Creating a Robust AspectMATLAB Compiler

Hongji Chen Laurie Hendren (advisor)

UCORE'16

Aspect oriented programming

- Pointcut $\Rightarrow$ pattern
- Pointcut advice $\Rightarrow$ action

Generating Dynamic Call Graph Using AspectMATLAB

A quick guide to AspectMATLAB

- Pointcuts(patterns)

```
patternCall : call(*(..))
patternExecution : execution(*(..))
```

Generating Dynamic Call Graph Using AspectMATLAB

A quick guide to AspectMATLAB

- Pointcuts(patterns)

```
patternCall : call(*(..))
patternExecution : execution(*(..))
```

- Pointcut advices(actions)

```
actionCallBefore : before call(*(..)) : (name)
  disp(sprintf('entering %s', name));
end
actionExecution : around execution(*(..)) : (name)
  ticHandle = tic;
  proceed();
  elapsedTime = toc(ticHandle);
  disp(sprintf('executed %s, Elapsed time: %d', name, elapsed
end
actionCallAfter : after call(*(..)) : (name)
  disp(sprintf('returning from %s', name));
end
```

Generating Dynamic Call Graph Using AspectMATLAB

A quick guide to AspectMATLAB

```
function [] = launchingFunc()
  % join point here
  % entrance point
  foo();     % join point here
end

function [] = foo()
  % join point here
  goo();     % join point here
  % do something here
  goo();     % join point here
end

function [] = goo()
  % join point here
  % do something here
end
```

Generating Dynamic Call Graph Using AspectMATLAB

A quick guide to AspectMATLAB

```
function [] = launchingFunc()
  % join point here
  % entrance point
  foo();     % join point here
end

function [] = foo()
  % join point here
  goo();     % join point here
  % do something here
  goo();     % join point here
end

function [] = goo()
  % join point here
  % do something here
end
```

Executing Result

```
entering foo
entering goo
executed goo, Elapsed time: <dou
returning from goo
entering goo
executed goo, Elapsed time: <dou
returning from goo
executed foo, Elapsed time: <dou
returning from foo
executed launchingFunc, Elapsed
```

Previous Work

- Toheed Aslam's AspectMATLAB Compiler
- Andrew Bodzay's AspcetMATLAB++ Compiler
- McSAF Framework, and Kind Analysis
- Samuel Suffos's Parser

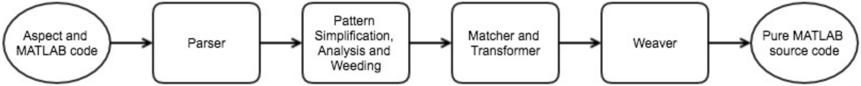
Improvement and Contributions

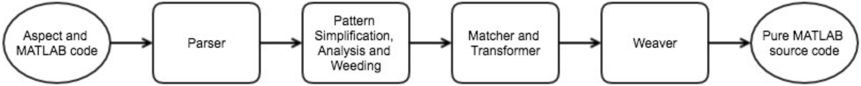
- Use of More Robust Front-end
- Clear Grammar
- Semantic Validation
- More Robust Transforming Strategy
- Extended Argument Matching

Compiler Structure

Existing AspectMATLAB Compiler Structure

New AspectMATLAB Compiler Structure





Parser

- New parser built using ANTLR3
- Extend set/get/call pattern to make patterns more clear

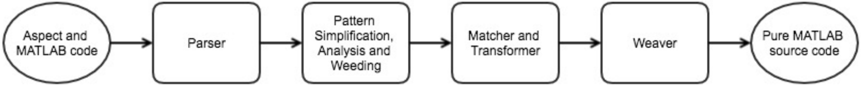
  ```
  get(x) & dimension([1, 1]) & istype(logical)
  ```

  $\Rightarrow$

  ```
  get(x : [1,1]logical)
  ```

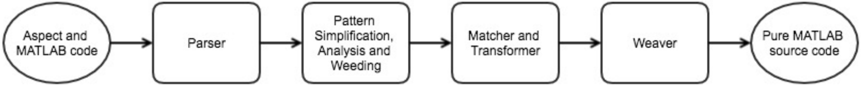
- allow dots wildcard to appear any where in the signature list

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



Semantic Validation

- Basic Pattern classification
  - Primitive pattern
  - Modifier pattern
- Pattern type analysis for compound pattern
- using logical equivalence to associate modifier pattern to primitive pattern
- inspect primitive pattern individually.



