Creating a Robust AspectMATLAB Compiler

Hongji Chen

UCORE'16

Aspect Oriented Programming in Java (AspectJ)

Pointcuts

A pointcut is a program element that picks out join points and exposes data from the execution context of those join points. [6]

```
pointcut pCall() : call (* *.*(..))          // all function calls
pointcut pExec() : execution (* *.*(..)) // all function execs
pointcut pComp() : pCall() || pExec()       // both call and execs
```

Pointcut Advice

Advice defines crosscutting behavior. It is defined in terms of pointcuts. The code of a piece of advice runs at every join point picked out by its pointcut. [7]

```
before() : pComp() {
    /* do something */
}
```

Aspect Oriented Programming in Java (AspectJ)

Aspect File

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class DemoAspect{
    @Pointcut("call(* DemoTarget.*(..))")
    public void pCall() {}

    @Pointcut("execution(* DemoTarget.*(..))")
    public void pExec() {}

    @Pointcut("pCall() || pExec()")
    public void pComp() {}

    @Before("pComp()")
    public void pAdvice(JoinPoint joinPoint) {
        System.out.println(String.format(
            "[line:%3d] %s",
            joinPoint.getSourceLocation().getLine(),
            joinPoint.toLongString()
        ));
    }
}
```

Target File

```
public class DemoTarget {
    public static void foo() {}

    public static void goo() {hoo();}

    public static void hoo() {}

    public static void main(String[] args) {
        foo();
        goo();
    }
}
```

Aspect Oriented Programming in Java (AspectJ)

Aspect File

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class DemoAspect{
    @Pointcut("call(* DemoTarget.*(..))")
    public void pCall() {}

    @Pointcut("execution(* DemoTarget.*(..))")
    public void pExec() {}

    @Pointcut("pCall() || pExec()")
    public void pComp() {}

    @Before("pComp()")
    public void pAdvice(JoinPoint joinPoint) {
        System.out.println(String.format(
            "[line:%3d] %s",
            joinPoint.getSourceLocation().getLine(),
            joinPoint.toLongString()
        ));
    }
}
```

Target File

```
public class DemoTarget {
    public static void foo() {}

    public static void goo() {hoo();}

    public static void hoo() {}

    public static void main(String[] args) {
        foo();
        goo();
    }
}
```

```
[line:  8] execution(public static void DemoTarget.main(java.lang.String[]))
[line:  9] call(public static void DemoTarget.foo())
[line:  2] execution(public static void DemoTarget.foo())
[line: 10] call(public static void DemoTarget.goo())
[line:  4] execution(public static void DemoTarget.goo())
[line:  4] call(public static void DemoTarget.hoo())
[line:  6] execution(public static void DemoTarget.hoo())
```

Aspect MATLAB Programming in MATLAB

Previous Work

- Toheed Aslam's AspectMATLAB Compiler [5]
- Andrew Bodzay's AspcetMATLAB++ Compiler [1]
- McSAF Framework [2], and Kind Analysis [3]
- Samuel Suffos's Parser [4]

Aspect MATLAB Programming in MATLAB

- Pointcuts $\Rightarrow$ Patterns
- Pointcut Advice $\Rightarrow$ Actions

Aspect MATLAB Programming in MATLAB

Patterns and Actions

Patterns

Annotate	annotation comment
Call	subroutine call
Execution	subroutine execution
Get	variable read
Loop	for-loop or while-loop
LoopHead	loop initialization
LoopBody	loop execution body

MainExecution	entry point
Operator	MATLAB matrix/array operation
Set	variable write
Dimension	restrict search by shape
IsType	restrict search by type
Within	restrict search by scope

Actions

Before	Action will be executed before the captured join point
After	Action will be executed after the captured join point
Around	Action will be executed instead of the captured join point

Aspect MATLAB Programming in MATLAB

Improvements and Contributions

- Use of More Robust Front-end
- Clear Grammar
- Semantic Validation
- More Robust Transforming Strategy
- Extended Argument Matching

Improvement on Grammar

Extended Get and Set Pattern more clear pattern

```
get(goo) & dimension([1,1]) & istype(logical)
set(*) & dimension([3,3,..]) & istype(double)
```

