McGill University
School of Computer Science
COMP 621

Optimizing a robust AspectMATLAB front-end

Report No. 2016-09

Hongji Chen

December 16, 2016

Contents

1 Introduction and Background **2**
1.1 AspectMATLAB . . . 2
1.2 AspectMATLAB Compiler Structure . . . 3

2 AspectMATLAB Transforming Framework **5**
2.1 Expression Transformer . . . 5
2.2 Statement Transformer . . . 6
2.3 Program Transformer . . . 6

3 Transforming Framework in Action **7**
3.1 Constant Folding . . . 7
3.2 Statement Execution Tracing . . . 9
3.3 Aspect Transformation . . . 10
3.3.1 Expression Level Transformation . . . 11
3.3.2 Statement Level Transformation . . . 11
3.3.3 Program Level Transformation . . . 12

4 Conclusion **12**

A Links to Source Code and Toolkits **13**
A.1 Andrew Bodzay's AspectMATLAB++ Compiler . . . 13
A.2 New AspectMATLAB Compiler (Development) . . . 13
A.3 McSAF (with updated parser and aspect AST Node) . . . 13

List of Figures

1 Previous AspectMATLAB compiler structure . . . 3
2 New AspectMATLAB compiler structure . . . 4
3 Expression Transformer Structure . . . 5
4 Statement Transformer Structure . . . 6
5 Program Transformer Structure . . . 7
6 Aspect Expression Transformer . . . 11
7 Aspect Statement Transformer . . . 11

List of Tables

1 AspectMATLAB pattern specifiction . . . 2
2 Updated Pattern Syntax . . . 3

1 Introduction and Background

AspectMATLAB is aimed to provide aspect-oriented programming (AOP) features to a widely used scientific programming language, MATLAB, which specialized in matrix computation. However, dynamic features provided by MATLAB, such as untyped variable, and shared namespace between functions and variables, pose challenges for transforming and weaving. In the previous two version of AspectMATLAB, attempts have been made to implement robust transforming strategies, using the McSAF analysis framework. These approaches are not only hard to maintain, but also limit the flexibility of the transforming strategies implementation. Thus, in the project, we purpose a new transforming framework, sharing similar API interfaces as the previous framework, but providing a more modular way to implement and assemble the transforming strategies.

1.1 AspectMATLAB

AspectMATLAB is an aspect-oriented programming (AOP) implementation in scientific programming language MATLAB. There has been two previous version of AspectMALTAB, Toheed Aslam's AspectMatlab compiler [3], and Andrew Bodzay's AspectMatlab++ compiler [1]. The new AspectMATLAB is built on the success of the two previous version, with improvement on pattern syntax, pattern validation and argument matching capability.

Since the first version of the AsepctMATLAB, a total of 13 patterns have been defined. Each of them captures specific type of program points or restricts the searching scope for the matcher. In the previous two versions of AspectMATLAB, the compiler doesn't distinguish the type of the patterns. The lack of ability to verify the patterns, may lead to unexpected behaviours during matching and weaving. Thus, in the new version of AspectMATLAB, we classify the pattern into two different sets, the *primitive* patterns and the *modifier* patterns. The primitive patterns will allow the matcher to select at least one program point to weave the action code. However, unlike the primitive pattern, the modifier patterns don't provide any program point, but instead, they restrict the searching scope of the matcher, and allow matcher to provide more accurate result. The following table shows each pattern along with its description and classification.

Pattern	Classification	Description
Annotation	Primitive	Capture annotation comments
Call	Primitive	Capture function calls
Execution	Primitive	Capture function execution bodies
Get	Primitive	Capture matrix accesses (read)
Loop	Primitive	Capture whole loop statements
Loop Body	Primitive	Capture loop execution bodies
Loop Head	Primitive	Capture loop head expressions / statements
Main Execution	Primitive	Capture the first script / function body that executed
Operator	Primitive	Capture arithmetic operation expressions
Set	Primitive	Capture matrix accesses (write)
Scope (within)	Modifier	Restrict the searching result by its scope
Shape (dimension)	Modifier	Restrict the searching result by its shape
Type (istype)	Modifier	Restrict the searching result by its type

Table 1: AspectMATLAB pattern specifiction

An new pattern type analysis have been developed in the new AspectMATLAB to statically classify a given compound pattern. If the programmer provide a pattern with no actual join point selection (i.e. a modifier compound pattern), instead of blindly accept it, the new AspectMATLAB compiler will mark it as an error, and then terminate the compilation process. Other than pattern validation, the new AspectMATLAB change the syntax for several patterns in order to accommodate the changes in argument matching strategies, and allow programmer to write more clear patterns. The following table shows the modified pattern syntaxes

in the new AspectMATLAB, along with its syntax in the two previous versions.

