Language Extensions for MATLAB

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

Overview

- How does one make language extensions for MATLAB using McLab?
- MetaLexer
- Aspects for MATLAB
- ["Types" for MATLAB]

McLab Extensible Front-end

- Scanner (MetaLexer)
- Parser (Beaver)
- AST attributes, rewrites (JastAdd)
- Attributed AST
- XML
- Other

MetaLexer

- Modular Lexer Generator
- M.Sc. thesis, Andrew Casey
- AOSD 2011
- www.sable.mcgill.ca/metalexer

Given a front-end specification for a language (i.e. MATLAB), current method to implement a front-end for an extension of that language (i.e. AspectMatlab)?

Desired Modular MetaLexer Approach
We also want to be able to combine lexical specifications for diverse languages.

- Java + HTML
- Java + Aspects (AspectJ)
- Java + SQL
- MATLAB + Aspects (AspectMatlab)

Would like to be able to reuse and extend lexical specification modules

- Nested C-style comments
- Javadoc comments
- Floating-point constants
- URL
- regular expressions
- ...

First, let’s understand the traditional lexer tools (lex, flex, jflex).

- programmer specifies regular expressions + actions
- tools generate a finite automaton-based implementation
- states are used to handle different language contexts

JFlex Lexing Structure

Specification in one file.

Leverhulme Lecture #2
Current (ugly) method for extending jflex specifications - copy & modify

- Copy jflex specification.
- Insert new scanner rules into copy.
  - Order of rules matters!
- Introduce new states and action logic for converting between states.

- Principled way of weaving new rules into existing rules.
- Modular and abstract notion of state and changing between states.

Example Structure of a MetaLexer Specification for MATLAB

Extending a MetaLexer Specification for Matlab

Sharing component specifications with MetaLexer

Scanning a properties file

```plaintext
#some properties
name=properties
date=2009/09/21

#some more properties
owner=root
```
Key problems to solve:

- How to implement the meta-token lexer?

- How to allow for insertion of new components, replacing of components, adding new embeddings (metalexer transitions).

- How to insert new patterns into components at specific points.
Implementing the meta-token lexer

1. Recognize a meta-pattern, i.e. when to go to a new component and when to return.

2. Recognize the matching suffix.

Implementing inheritance (structured weaving).

Implementing MetaLexer layout inheritance

- Layouts can inherit other layouts
- `%inherit` directive put at the location at which the inherited transition rules (embeddings) should be placed.
- Each `%inherit` directive can be followed by:
  - `%unoption`
  - `%replace`
  - `%unembed`
  - New embeddings

Implementing MetaLexer component inheritance

Weaving in an inherited component

New Component adds some rules and inherits original component.

Woven output

Results:
Applied to three projects with complex scanners:

- AspectJ (abc and extensions)
- Matlab (Annotations and AspectMatlab extensions)
- MetaLexer
**MetaLexer scanner implemented in MetaLexer**

- 1st version of MetaLexer written in JFlex, one for components and one for layouts.
- 2nd version implemented in MetaLexer, many shared components between the component lexer and the layout lexer.

**Related Work for MetaLexer**

- Ad-hoc systems with separate scanner/ LALR parser
  - Polyglot
  - JastAdd
  - abc
- Recursive-descent scanner/parser
  - ANTLR and systems using ANTLR
- Scannerless systems
  - Ratsl (PEGs)
- Integrated systems
  - Copper (modified LALR parser which communicates with DFA-based scanner)

**Metalexer Conclusions**

- MetaLexer allows one to specify modular and extensible scanners suitable for any system that works with JFlex.
- Two main ideas: meta-lexing and component/layout inheritance.
- Used in large projects such as abc, McLab and MetaLexer itself.
- Available at: www.sable.mcgill.ca/metalexer

**AspectMatlab**

- Simple Aspect-Oriented extension to MATLAB
- M.Sc. thesis, Toheed Aslam
- Analysis by Jesse Doherty, applications by Anton Dubrau, extensions by Olivier Savary-Belanger
- AOSD 2010
- www.sable.mcgill.ca/mclab

**Why AspectMatlab?**

- Test the McLab framework for extensibility
- Bring a simple and relevant version of AOP to scientists.
- simple language constructs
- focus on arrays and loops
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

![Profiling Array Sparsity Example]

- 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
        data
count = 0;
    end
    helper functions
        function x=getCount(this)
            x = this.count;
        end
    end
end

Aspect Definition

aspect myAspect
    properties
        data
        count = 0;
    end
    helper functions
        function x=getCount(this)
            x = this.count;
        end
    end
    patterns
        pointcuts
            foci:call(foo);
        actions
            foocounter : before foci;
            this.count = this.count + 1;
        end
    end
end

