomes:

```
type(a) = {
  <overallType = double, is2D = T, isScalar = F,
   isInteger = F, sizeKnown = F, size = (),
   handle = null, cellTypes = {}>
}
```

Transfer Functions

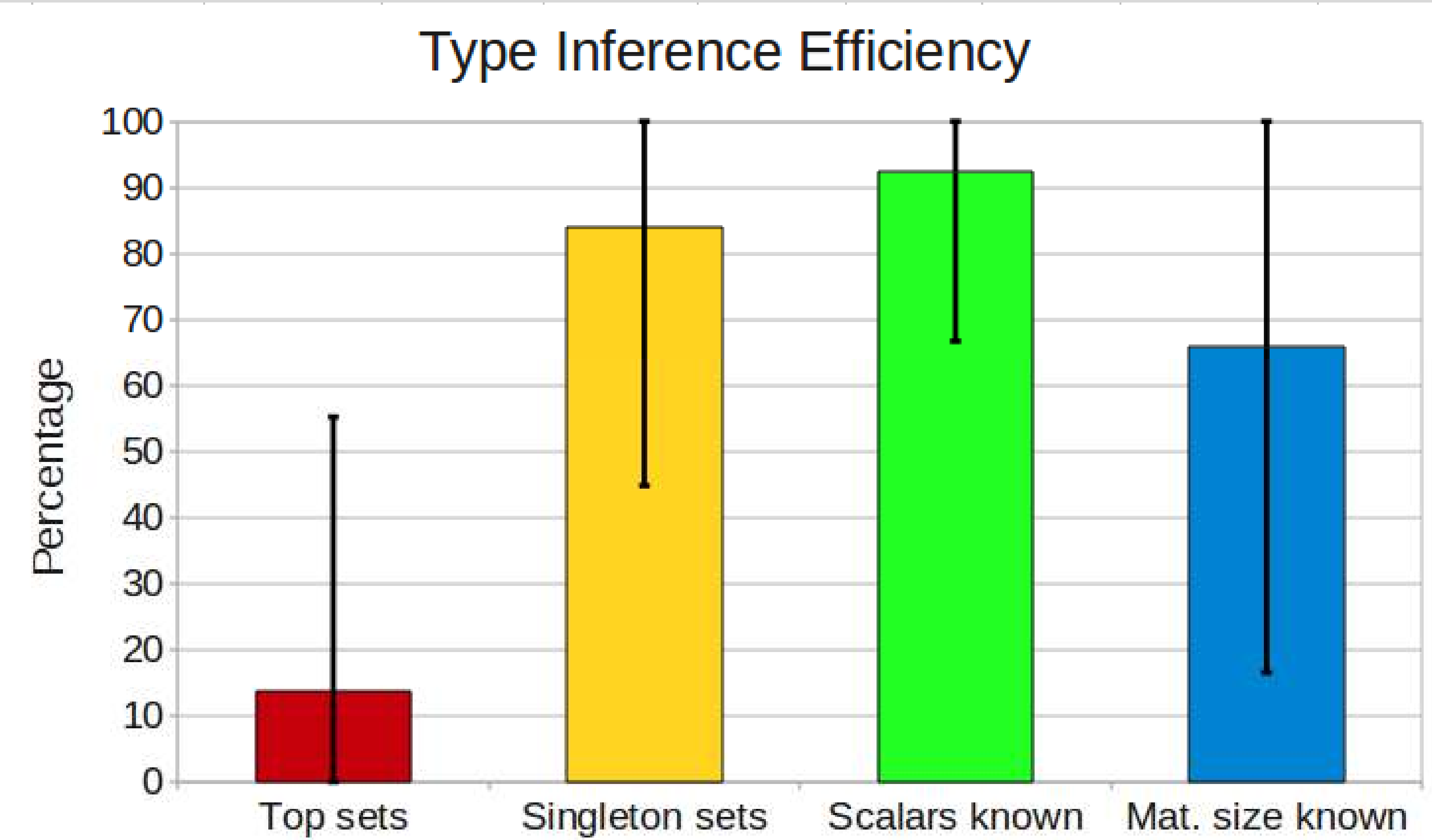
```
a = 5 * 10e11;
b = 1.0e12 * [0.8533 1.7067];
c = [a b];
```

```
type(a) = {
  <overallType = double, is2D = T, isScalar = T, isInteger = T,
     sizeKnown = T, size = (1,1), handle = null, cellTypes = {}>
}

type(b) = {
  <overallType = double, is2D = T, isScalar = F, isInteger = F,
     sizeKnown = T, size = (1,2), handle = null, cellTypes = {}>
}

type([a b]) = {
  <overallType = double, is2D = T, isScalar = F,isInteger = F,
     sizeKnown = T, size = (1,3), handle = null, cellTypes = {}>
}
```