Pattern	Previous Syntax	Updated Syntax
Get	get(⟨ identifier ⟩)	get(⟨ identifier ⟩:[⟨ shape ⟩]⟨ type ⟩)
Set	set(⟨ identifier ⟩)	set(⟨ identifier ⟩:[⟨ shape ⟩]⟨ type ⟩)
Call	call(⟨ identifier ⟩:[⋆,]*[...])	call(⟨ identifier ⟩:[⋆, ..., [⟨ shape ⟩]⟨ type ⟩]*)
Execution	execution(⟨ identifier ⟩:[⋆,]*[...])	execution(⟨ identifier ⟩:[⋆, ..., [⟨ shape ⟩]⟨ type ⟩]*)
Loop	loop(⟨ identifier ⟩)	loop([for, while, ⋆]:⟨ identifier ⟩)
Loop Head	loophead(⟨ identifier ⟩)	loophead([for, while, ⋆]:⟨ identifier ⟩)
Loop Body	loopbody(⟨ identifier ⟩)	loopbody([for, while, ⋆]:⟨ identifier ⟩)

Table 2: Updated Pattern Syntax

1.2 AspectMATLAB Compiler Structure

In general, the AspectMATLAB compiler consists of a lexer, a parser, a matcher and a weaver. **Lexer** splits the source file character stream into tokens. **Parser** assembles the tokens into an abstract syntax tree. **Matcher** will traverse the AST tree, and mark all the possible points for inject the action code, and **weaver** will apply appropriate modification to the AST.

In the previous two version of AspectMATLAB, attempts have been made, but not in a modular way. The following figure shows the general compiler structure of two previous version.

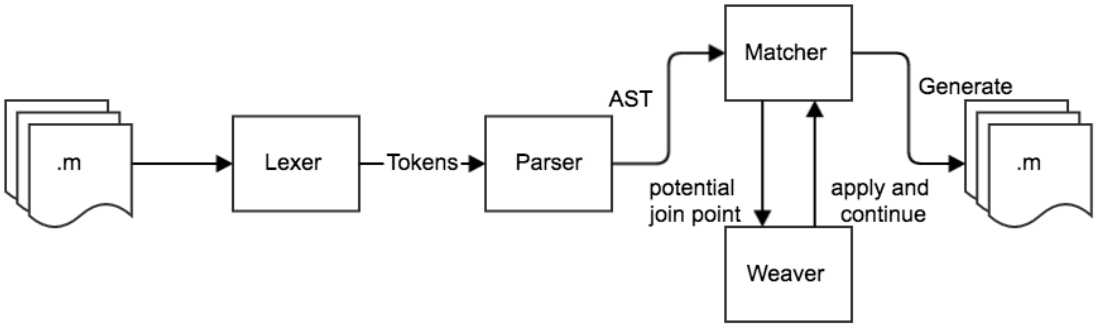

Figure 1: Previous AspectMATLAB compiler structure

As shown in the figure, the previous two version of AspectMATLAB actually not implement the matcher and weaver separately. Instead, every time when a matcher find a potential join point within the AST, it will call the weaver directly. In the first version of the AspectMATLAB (Toheed Aslam's AspectMATLAB compiler), this implementation is proved to be simple and effective. But, when the scale of the compiler grows larger, this implementation result in incomplete or even incorrect AST transformation. For example, in the second version of Aspect MATLAB (Andrew Bo dzay's AspectMATLAB++ compiler), this implementation leads to duplicate weaving,

```
1  aspect dW
2
3  patterns
4      p1 : get(*);
5      p2 : op(+);
6  end
7
8  actions
9      a1 : before p1 : (name)
10         disp(['Variable Get', name]);
11     end
12     a2 : before p2 : ()
13         disp('Plus operation');
14     end
15 end
16
17 end
```

The above aspect will attempt to insert an action before each variable access, and before each arithmetic addition. Hence, when apply to the following target code, we should only expect the matrix $x1$, and a arithmetic addition to be captured.