Semantic Validation

Examples

matching under "before" case

```
(get(x) | call(foo(..))) & ~(istype(logical) | dimension([1,1]))
```

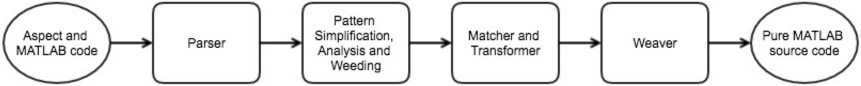
⇒

```
(get(x) | call(foo(..))) & (~istype(logcial) & ~dimension([1,1])
```

⇒

```
(get(x) & (~istype(logcial) & ~dimension([1,1]))) |
    (call(foo(..)) & (~istype(logcial) & ~dimension([1,1])))
```

⇒
Reject



Matcher and Transformer

- handling spanned comma separated list, e.g.

```
c{:}    m(:).f    c{1:5}
```

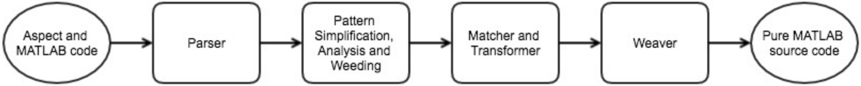
- correctly resolve end expression

```
m(1, end - 1)
```

- better variable capture for set pattern

```
[var1, var2] = foo();
```

- ...



Weaver

Extended Argument Signature List Matching

Current AspectMATLAB Compiler

- subroutine name
- number of subroutine input arguments

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

New AspectMATLAB Compiler

- shape information of the argument
- type information of the argument
- subroutine outputs

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

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]*)
```

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

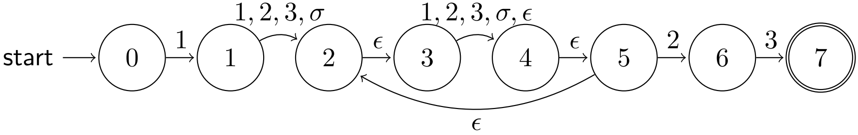
Extended Argument Matching

Building nondeterministic finite automaton for shape patterns

Pattern with only shape matching

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

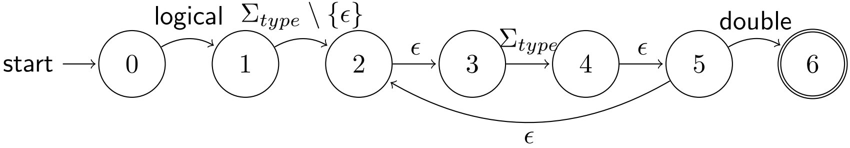
Alphabet : $\Sigma_{shape} = \{\epsilon, 1, 2, 3, \sigma\}$



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

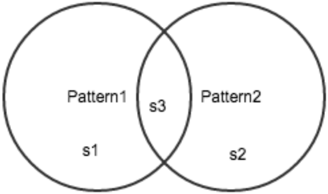
More complicate pattern

```
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.

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

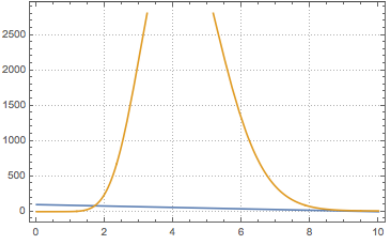


Extended Argument Matching

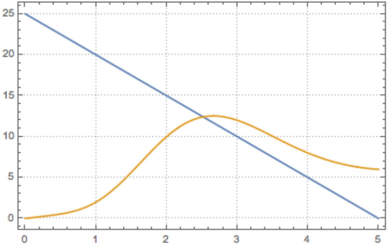
Analysis on performance

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, ...
                         );
```

Applications and Future Work

- McWeb IDE
- Sparse matrix tracing
- Type and index checking
- ...

Applications and Future Work

- McWeb IDE
- Sparse matrix tracing
- Type and index checking
- ...

- Optimizations
- OOP features
- Static code insertions
- ...