$\Rightarrow$

```
get(goo : [1,1]logical)
set(* : [3,3,..]double)
```

Enhanced Usage of Wildcards allow wildcard to appear at any position within argument list

```
dimension([3,..,4])
```

Extended Call and Execution Pattern restrict matching according to shape and type of arguments and return values.

```
call(foo([1,1]function_handle, *, [..]double) : [1,3]logical)
```

Semantic Validation

Semantic Invalid Pattern and Action may lead to unpredictable result during AST transformation and weaving.

```
% unbounded 'within pattern'
a1 : before within(function, foo) : ()
% annotation has no type nor shape
a2 : before annotate(*(..)) & istype(logical) & dimension([3,3])
% or operator RHS unbounded
a3 : after loop(for : i) || within(script, demo.m) : ()
% cannot predict type and shape of return values without invokin
a4 : before call(foo(..) : [1,1]logical) : ()
% same as above, but more complicate
a5 : before (get(*) & call(*(..))) & istype(integer) : ()
% unclear pattern apply not operation to get pattern
a6 : before ~get(foo : double) : ()
```

Semantic Validation

- Basic Pattern Classification (primitive pattern and modifier pattern)
- Pattern Type Analysis
- Logical Reduction (resolve modifier pattern on primitive pattern)
- Modifier Validation and Weaving Method Validation

Semantic Validation

Basic Pattern Classification

Primitive Pattern

Primitive patterns are patterns with actual joint point within MATLAB source code.

- Get/Set Pattern
- Call/Execution Pattern
- Loop/LoopHead/LoopBody Pattern
- MainExecution Pattern
- Annotation Pattern
- Operator Pattern

Modifier Pattern

Unlike primitive patterns, modifier patterns do not provide joint points for action weaving, instead, it pose restriction on the primitive pattern which they bound to.

- Dimension Pattern
- IsType Pattern
- Scope Pattern

Semantic Validation

Pattern Type Analysis

And Compound Pattern

(Primitive $\wedge$ Primitive) $\rightarrow$ Primitive

(Primitive $\wedge$ Modifier) $\rightarrow$ Primitive

The Modifier pattern is bounded towards the primitive pattern and remove from further analysis.

(Modifier $\wedge$ Primitive) $\rightarrow$ Primitive

(Modifier $\wedge$ Modifier) $\rightarrow$ Modifier

Not Compound Pattern

$\neg$ Primitive $\rightarrow$ Invalid

Ambiguous Pattern

$\neg$ Modifier $\rightarrow$ Modifier

Or Compound Pattern

(Primitive $\vee$ Primitive) $\rightarrow$ Primitive

(Primitive $\vee$ Modifier) $\rightarrow$ Invalid

Cannot resolve right hand side modifier pattern to a primitive pattern, thus the modifier pattern is unbounded.

(Modifier $\vee$ Primitive) $\rightarrow$ Invalid

(Modifier $\vee$ Modifier) $\rightarrow$ Modifier

Semantic Validation

Pattern Simplification

Simplify pattern using logical equivalence, moving modifier pattern towards its bounded primitive pattern.

```
(get(*) & call(*(..))) & ~(istype(integer) & istype(logical))
```

$\neg(A \wedge B) \equiv \neg A \vee \neg B$

$\Rightarrow$

```
(get(*) & call(*(..))) & (~istype(integer) | ~istype(logical))
```

$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$

$\Rightarrow$

```
(get(*) & (~istype(integer) | ~istype(logical))) &
    (call(*(..)) & (~istype(integer) | ~istype(logical)))
```

Semantic Validation

Modifier Validation and Weaving Method Validation

Pattern	Dimension			IsType			Scope
	Before	After	Around	Before	After	Around	
Get/Set	✓	✓	✓	✓	✓	✓	✓
Call/Exec	×	✓	×	×	✓	×	✓
Annotation	×	×	×	×	×	×	✓
Loops	×	×	×	×	×	×	✓
MainExec	×	×	×	×	×	×	×
Operator	×	✓	×	×	✓	×	✓

```
a5 : before (get(*) & call(*(..))) & istype(integer) : ()
```

$\Rightarrow$

```
a5 : before (get(*) & (~istype(integer) | ~istype(logical))) &
            (call(*(..)) & (~istype(integer) | ~istype(logical))
```

$\Rightarrow$ **Reject**

More Robust Transforming Strategy

- Comma Separated List Handling
- End Expression Resolve
- Set Pattern New Variable Capture
- ...