Listing 1: target.m

```
1  function [] = target()
2      x1 = 1 : 10;
3      x1 = x1 + sin(x1);
4  end
```

Listing 2: generated weaved code

```
1  function [] = target()
2    global AM_GLOBAL;
3    if isempty(AM_GLOBAL)
4      AM_GLOBAL.dW = dW;
5      AM_EntryPoint_0 = 1;
6    else
7      AM_EntryPoint_0 = 0;
8    end
9    x1 = (1 : 10);
10   AM_GLOBAL.dW.dW_a1('x1');
11   AM_CVar_0 = x1;
12   AM_GLOBAL.dW.dW_a1('x1');
13   AM_CVar_1 = x1;
14   AM_CVar_2 = sin(AM_CVar_1);
15   AM_GLOBAL.dW.dW_a2();
16   AM_tmpBE_0 = (AM_CVar_0 + AM_CVar_2)
17   AM_GLOBAL.dW.dW_a1('AM_tmpBE_0');
18   x1 = AM_tmpBE_0;
19   if AM_EntryPoint_0
20     AM_GLOBAL = [];
21   end
22 end
```

Listing 3: expected weaved code

```
1  function [] = target()
2    global AM_GLOBAL;
3    if isempty(AM_GLOBAL)
4      AM_GLOBAL.dW = dW;
5      AM_EntryPoint_0 = 1;
6    else
7      AM_EntryPoint_0 = 0;
8    end
9    x1 = (1 : 10);
10   AM_GLOBAL.dW.dW_a1('x1');
11   AM_CVar_0 = x1;
12   AM_GLOBAL.dW.dW_a1('x1');
13   AM_CVar_1 = x1;
14   AM_CVar_2 = sin(AM_CVar_1);
15   AM_GLOBAL.dW.dW_a2();
16   AM_tmpBE_0 = (AM_CVar_0 + AM_CVar_2)
17   x1 = AM_tmpBE_0;
18   if AM_EntryPoint_0
19     AM_GLOBAL = [];
20   end
21 end
```

As we can see in the above generated weave code, the variable AM_tmpBE_0, which is a temporary variable introduced when perform the operator expression transformation, is identified as a variable access. In order to address these problems, the compiler structure of the new AspectMATLAB has been modified. The following figure shows the general compiler structure of the new version.

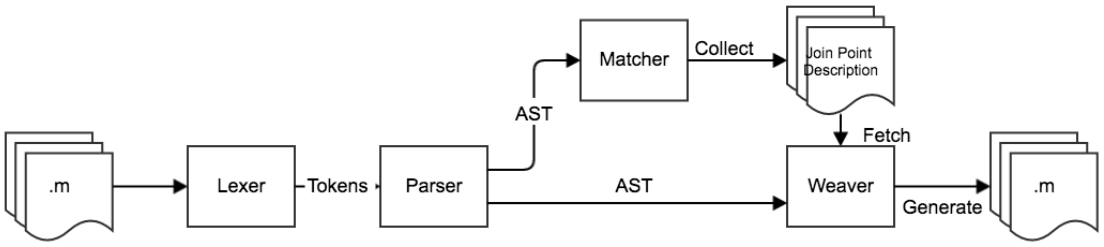

Figure 2: New AspectMATLAB compiler structure

In the new AspectMATLAB compiler, we separate the design of the matcher and the weaver. Unlike the previous two version of AspectMATLAB which the matching and weaving are implemented in one pass, the new version has them implemented in two separate passes. The matching pass communicates with the weaving pass through a set of objects called 'Join Point Descriptions'. A 'Join Point Descriptions' is triple consists of a AST node (i.e. where to insert the code), a action node (i.e. which action to apply), and the type of the join point (i.e. which weaving strategy to apply). The matcher is responsible for traversing through the AST, collecting potential join points, apply then appropriate transforming policy, and then

enveloping the join point information in to a 'Join Point Description' object. On the other hand, the weaver is responsible for fetching 'Join Point Description' object, and then apply the appropriate weaving policy.

As we have described in previous section, a robust transformation and weaving framework is a vital part to a robust AspectMATLAB compiler. In the previous two version of AspectMATLAB, the transformation framework is built on the McSAF analysis framework [2]. McSAF analysis framework is designed to be a static analysis framework instead of a AST node transforming framework. Thus, its difficult to implement a transformation using McSAF in an elegant way. Moreover, in some cases we would like to implement a in-place transformation (i.e. modify the AST node directly in the original AST). This result in potential faults in AST transverse. Thus, a new transforming framework has been introduced in the new AspectMATLAB compiler.

2 AspectMATLAB Transforming Framework

In order to address the problems we had in the previous two version of AspectMATLAB, a new transforming framework is developed. The new transforming framework shares a similar interface with the previous one. Thus the code can be easily merged to the new transforming framework. Further, the new transforming framework utilizes the generic type-safe check provided by the Java compiler, to enforce the behaviour of the transforming implementation. The new AspectMATLAB transforming framework is also designed in a modular way. The programmer can implement each part of the transformer separately, and then assemble them. The transforming framework consists of three parts - *expression transformer*, *statement transformer*, and *program transformer*. Further, the new AspectMATLAB framework defines two types of transformer - *in-place transformer* and *copy transformer*. The in-place transformer is expected to apply the transformation on the AST directly, while the copy the transformer does not modified the original AST, instead, they return a copy of transformed AST.