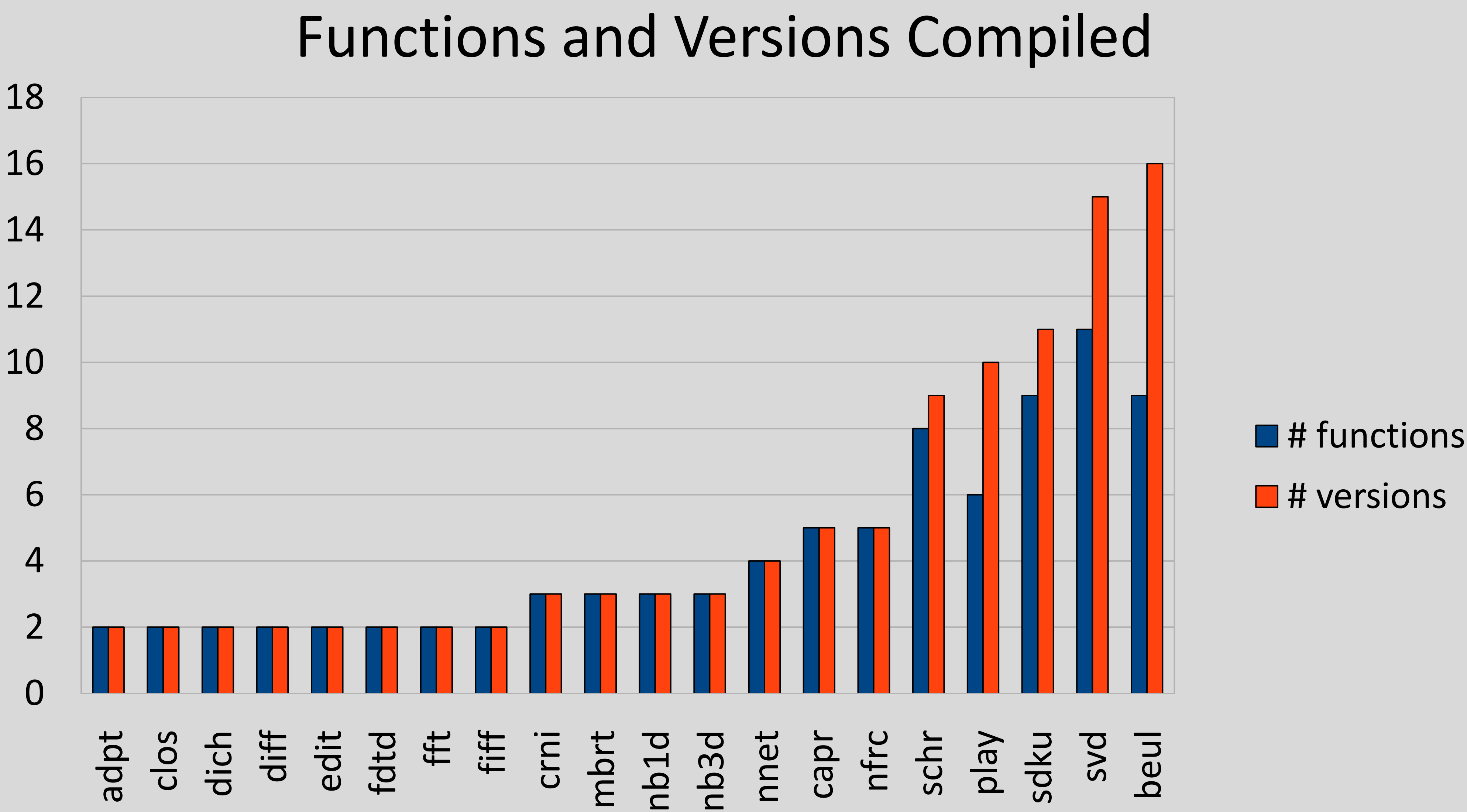
Experimental Results

- 20 benchmark programs
  - FALCON, OTTER, etc. Some made by McLAB
- Measured
  - Dynamic availability of type info.