More Robust Transforming Strategy

Comma Separated List Handling

```
foo(c{:}, m(:).f, p);
```

with pattern

```
get(*) | call(*(..))
```

Current AspectMATLAB

```
AM_VAR_1 = c{:};
AM_VAR_2 = m(:).f;
AM_VAR_3 = p;
AM_VAR_4 = foo(AM_VAR_1, AM_VAR_2, AM_VAR_3);
```

$\Rightarrow$

```
AM_VAR_1 = {c{:}};
AM_VAR_2 = {m(:).f};
AM_VAR_3 = p;
AM_VAR_4 = foo(AM_VAR_1{:}, AM_VAR_2{:}, AM_VAR_3);
```

More Robust Transforming Strategy

End Expression Resolve

```
m(p1, end, c{:}, :)
```

with pattern

```
get(*)
```

Current AspectMATLAB

Ignored

$\Rightarrow$

```
AM_VAR_1 = p1;
AM_VAR_2 = {c{1:builtin('end', c, 1, 1)}};
AM_VAR_3 = sum([1, 1, length(AM_VAR_2), 1]);
AM_VAR_4 = m(
            AM_VAR_1,
            builtin('end', m, sum([1, 1]), AM_VAR_3),
            AM_VAR_3,
            1:builtin('end', m,
                      sum([1, 1, length(AM_VAR_2), 1]),
                      AM_VAR_3)
           );
```

More Robust Transforming Strategy

Set Pattern New Variable Capture

```
[var1, c{1:2}] = foo();
```

with pattern

```
set(*)
```

Current AspectMATLAB

```
[var1, c{1:2}] = foo();                % both new value as foo()
```

⇒

```
if exist('var1', 'var')
    AM_VAR_1 = var1;
end
if exist('c', 'var')
    AM_VAR_2 = c;
end
[AM_VAR_1, AM_VAR_2{1:2}] = foo();
[var1] = deal(AM_VAR_1);                 % new value as AM_VAR_1
[c{1:2}] = deal(AM_VAR_2{1:2});          % new value as AM_VAR_2
```

Extended Argument Matching

Current AspectMATLAB

```
call(foo(*,..))
```

- Matching according to the number of input arguments
- dots wildcard only allow to appear at the end of the pattern list

Extended Argument Matching

Current AspectMATLAB

```
call(foo(*,..))
```

- Matching according to the number of input arguments
- dots wildcard only allow to appear at the end of the pattern list

```
call(foo(*, .., [1,*,..,2,3]double) : [..]logical, [2,2]*)
```

- Matching according to both input and output (restriction applied)
- Dots wildcards can appear at any part in the signature list
- Shape and Type information matching

Extended Argument Matching

- Collect Alphabet
- Building nondeterministic finite automaton for shape patterns
- Convert nondeterministic finite automaton into deterministic finite automaton
- Emit matcher function

Extended Argument Matching

Collect Alphabet

Why? We need a finite alphabet to built NFA and DFA. But, in MATLAB, any valid identifier can be a valid type name, and dimension can be a list of any positive integers.
Trivial solution. $\times$

Extended Argument Matching

Collect Alphabet

Why? We need a finite alphabet to built NFA and DFA. But, in MATLAB, any valid identifier can be a valid type name, and dimension can be a list of any positive integers.
Trivial solution. $\times$
Construct alphabet Σ as follow:

- $\epsilon \in \Sigma$
- if symbol s appears in signature, then $s \in \Sigma$
- let σ be a special symbol, denoting any other symbol that don't appear in the signature

Then the alphabet Σ is a finite set, as we have a finite signature.