2.1 Expression Transformer

The expression transformer is responsible to apply the transforming policy onto expressions. There are two types of expression transformer - *in-place expression transformer* and *copy expression transformer*. The following figure shows the structure the expression transformer.

Figure 3: Expression Transformer Structure

The *CopyExprTransformer* is an implementation of copy transformer on expression transformer, and *InplaceExprTransformer* is an implementation of in-place transformer on expression transformer. Similar to previous transformer framework, the programmer can modified the behaviour of the transformer by override the methods associate the specific kind of the expression. By contract, when programmers extend their own version of *CopyExprTransformer*, the behaviour of the transformer should follows the specification defined in the copy transformer (i.e. it does not modified on the AST directly, instead, they return a copy of AST with transformation applied).

In the previous version, each node case handler is a void method, and its the callee responsibility to assemble the AST list. This makes the implementation of transformation contains lots of redundant part, and even lead to potential faults in the new implementation. Thus in the new AspectMATLAB transformation, the responsibility of assembling the transformed expression is taken by the caller method (i.e. the transformation framework transversing method). In order to achieve this features, the return value of the AST

node handle method, has been modified from void to *ast.Expr*. The programmer only need to implement the transformation to the expression and pass the transformed expression AST node as a return value.

2.2 Statement Transformer

Similar to the expression transformer, the statement transformer have two different versions - *CopyStmtTransformer* and *InplaceStmtTransformer*. The *CopyStmtTransformer* is an implementation of copy transformer on statement transformer, and *InPlaceStmtTransformer* is an implementation of in-place transform on statement transformer. The following figure shows the structure of the statement transformer.

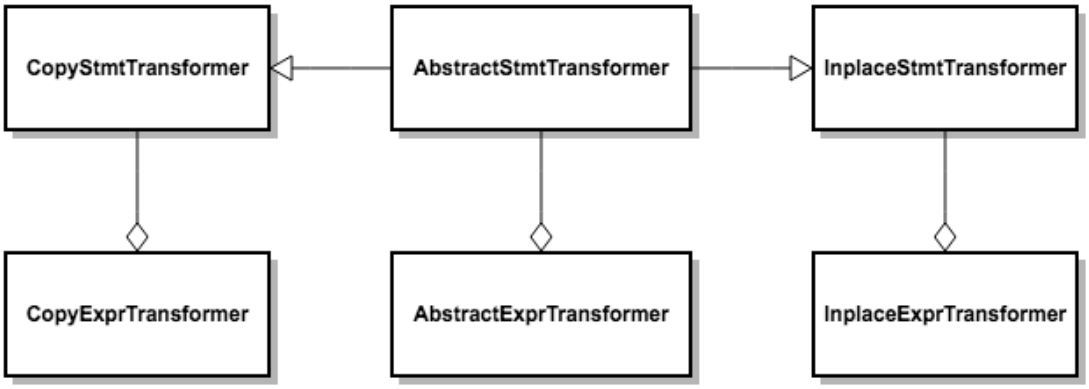

Figure 4: Statement Transformer Structure

Each statement transformer consists of a expression transformer. By contract, this expression transformer should handle all the expression transformation required by the statement transformer. Once a statement transformer instance is created, it bind with a expression transformer. Notice that, a copy statement transformer can only bind with a copy expression transformer, and a in-place statement transformer can only blind with a in-place expression transformer. Such rule is imposed by the generic type-safe check by Java. Thus, this make sure the behaviour of the transformer is consistent (i.e. a in-place statement transformer will never copy the AST, and a copy state transformer will never modified the original AST). Here are the signatures of two type of the statement transformers.

Listing 4: Statement Transformer Signatures

```
1 public class CopyStmtTransformer <T extends CopyExprTransformer>
2         extends AbstractStmtTransformer<T>;
3 public class InplaceStmtTransformer<T extends InplaceExprTransformer>
4         extends AbstractStmtTransformer<T>;
```

Like the expression transformer, return type of the statement transformer has been changed from void to *List⟨ast.Stmt⟩*. But unlike the expression transformer, which only allow exactly one expression AST node as the return value, the statement case handler methods allow a list of statement AST node. The order of assembling the return value, is exactly the same as the order of the statement AST node within the list. In this way, the programmer can implement the transforming strategy that replace one single statement, into multiple statements. The detailed examples will be covered in the next section.