Function and Operator Patterns

patterns
    pCallFoo : call(foo);
    pExecBar : execution(bar);
    pCallForLargs : call(foo("",""));
    pfExecMain : mainexecution();
    end
patterns
    plusOp : op('+');
    timesOp : op('*') || op('**');
    matrixOp : op(matrix);
    allButMinus : op(all) & ~op('-');
    end

Array Patterns

a(i) = b(j,k)

Context Info
name
object (value)
line number
location
blob
file name

a(i) = b(j,k)

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

\[
\text{\texttt{pLoopI : loop(i);}} \\
\text{\texttt{pLoopHeadI : loophead(i);}} \\
\text{\texttt{pLoopBodyI : loopbody(i);}} \\
\text{\texttt{end}}
\]

```
T = [1,3,5,7,...,n];
for i = 1:numel(T)
    ...
end
```

Scope Patterns

```
pWithinFoo : within(function, foo);
pWithinBar : within(script, bar);
pWithinMyClass : within(class, MyClass);
pWithinLoops : within(loops, 'i');
pWithinAllAbc : within('a', abc);
end
```

Compound Patterns

- Logical combinations of primitive patterns

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

Before & After Actions

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

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

Context Exposure

```
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

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

actions
actcall : around pCallFoo : (args)
    % proceed not called, so varargout is set
    varargout{1} = bar(args{1}, args{2});
end
```
Actions Weaving Order

```plaintext
actions
    before1 : before pCallFoo
    around1 : around pCallFoo
    after1 : after pCallFoo
    before2 : before pCallFoo
    around2 : around pCallFoo
    after2 : after pCallFoo
end
```

Compiler Structure

- Front-end
- Separator
- Matlab AST
- Matlab AST of Aspects
- Transformations
- Matlab AST of Aspects
- AspectInfo
- AspectInfo
- Matlab AST
- Matlab AST of Aspects
- Post-processing
- Woven AST
- Matches & Weaver
- Woven AST
- Resolved Name Set
- Resolved Name Set
- Name Resolution Analysis
- Name Resolution Analysis
- Simplified AST
- Simplified AST

Name Resolution Analysis

```plaintext
patterns
pCallFoo : call(foo);
pGetFoo : get(foo);
end
```

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

Related Work for AspectMatlab

- AspectJ (Kiczales et al., ECOOP '01)
  - abc (The de Moor and Hendren gang, AOSD '05)
  - Array pointcuts (Chen et al., JSES '07)
  - Loop pointcuts (Mandel et al., AOSD '06)
- AspectCobol (Lamme et al., AOSD '05)
- Domain-Specific Aspects in Matlab (Cardoso et al., DSAL workshop held at AOSD '10)
Conclusions

- McLab supports extensions to MATLAB
- We developed MetaLexer to support modular and extensible lexers, and then used it in McLab.
- We designed and implemented AspectMatlab as an exercise in using McLab for extensions, and also to provide simple and relevant AOP for scientists.

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
```

```
>> Ex1(3)
an = 0.8415 0.9093 0.1411
```

```
>> Ex1(2.3)
an = 0.8415 0.9093
```

```
>> Ex1(int32(3))
??? Undefined function or method 'sin' for input arguments of type 'int32'.
Error in ==> Ex1 at 5
r(i) = sin(i);
```

```
>> Ex1('c')
??? For colon operator with char operands, first and last operands must be char.
Error in ==> Ex1 at 4
for i=1:n
```

```
>> Ex1(@sin)
??? Undefined function or method '_colonobj' for input arguments of type 'function_handle'.
Error in ==> Ex1 at 4
for i=1:n
```

```
>> Ex1(complex(1,2))
Warning: Colon operands must be real scalars.
> In Ex1 at 4
ans = 0.8415
```

```
>> Ex1(true)
Warning: Colon operands should not be logical.
> In Ex1 at 4
ans = 0.8415
```

```
>> Ex1([3,4,5])
an = 0.8415 0.9093 0.1411
```

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).
```
```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');
end
```

```
>> Ex1(3)
an = 0.8415 0.9093 0.1411
```

```
>> Ex1('c')
Type error in Ex1.m, Line 4: Expecting 'n' to have
'type "scalar of Float"', but got the type
'scalar of char'.
```

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

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

High-level types in MATLAB

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

Simple Example

- numeric
- complex
- float
- int
- signed
- double
- unsigned

Capturing reflective information

- a.type
- a.value
- a.dims
- a.basetype
function \( r \) = foo( a, b )
\[ \text{atype('a','array[<n>,<m>] of real');} \]
\[ \text{atype('b','array[a.m,<p>] of a.basetype');} \]
\[ \% \ldots \]
\[ \% \text{ body of foo } \]
\[ \% \ldots \]
\[ \text{atype('r','array[a.m,b.p] of a.basetype');} \]
\[ \text{end} \]

* \( <n> \) can be used as a dimension spec
* value of \( n \) is instantiated from the runtime dimension
* repeated use in same atype statement implies equality