Extended Argument Matching

Collect Alphabet

Why? We need a finite alphabet to built NFA and DFA. But, in MATLAB, any valid identifier can be a valid type name, and dimension can be a list of any positive integers.
Trivial solution. $\times$
Construct alphabet Σ as follow:

- $\epsilon \in \Sigma$
- if symbol s appears in signature, then $s \in \Sigma$
- let σ be a special symbol, denoting any other symbol that don't appear in the signature

Then the alphabet Σ is a finite set, as we have a finite signature.

```
call(foo(*, .., [1,*,..,2,3]double) : [..]logical, [2,2]*)
```

$\Sigma_{shape} = \{\epsilon_{shape}, 1, 2, 3, \sigma_{shape}\}$
$\Sigma_{type} = \{\epsilon_{type}, \text{double}, \text{logical}, \sigma_{type}\}$

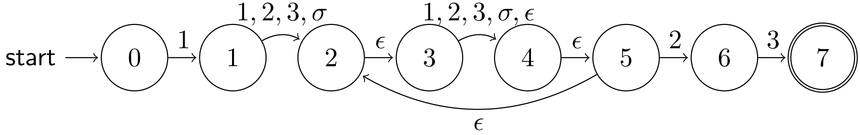
Extended Argument Matching

Building nondeterministic finite automaton for shape patterns

Pattern with only shape matching

```
dimension([1, *, .., 2, 3])
```

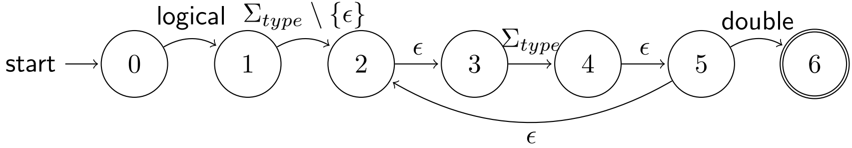
Alphabet : $\Sigma_{shape} = \{\epsilon, 1, 2, 3, \sigma\}$



Pattern with only type matching

```
call(foo(logical, *, .., double))
```

Alphabet : $\Sigma_{type} = \{\epsilon, \text{logical}, \text{double}, \sigma\}$



Extended Argument Matching

Convert NFAs into DFAs

Using subset construction method

```
dimension([1, *, .., 2, 3])
```

Alphabet Map :
$\{1 = 1, 2 = 2, 3 = 3, \sigma = 4\}$
Matrix representation of DFA in MATLAB:

```
AM_FUNC_8 = [2, 6, 6, 6;
             3, 3, 3, 3;
             3, 4, 3, 3;
             3, 4, 5, 3;
             3, 4, 3, 3;
             6, 6, 6, 6];
```

```
call(foo(logical, *, .., double))
```

Alphabet Map :
$\{\text{logical} = 1, \text{double} = 2, \sigma = 3\}$
Matrix representation of DFA in MATLAB:

```
AM_FUNC_4 = [2, 5, 5;
             3, 3, 3;
             3, 4, 3;
             3, 4, 3;
             5, 5, 5];
```

Extended Argument Matching

Emit matcher function

```
call(foo(logical, *, .., double))
```

$\Rightarrow$

```
function [AM_FUNC_3] = AM_VAR_1(AM_FUNC_2)
  AM_FUNC_4 = [2, 5, 5; 3, 3, 3; 3, 4, 3; 3, 4, 3; 5, 5, 5];
  AM_FUNC_5 = 1;
  for AM_FUNC_6 = (1 : length(AM_FUNC_2))
    AM_FUNC_5 = AM_FUNC_4(AM_FUNC_5, AM_VAR_0(AM_FUNC_2{AM_FUNC_6}));
  end
  AM_FUNC_7 = [4];
  for AM_FUNC_8 = (1 : length(AM_FUNC_7))
    if (AM_FUNC_5 == AM_FUNC_7(AM_FUNC_8))
      AM_FUNC_3 = true;
      return
    end
  end
  AM_FUNC_3 = false;
  return
  function [AM_FUNC_1] = AM_VAR_0(AM_FUNC_0)
    if isa(AM_FUNC_0, 'double')
      AM_FUNC_1 = 2;
    elseif isa(AM_FUNC_0, 'logical')
      AM_FUNC_1 = 1;
    else
      AM_FUNC_1 = 3;
    end
  end
end
```

Extended Argument Matching

More complicate pattern

```
call(foo([1, 2, .., 3]logical, *, .., [1, .., 2, 3]logical))
pattern1 = [1, 2, .., 3]logical
pattern2 = [1, .., 2, 3]logical
```

Previous solution won't work, as alphabet is not a surjective map from variables to symbol code.