2.3 Program Transformer

The program transformer represent the hight level of transformation in the transformer framework. It is responsible for transversing through the functions, scripts, class definitions and aspect definitions. Every time the programer transformer encounter a statement, a expression or a pattern, it will call its internal

statement transformer, expression transformer and pattern transformer respectively. The following figure below shows the structure of a program transformer, and signatures of program transformers.

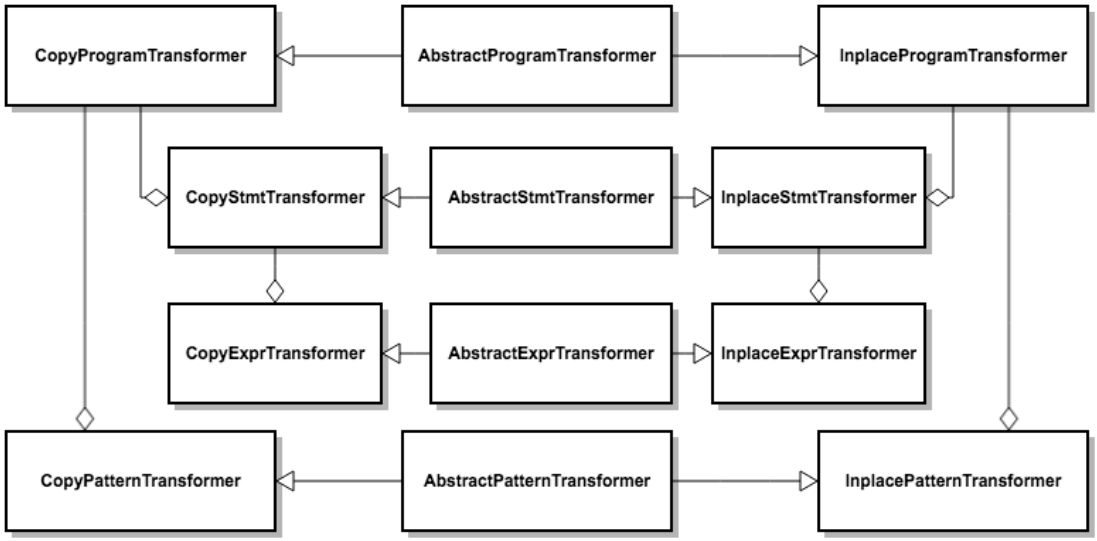

Figure 5: Program Transformer Structure

Listing 5: Program Transformer Signatures

```
1  public class CopyProgramTransformer
2          <TStmt extends CopyStmtTransformer, TPattern extends CopyPatternTransformer>
3          extends AbstractProgramTransformer<TStmt, TPattern>;
4  public class InplaceProgramTransformer
5          <TStmt extends InplaceStmtTransformer, TPattern extends InplacePatternTransformer>
6          extends AbstractProgramTransformer<TStmt, TPattern>;
```

Similar to the statement transformer, the behaviour of the program transformer is consistent, and such feature is enforced by the Java generic type-safe check. The return type of the AST node handler methods is also changed from void to $List\langle ast.ASTNode\rangle$. The programmer can simply implement the transformation strategy directly, and let the transformation framework take care of the assemble process.

3 Transforming Framework in Action

In this section, we will go over several example implementation of transforming policies using the new transforming framework. We will first start with some small examples, and finally move the the implementation of the aspect matching transformation.

3.1 Constant Folding

The simple example is constant folding. First, we need to decide if we want to build the transformer as a in-place transformer or a copy transformer. In this example, we are going to implement this transformer using the copy transformer as base class. The next step is to decide each component of the transformer, as in this example, as we are only interested in the expression (i.e. constant value expression), the only component we need to implement is the expression transformer, and simply use the trivial statement transformer and program transformer to assemble the whole transformer.

After we decide which component we need to implement, we need to consider which AST node case handler method we need to override. In this example, we aimed at constant folding, so the methods we are

going to implement are *PlusExpr*, *MinusExpr*, *MTimesExpr* and *MDivExpr* (to keep this example simple, we only focus on these four cases). Lets fist see the implementation of the *PlusExpr*,

Listing 6: Constant Folding - Plus Expression

```
1  @Override
2  protected Expr casePlusExpr(PlusExpr plusExpr) {
3      Expr lhsExpr = this.transform(plusExpr.getLHS());
4      Expr rhsExpr = this.transform(plusExpr.getRHS());
5
6      if (lhsExpr instanceof IntLiteralExpr && rhsExpr instanceof IntLiteralExpr) {
7          int lhsValue = ((IntLiteralExpr) lhsExpr).getValue().getValue().intValue();
8          int rhsValue = ((IntLiteralExpr) rhsExpr).getValue().getValue().intValue();
9
10         int resultValue = lhsValue + rhsValue;
11
12         return new IntLiteralExprBuilder().setValue(resultValue).build();
13     } else {
14         return new PlusExpr(lhsExpr, rhsExpr);
15     }
16 }
```