  - Number of versions compiled
  - Compilation time
    - 0.55s per benchmark, on average

Results of Type Analysis



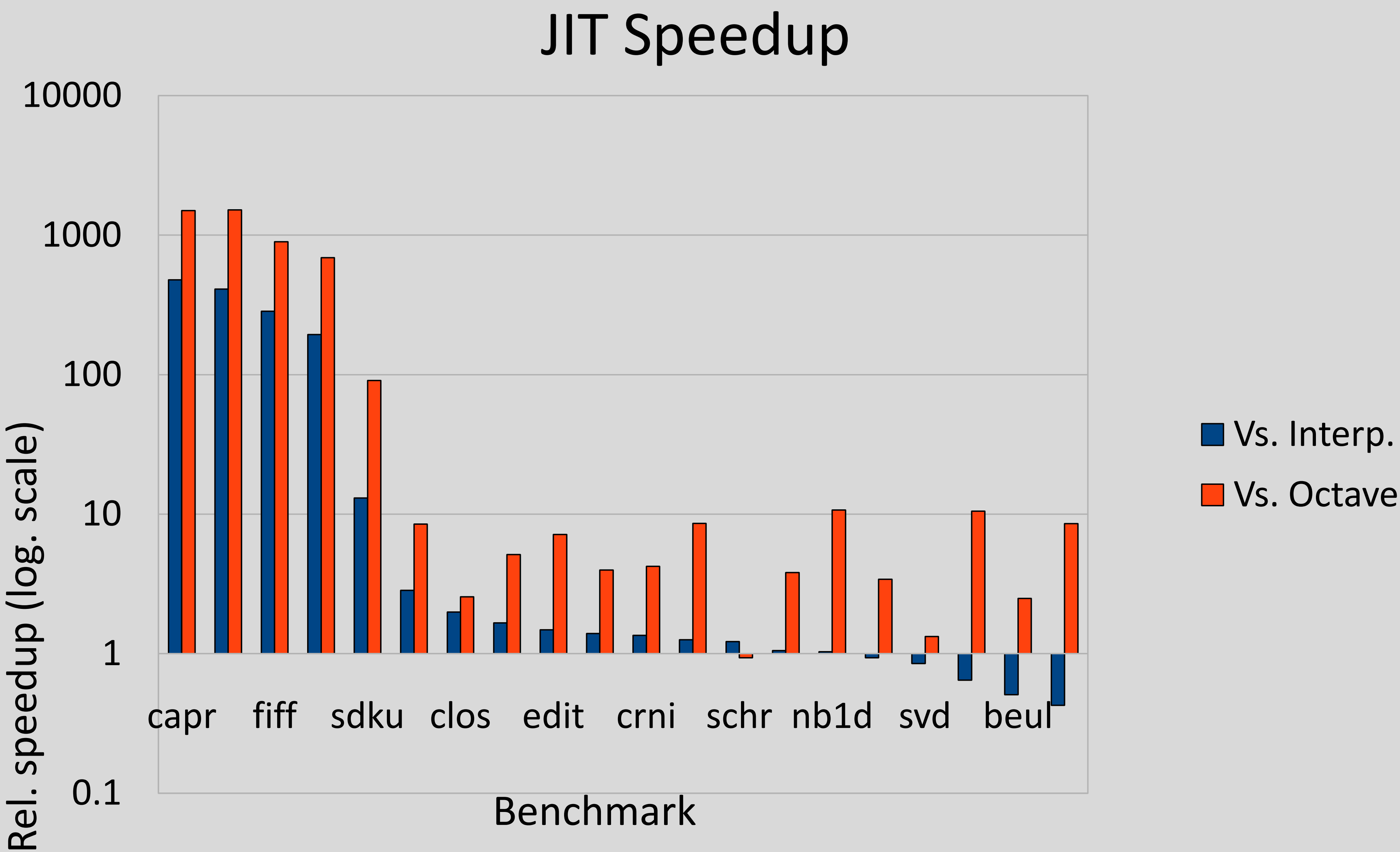
How many versions...