Alternative solution: let $S_{pattern1}$ denotes the set of variable matching to pattern1, $S_{pattern2}$ denotes the set of variable matching to pattern2, and S denotes the set of all possible input variables. Then alphabet $A = \{\epsilon, s_1, s_2, s_3, \sigma\}$, with following map $\tau : S \to A$ is a suitable candidate for NFA/DFA construction, and $A \supseteq \operatorname{Im} \tau$ is a finite set with at most $2^{|\sharp\text{patterns}|} + 1$ symbols.

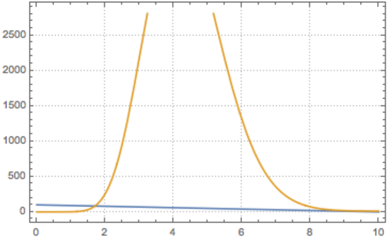
$$\tau(x) = \begin{cases} \epsilon & \epsilon\text{-transition} \\ s_1 & x \in S_{pattern1} \setminus S_{pattern2} \\ s_2 & x \in S_{pattern2} \setminus S_{pattern1} \\ s_3 & x \in S_{pattern1} \cap S_{pattern2} \\ \sigma & x \in (S_{pattern1} \cup S_{pattern2})^{\mathsf{c}} \end{cases}$$

Extended Argument Matching

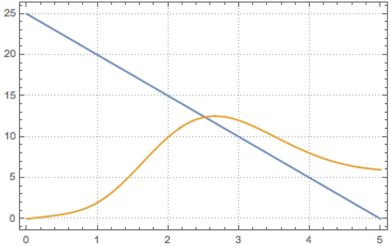
Analysis on performance

If a pattern has m signature with k dots wildcards.
The modified NFA/DFA method use $m * (m - k)$ times shape and type checking.
The for-loop based method use $k^{m-k} + k$ times shape and type checking.

Plot under $m = 10$,



Plot under $m = 5$,



$\Rightarrow$

Implemented as for-loop based method for patterns with few dots wildcard (1-2), NFA/DFA method for other scenario.

Extended Argument Matching

Using matcher function

Input argument matching:

```
foo(var1, var2(:).f, var3{:})
```

⇒

```
AM_VAR_1 = {var1, var2(:), var{:}};
AM_MATCH_RESULT(1) = matcher1(AM_VAR_1);
AM_MATCH_RESULT(2) = matcher2(AM_VAR_2);
% ...
foo(AM_VAR_1{:})
```

Output argument matching:

```
foo(var1, var2(:).f, var3{:})
```

⇒

```
% callWithMatcher is a MEX implemented subroutine using C
AM_VAR_1 = {var1, var2(:), var{:}};
[AM_MATCH_RESULT, ...] = callWithMatcher(
                              @foo, AM_VAR_1,
                              @matcher1, @matcher2, ...
                          );
```

Laurie Hendren Andrew Bodzay.
Aspectmatlab++: Annotations, types, and aspects for scientists.
MODULARITY'15, 2015.

Jesse Doherty.
Mcsaf: An extensible static analysis framework for the matlab language.
Master's thesis, McGill University, 2011.

Laurie Hendren Jesse Doherty and Soroush Radpour.
Kind analysis for matlab.
OOPSLA11, 2011.

Samuel Suffos.
Mclab-parser.
URL: `https://github.com/Sable/mclab-parser`.

Anton Dubrau Toheed Aslam, Jesse Doherty and Laurie Hendren.
Aspectmatlab: An aspect-oriented scientific programming language.
AOSD'10, 2010.

Palo Alto Research Center Xerox Corporation.
The aspectj programming guide, 2003.

URL: https://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html.

Palo Alto Research Center Xerox Corporation.
The aspectj programming guide, 2003.
URL: https://eclipse.org/aspectj/doc/next/progguide/semantics-advice.html.