In the *PlusExpr* AST node case handler method, we first perform the transformation on the left-hand-side expression and right-hand-side expression recursively. Then we check if both expressions are constant integer value expression. If we we have both side of the expressions are constant integer value expression, then we perform the addition between the values, and return a *IntLiteralExpression*, otherwise, we return a copy of the original expression. In the same manner, we implement other four AST node case handler methods,

Listing 7: Constant Folding

```
1  package transformer.expr.examples;
2
3  import ast.*;
4  import natlab.FPNumericLiteralValue;
5  import transformer.expr.CopyExprTransformer;
6  import utils.codeGen.builders.IntLiteralExprBuilder;
7
8  public final class ConstantFolding extends CopyExprTransformer {
9      @Override
10     protected Expr casePlusExpr(PlusExpr plusExpr) {
11         Expr lhsExpr = this.transform(plusExpr.getLHS());
12         Expr rhsExpr = this.transform(plusExpr.getRHS());
13
14         if (lhsExpr instanceof IntLiteralExpr && rhsExpr instanceof IntLiteralExpr) {
15             int lhsValue = ((IntLiteralExpr) lhsExpr).getValue().getValue().intValue();
16             int rhsValue = ((IntLiteralExpr) rhsExpr).getValue().getValue().intValue();
17
18             int resultValue = lhsValue + rhsValue;
19
20             return new IntLiteralExprBuilder().setValue(resultValue).build();
21         } else {
22             return new PlusExpr(lhsExpr, rhsExpr);
23         }
24     }
25
26     @Override
27     protected Expr caseMinusExpr(MinusExpr minusExpr) {
28         Expr lhsExpr = this.transform(minusExpr.getLHS());
29         Expr rhsExpr = this.transform(minusExpr.getRHS());
30
31         if (lhsExpr instanceof IntLiteralExpr && rhsExpr instanceof IntLiteralExpr) {
32             int lhsValue = ((IntLiteralExpr) lhsExpr).getValue().getValue().intValue();
33             int rhsValue = ((IntLiteralExpr) rhsExpr).getValue().getValue().intValue();
34
35             int resultValue = lhsValue - rhsValue;
36
37             return new IntLiteralExprBuilder().setValue(resultValue).build();
```

```
38          } else {
39              return new MinusExpr(lhsExpr, rhsExpr);
40          }
41      }
42
43      @Override
44      protected Expr caseMTimesExpr(MTimesExpr mTimesExpr) {
45          Expr lhsExpr = this.transform(mTimesExpr.getLHS());
46          Expr rhsExpr = this.transform(mTimesExpr.getRHS());
47
48          if (lhsExpr instanceof IntLiteralExpr && rhsExpr instanceof IntLiteralExpr) {
49              int lhsValue = ((IntLiteralExpr) lhsExpr).getValue().getValue().intValue();
50              int rhsValue = ((IntLiteralExpr) rhsExpr).getValue().getValue().intValue();
51
52              int resultValue = lhsValue * rhsValue;
53
54              return new IntLiteralExprBuilder().setValue(resultValue).build();
55          } else {
56              return new MTimesExpr(lhsExpr, rhsExpr);
57          }
58      }
59
60      @Override
61      protected Expr caseMDivExpr(MDivExpr mDivExpr) {
62          Expr lhsExpr = this.transform(mDivExpr.getLHS());
63          Expr rhsExpr = this.transform(mDivExpr.getRHS());
64
65          if (lhsExpr instanceof IntLiteralExpr && rhsExpr instanceof IntLiteralExpr) {
66              int lhsValue = ((IntLiteralExpr) lhsExpr).getValue().getValue().intValue();
67              int rhsValue = ((IntLiteralExpr) rhsExpr).getValue().getValue().intValue();
68
69              if (rhsValue == 0) throw new ArithmeticException();
70              double resultValue = ((double) lhsValue) / rhsValue;
71
72              return new FPLiteralExpr(new FPNumericLiteralValue(Double.toString(resultValue)));
73          } else {
74              return new MDivExpr(lhsExpr, rhsExpr);
75          }
76      }
77  }
```

The transformer above will traverse through the AST and folding the constant valued integer plus expressions, minus expressions, matrix multiplication expressions, and matrix division expressions. Let's apply our constant-folding transformer to a MATLAB script, and here is the result.