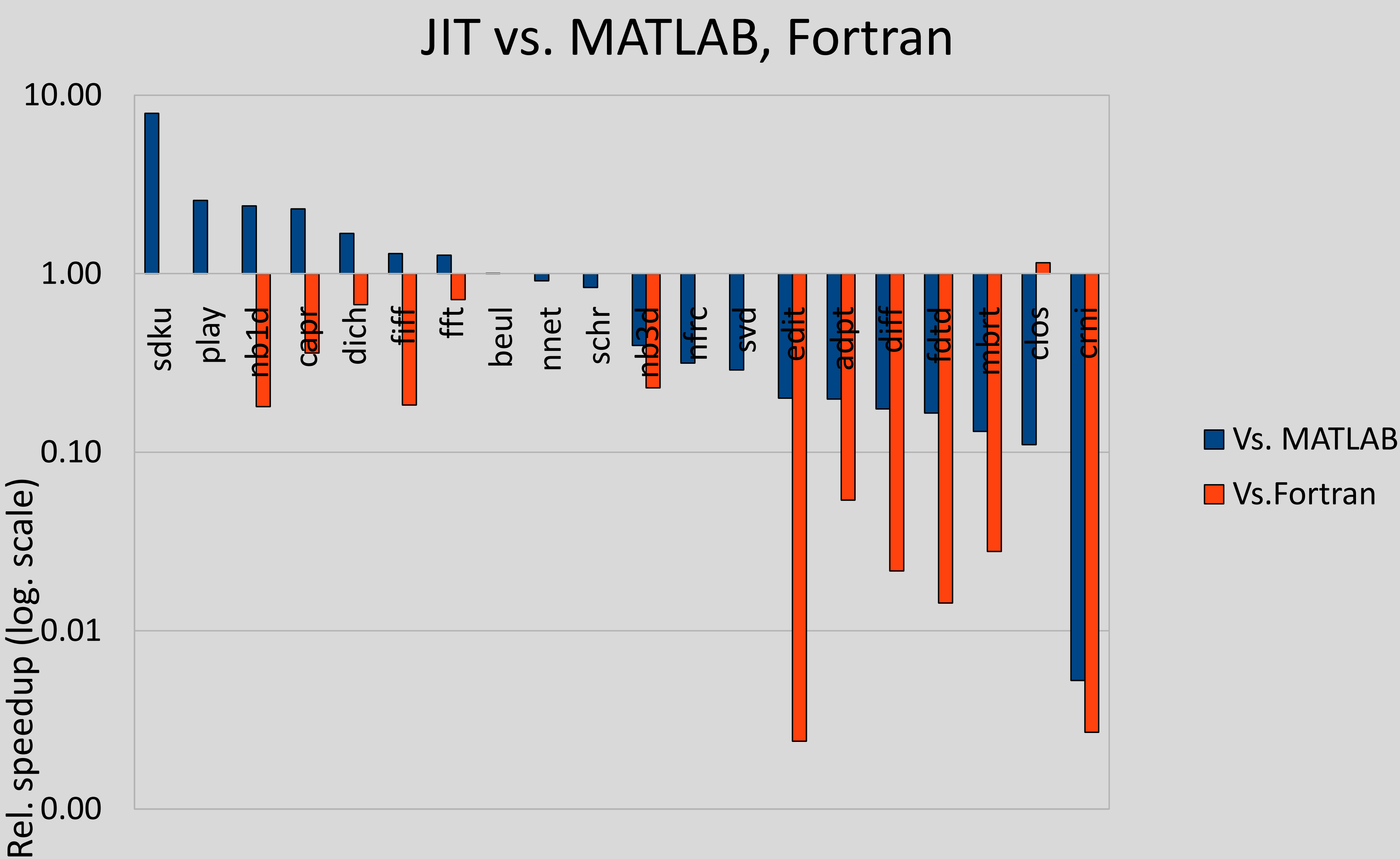
Performance Results

- Experimental setup
  - Core 2 Quad Q6600, 4GB RAM
  - Ubuntu 9.10, kernel 2.6.31, 32-bit
  - All timings averaged over 10 runs
- Comparing
  - McVM interpreter, McVM JIT w/ spec.
  - MATLAB R2009a
  - GNU Octave 3.0.5
  - McFor / GNU Fortran 4.4.1

Results (Speed-up vs Interpreters)



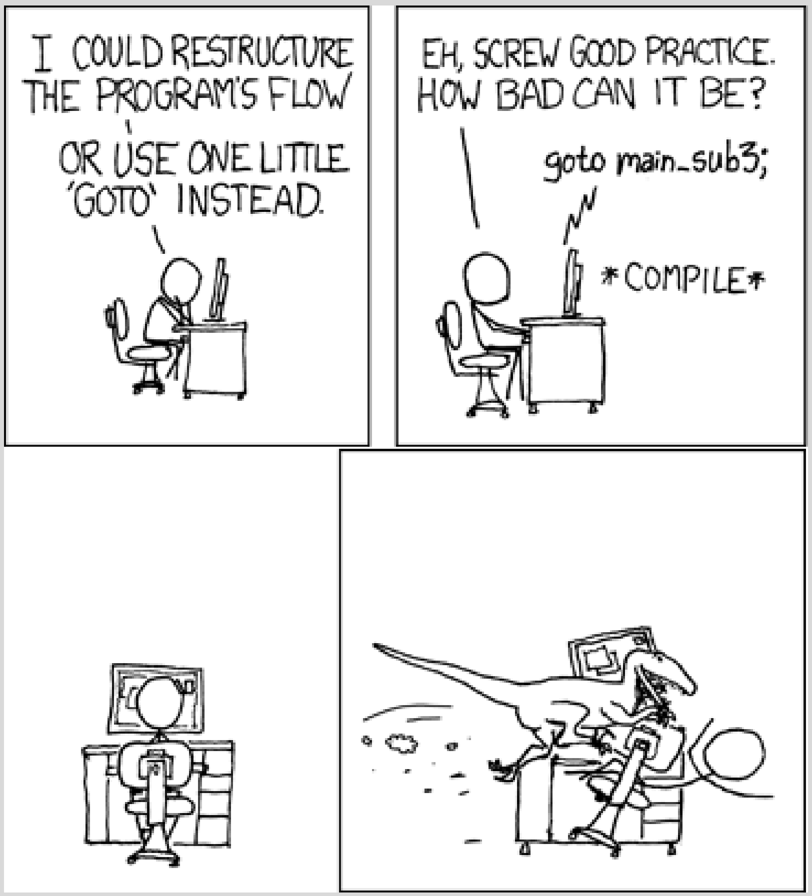
Results (Speed-up vs MATLAB, McFOR)



About Slow Benchmarks

- **`crni`** benchmark takes 1321s to execute in McVM, 6.95s in MATLAB
  - ~4x faster than Octave, but still slow
- Why?
  - Scalars known 68.7%, one of the lowest ratios
  - Unknown types propagated through entire benchmark
- Weakness of type inference system to be fixed in future work

Back-end #3: McFOR FORTRAN95 generator



- First version of McFOR based on Jun Li's M.Sc. thesis, McGill.
- Current version under development by Anton Dubrau, M.Sc. candidate, McGill.

Goals of McFOR

- Handle as large a sub-set of MATLAB as possible, while staying in the "static" setting.
- Generate code that can be effectively compiled by modern FORTRAN compilers.
- Make the generated code readable by programmers.
- Allow longer compile times and whole program analysis.
- Limit the need for type annotations.

Challenges

- Determining which identifiers are variables and which are functions.
- Finding static types which match those of FORTRAN.
- Mapping high-level MATLAB array operations to the FORTRAN95 equivalents.
- Handling reshaping implicit in MATLAB operations, including concatenation.

Handling Incompatible Types

Alpha Nodes

```
x = 0;
if  (i>0)
  S1: x = foo(i);
else
  S2: x = bar(i);
end
x = alpha(S1,S2);
y = x;
```

Eliminating Alpha Nodes

```
x = 0;
if (i>0)
  x = foo(i);
  y = x;
else
  x1 = bar(i)
  y = x1;
end
```

Value Range Propagation

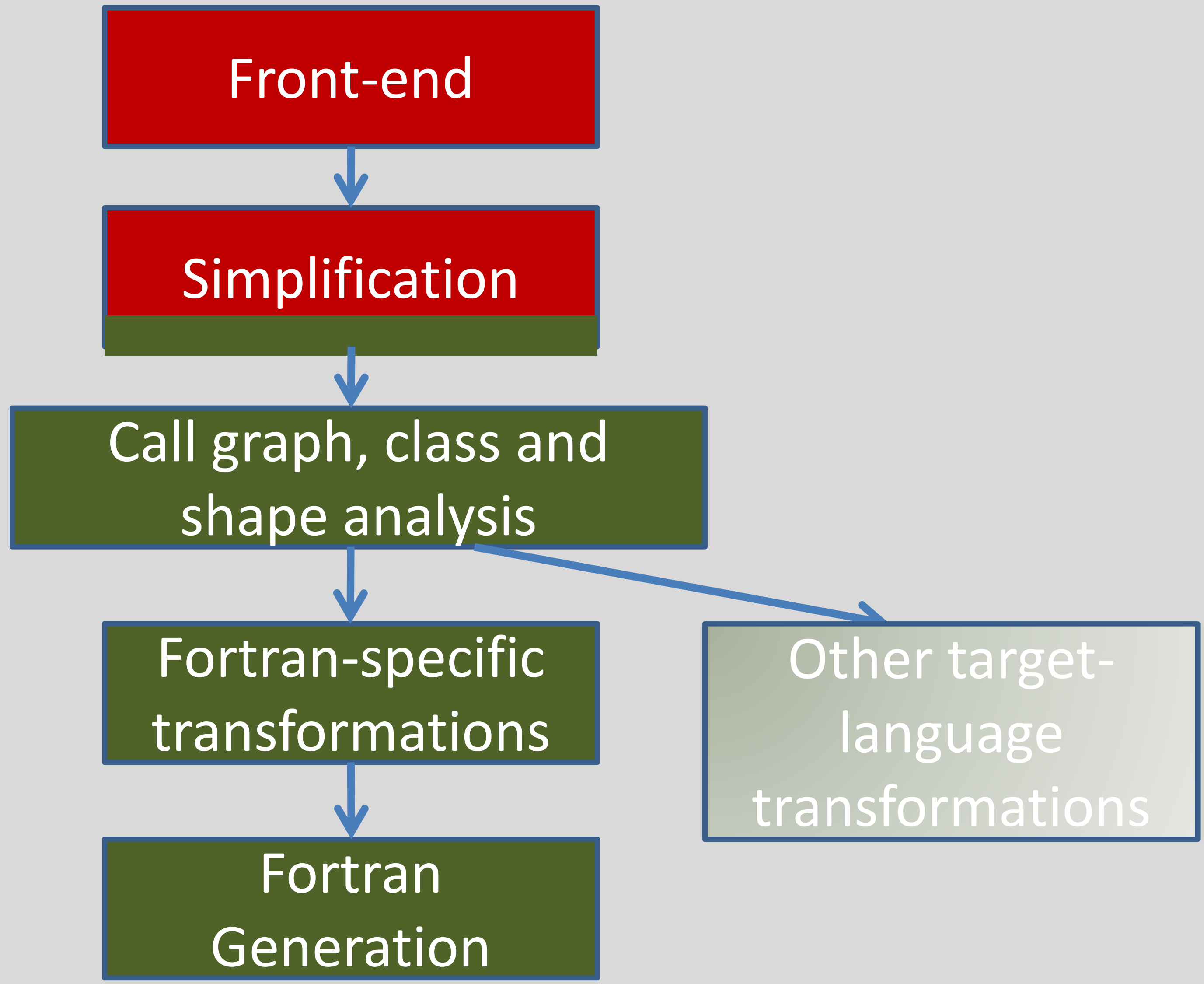
MATLAB

```
n = floor(11.5);  //n =[n,n]
for i = 1:n       // i=[1,n]
  x = 1+2*i;      // x=[3,1+2*n]



 A(x) = i;        // Size(A)=
end                     1+2*n
n = fix(n/2);     // n = [n,n]

A(n+1) = n;
```