Listing 8: Constant Folding Input Script

```
1  a = 10 + 20;
2  b = 10 - 20;
3  c = 10 * 20;
4  d = 10 / 20;
5  e = a + (20 + 30) / 10;
```

Listing 9: Constant Foling Output Script

```
1  a = 30;
2  b = (-10);
3  c = 200;
4  d = 0.5;
5  e = (a + 5.0);
```

3.2 Statement Execution Tracing

This example shows the capability of the statement transformer. In Assignment, we implement a tracing tool using the McSAF framework. In McSAF framework, a ignore set is required, in order to avoid the recursive transform problem. The new AspectMATLAB transformer framework provide a more elegant solution. The responsibility of maintain the ignore set, traverse through the AST in correct order and assemble the AST, falls on the transform framework. In this example, we are going to implement a statement tracing transformer using the in-place statement transformer.

Let's follow the same steps that we used when design the constant-folding transformer. First we need to isolate which AST node we need to perform the transformation. In this case, we need to insert a tracing

statement after each statement. Thus, we need to override the *transform* method in the in-place statement transformer.

Listing 10: Statement Tracing Transformer

```
1  package transformer.stmt.examples;
2
3  import ast.ExprStmt;
4  import ast.Stmt;
5  import ast.StringLiteralExpr;
6  import transformer.expr.InplaceExprTransformer;
7  import transformer.stmt.InplaceStmtTransformer;
8  import utils.codeGen.builders.ParameterizedExprBuilder;
9
10 import java.util.Collections;
11 import java.util.LinkedList;
12 import java.util.List;
13
14 public final class StatementTracing extends InplaceStmtTransformer<InplaceExprTransformer> {
15     public StatementTracing() {
16         super(new InplaceExprTransformer());
17     }
18
19     @Override
20     public List<Stmt> transform(Stmt statement) {
21         List<Stmt> retList = new LinkedList<>(super.transform(statement));
22         ExprStmt appendStmt = new ExprStmt(new ParameterizedExprBuilder()
23                 .setTarget("disp")
24                 .addParameter(new StringLiteralExpr(
25                         "statement of type " + statement.getClass().toString()
26                 ))
27                 .build());
28         appendStmt.setOutputSuppressed(true);
29         retList.add(appendStmt);
30         return Collections.unmodifiableList(retList);
31     }
32 }
```

The above transformer will append a *disp* statement that printing out the type of AST Node, after each statement in original AST. As this is a in-place transformer, after we invoke the *transform* method, all AST modifications are applied to the original AST. The following are the result that we apply this transformer to a example program.

Listing 11: Statement Tracing Input

```
1  function test()
2      x = 1;
3      y = 2;
4      if x < y
5          disp('then branch');
6      else
7          disp('else branch');
8      end
9  end
```

Listing 12: Statement Tracing Output

```
1  function [] = test()
2    x = 1;
3    disp('statement of type class ast.AssignStmt');
4    y = 2;
5    disp('statement of type class ast.AssignStmt');
6    if (x < y)
7      disp(''''then branch'''');
8      disp('statement of type class ast.ExprStmt');
9    else
10     disp(''''else branch'''');
11     disp('statement of type class ast.ExprStmt');
12   end
13   disp('statement of type class ast.IfStmt');
14 end
```

3.3 Aspect Transformation

The final example that will given in this report is the AspectMATLAB aspect transformation. Before weave the action code, we need to apply appropriate transformation to the AST to make the join point exposed. The new AspectMATLAB compiler takes the advantage of the modularity of the new transformation framework.

Instead of build all the transformation in a giant class (the previous approach), we split the implementation of the transformation in to multiple layers. The first layer is the expression level, this include the *get*, *set*, *call*, and *operator* patterns. The second layer is the statement level. This include the *loop*, *loophead*, ,*loopbody* and *annotation* pattern. The statement level transformation is also responsible for injecting the pre-expression statements, and post-expression statements that generated by the expression level transformer. The final layer of the transformation framework is the program level transformer. The program transformer focus on the *execution*, *mainexecution* pattern.

3.3.1 Expression Level Transformation

At this level, we focus on each element with the expression, and we generate two set of statements - the pre-expression statements, and post-expression statements. During the expression transformation, it is inevitable to generate temporary variables, and assign values to them. However, this is impossible to be performed only at expression level. Thus, we add the pre-expression statements set, and the post-expression statements set. The pre-expression statements set contains the statements that will be inject just before the expression, and the post-expression statement set contains the statements that will be inject just after the expression. The following figure demonstrate the process of a expression transformer.