FORTRAN

```
n = foor(10.5);
DO i = 1,n
  x = (1+(2*i));
  IF((.NOT.ALLOCATED(A))) THEN
    ALLOCATE(A((1+(2*n))));
  END IF
  A(x) = i;
END DO
n = fix(n/2);
ARRAYBOUNDSCHECKING(A,[n+1]);
A(n+1) = n;
```

Basic structure of 2nd generation McFOR



Related Work

- Procedure cloning: Cooper et al. (1992)
- MATLAB type inference: Joisha & Banerjee (2001)
  - Suggested for error detection
- MATLAB Partial Evaluator: Elphick et al. (2003)
  - Source-to-source transformation
- MaJIC: JIT compilation and offline code cache (2002)
  - Speculative compilation MATLAB to C/Fortran
- Psyco: Python VM with specialization by need (2004)
- TraceMonkey: JIT optimization of code traces (2009)

Ongoing Work

- McVM:
  - profile-guided optimization and re-optimization with on-stack replacement
  - target GPU/multi-core
- McFOR:
  - "decompile" to more programmer-friendly FORTRAN95
  - refactoring toolkit to help restructure "dynamic" features to "static" features

Conclusions

- McLAB is a toolkit to enable PL, Compiler and SE research for MATLAB
- front-end for language extensions
- analysis framework
- three back-ends including McVM and McFOR

`http://www.sable.mcgill.ca/mclab`

Intermediate Representation

EXTRA SLIDES

Internal Intermediate Representation

- A simplified form of the Abstract Syntax Tree (AST) of the original source program
- It is machine independent
- All IIR nodes are garbage collected