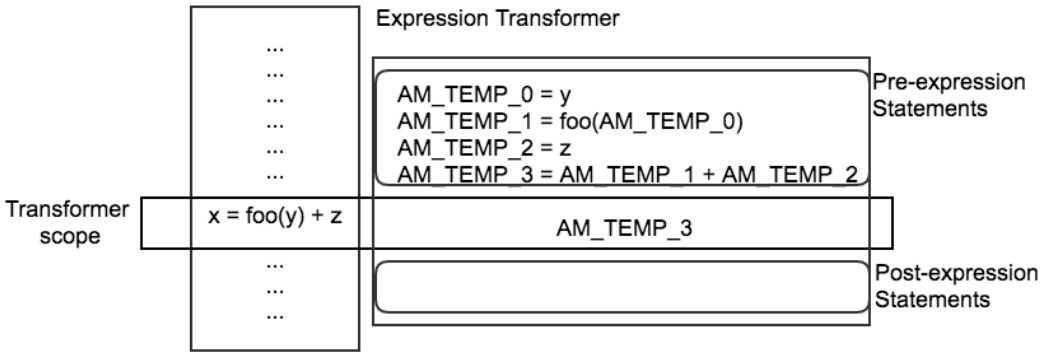

Figure 6: Aspect Expression Transformer

3.3.2 Statement Level Transformation

The statement transformation is the second layer of the aspect transformation framework. It has two major

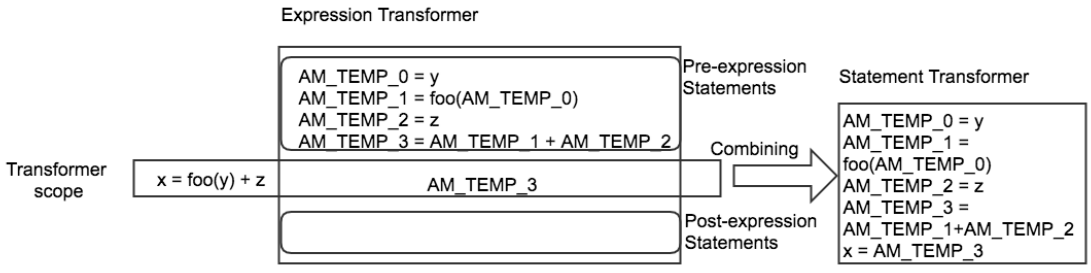

Figure 7: Aspect Statement Transformer

responsibilities, the first one is to apply the appropriate transformation to *annotation*, *loop*, *loophead*, and *loopbody* pattern. The second responsibility is to handle the pre-expression statement and post-expression

statement requests generated by the expression transformer. As in the new transformation framework, the return type of the statement transformation has been changed from void to $List\langle ast.Stmt\rangle$, the implementation of the statement transformer is quite straight forward. The above diagram demonstrate the process of a statement transformer.

3.3.3 Program Level Transformation

The program level transformation is the final piece in the aspect transformation puzzle. Like statement level transformation it has two major responsibilities - to handle the request generate by the statement level transformation, and to apply appropriate transformation for the *execution* and *mainexecution* pattern. After finish the transformation, the program level transformer is also responsible to check the validity of the transformed AST, for example, to make sure there do not exist a self-loop within the AST. This make sure the transformation is robust, and eliminate the possibility of unsound transformer implementation. Unlike the previous version of transformation framework, we can implement these parts modularly. This makes the compiler easy to maintain, and flexible for further development.

4 Conclusion

In this project, we go over the details of implement an AspectMATLAB front-end, and purpose a new modular transforming framework for more robust transformation design. The new transformation framework split the whole transformation process into three different parts, expression transformation, statement transformation and program transformation. At each level, programmer can focus on different aspect of design. The new transformation framework also utilize the Java generic type-safe check mechanism. This make sure the behaviour of the transformer is consistent (i.e. the in-place transformer will also apply the transformation directly on the original AST, while the copy transformer will return a copy of AST with modification applied).

A Links to Source Code and Toolkits

A.1 Andrew Bodzay's AspectMATLAB++ Compiler

https://github.com/Sable/AspectMatlab

A.2 New AspectMATLAB Compiler (Development)

https://github.com/Sable/AspectMATLAB-robust

A.3 McSAF (with updated parser and aspect AST Node)

https://github.com/Sable/mclab-parser

References

[1] Laurie Hendren Andrew Bodzay. Aspectmatlab++: Annotations, types, and aspects for scientists. *MODULARITY*, 2015.

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

[3] Anton Dubrau Laurie Hendren Toheed Aslam, Jesse Doherty. Aspectmatlab: An aspect-oriented scientific programming language. *AOSD*, 2010.