IIR: A Simple MATLAB Program

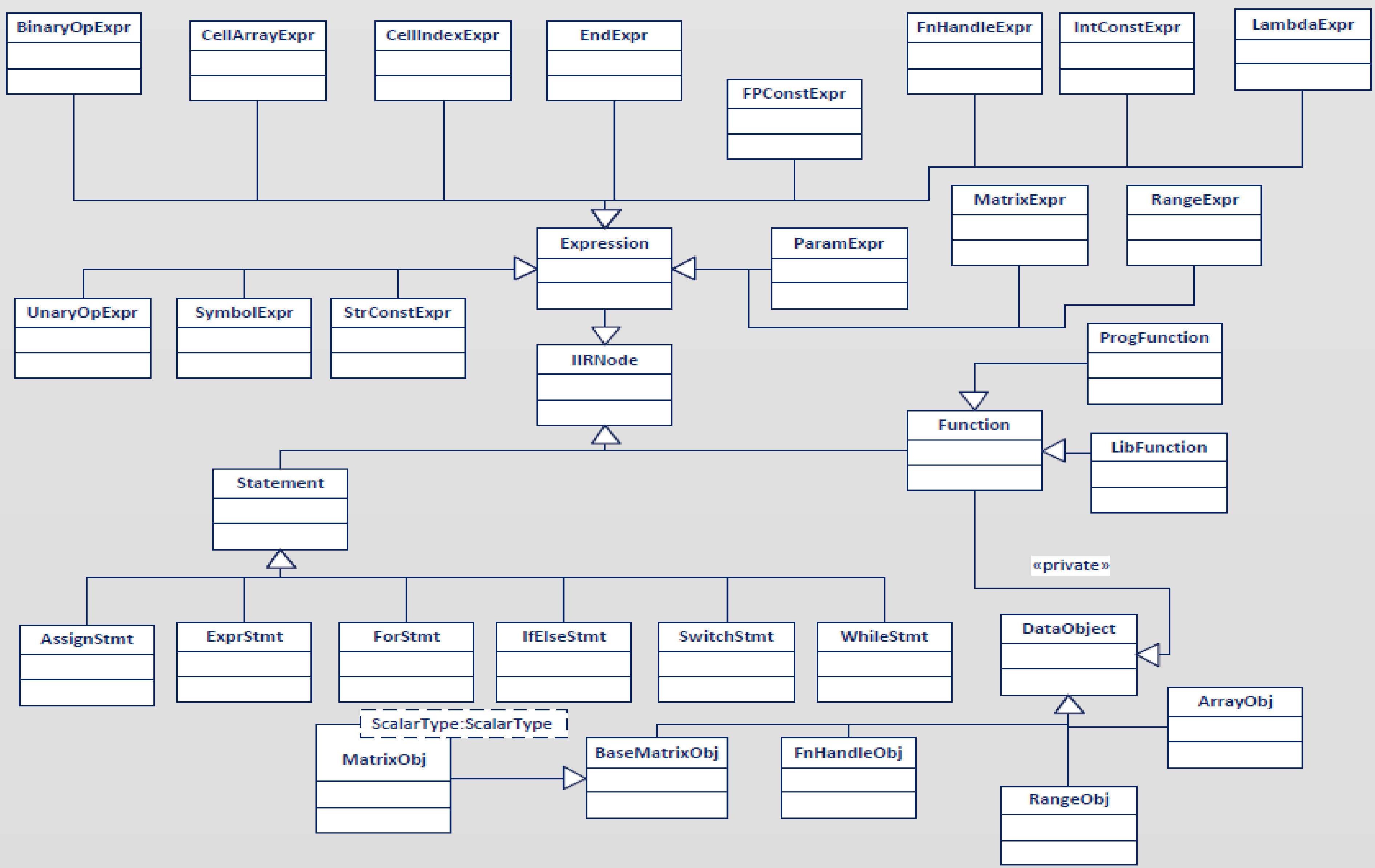
.m file

```
function a = test(n)
  a =  zeros(1,n);
  for i = 1:n
    a(i) = i*i;
  end
end
```

IIR form

```
function [a] = test(n)
   a = zeros(1, n);
   $t1 = 1; $t0 = 1;
   $t2 = $t1; $t3 = n;
   while True
      $t4 = ($t0 <= $t3);
      if ~$t4
         break;
      end
      i = $t0;
      a(i) = (i * i);
      $t0 = ($t0 + $t2);
   end
end
```

McVM Project Class Hierarchy (C++ Classes)



Supported Types

- Logical Arrays
- Character Arrays
- Double-precision floating points
- Double-precision complex number matrices
- Cell arrays
- Function Handles