A Bytecode Level Operational Semantics Analysis on Aspectj Pointcut Matching

Hongxu Chen*

Abstract

Aspect Oriented Programming(AOP) can be viewed as a complement for Object Oriented Programming(OOP) with the ability to support *seperation of concerns*.There have been many implementations for AOP,among which Aspectj is the most well known.Aspectj is now an extension for java development toolkit and usually used as a plugin for IDEs like eclipse.Apart from most of the other tools,it uses the aspectj compiler(ajc) to weaving the advice into original source code statically on the bytecode level.This review is based on [1] describes the operational semantics of aspectj's static advice weaving with a subset of Aspectj and Java Virtual Machine Language Instruction(JVMLI).

keywords: aspectj,bytecode,static pointcuts

*STAP@SJTU 1110379002

1 Introduction

Aspect Oriented Programming(AOP) is a new way of modularizing programs compared to object oriented programming (OOP).It's designed to solve two central problems in OOP: `code tangling` and `code scattering`, which refers to one module with many concerns and many modules with one concern respectively. A common example is the logging behaviour for a program, which may be scattered across methods, classes, object hierarchies, or even entire object models(We will give a simple example below to explain). AOP improves code reuse across different object hierarchies by providing explicit support for separation of concerns. Due to this benefit, AOP is usually used for logging,verification,policy enforcement,security management,profiling,memory management,visualization of program executions and so on.

Since the concepts explicitly introduced by Gregor Kiczales in 1997[4], AOP has been widely used,especially in enterprise applications.There has now been quite a lot of implementations of AOP,such as Aspectj, aspectwerkz, Spring AOP,Aspect.NET etc. This review is mainly about the most well-known Aspectj designed by Gregor et.al and focuses on the java bytecode level semantics.

In aspectj aspects are woven into programs statically,so the bytecode corresponding to the original source files would be changed as long as there is an advised being matched.It illuminates us that we can model the semantics on the bytecode rather than source code,which is calculus into which source-level AOP constructs can be translated. This makes sense since it can be applied to other JVM programming languages such as Scala,JRuby,Jython etc.

The remainder of this report is organized as follows:Section 2 talks about some background knowledges for AOP and aspectj,also gives a simple example for the readers.Section 3 is tells the environments and configurations of the semantics.Next section lists all the utility functions for the rules.Section 5 deduces five rules of related to the weave process.And the last section makes a summary.

2 Preliminary

2.1 Terminologies

There are some basic terminologies in program with AOP/aspectj:

jointpoint well-defined points in the execution of a program.It requires that the joinpoints are identifiable by the AOP tools.Those join points include class constructions,get/set operations,method calls,method executions etc.

Pointcut a constructor that designates a set of join points defined in *aspectj file*(file with suffix of .aj).

advice pieces of code attached to pointcuts and executed when join points satisfying their pointcuts are reached.Formally it is written as:$Advice = kind \times pointcut \times code$,where $kind \in \{before, after, around\}$.

aspect set of advices.

2.2 Aspectj Weaving Mechinism

Since `Aspectj1.1`(the current version is 1.5),the advice weaving has been based on bytecode transformation rather than on source code. In this case,the AspectJ compiler(ajc for short) is composed of two stages.

front-end compiler an extension to the Java compiler and compiles applications and aspects into pure Java bytecode enriched with additional annotations to handle non pure Java information as advices and pointcuts.

back-end compiler weaves compiled aspects with compiled applications producing woven class files.

2.3 Pointcut Designators

As to compile time information access,there are two kinds of pointcuts:the *static* pointcut and the *dynamic* one.While static pointcut can be directly mapped to the original code, we have to include conditional

logic(called `residuals`) to check the dynamic properties with the dynamic pointcuts.This paper only talks about the former.Refer to more details in [2] for more infomation.
So we only consider the designators below 8 kinds of pointcuts. The regular expression patterns for pointcuts

designator	**matches**	**Name in this paper**
call	calls to a `method` or constructor	`mcall`
execution	execution of a method or constructor	`mexecution`
get	the reference to a class attribute	`get`
set	the assignment of a class attribute	`set`
staticinitialization	execution of a class's static initialization	`staticInit`
adviceexecution	on advice join points	`aexecution`
withincode	join points within a method/constructor	`withincode`
within	join points within a specific class type	`within`

Table 1: static pointcut designators

such as `some_method(..)` or `*` is also omitted to ease the semantics reading since they can be extended accurately in compile time.

2.4 JVM Language Instructions

To make it simpler,we also use a subset of JVMLI for our research.In this case,those instructions used for efficiency in the JVM specification is also ignored(such as `iconst_0`);also it omits some primitive type for JVM such as float,array and disable the cast operations(so there are no more `anewarray` or `d2l`).Also the parts of the other bytes are simplified as numbers(i) or address(adr) for short. The instructions we used is as below:

pop	push n	dup	iadd
return	ireturn	areturen	athrow
aload i	iload i	astore i	istore i
getfield i	putfield i	getstatic i	putstatic i
invokeinterface i,n	invokestatic i	ifeq adr	ifne adr
invokevirtual i	invokespecial i	goto adr	new i
monitorenter	monitorexit		

Table 2: JVML Opcode set

2.5 An Example

Here we give a very simple example to exaplain how aspectj advice is woven.

Listing 1: original file Hello.java

```java
public class Hello {

    public static void sayHello(){
        System.out.println("hello");
    }


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

Listing 2: aspectj file World.aj

```java
public aspect World {
    pointcut greeting() : execution(* Hello.sayHello(..));

    after() returning() : greeting() {
        System.out.println(" world!");
    }

    before():greeting(){
        System.err.println("hi,");
    }
}
```

The source file `Hello.java` can only output *hello* without aspectj advice.However when we compile the source file with the aspectj `World.aj` using `javac` and `ajc`,the new program's output changes to *hi,hello world!*.

So how did it happen? The key is that we add a `pointcut` *greeting* in the aspectj file, which matches the `join point` *sayHello*(a static method) in file `Hello.java`. There is also a designator called **execution** that matches the exact *execution* site of the code. This pointcut corresponds to a so-called cross cut concern of the original file, but notice that it is defined in the aspectj file.With the advice kind like **before()** we get an advice,which usually will do some additions in respect to original files.In this example,it means when the stack of JVM runs into the execution of method `sayHello`(this designator is `mexecution`),there will be a additional behaviour that output *hi,* before the output of *hello*. The `after` advice is similar except that it is called just before the context of method `sayHello` is popped.

So the common practice when we using Aspectj(or generally AOP),we need firstly define **pointcuts** in the aspect file to match `join point` in the original file.Then we write `advice` for those `pointcuts`.Note that advice and pointcut is not one-to-one mapped,that is to say the relationship of them can be $1-norn-1orm-n$. If all works fine,we weave our advice into the original bytecode. The benefit is that we do not need to rewrite the existent source file while we change the behaviour.

Let's look the minor differences in the bytecode level. By using `javap -c` we get the disassamble infomation of `sayHello()` in `Hello` class.

Listing 3: sayHello() function in Hello class file

```
public static void sayHello();
  Code:
     0: invokestatic #43 // Method date20120901/AspectjHello/World.aspectOf:()Ldate20120901/AspectjHello/World;
     3: invokevirtual #49 // Method date20120901/AspectjHello/World.ajc$before$date20120901
     _AspectjHello_World$2$f69f5afa:()V
     6: getstatic #9 // Field java/lang/System.out:Ljava/io/PrintStream;
     9: ldc #15 // String Hello
    11: invokevirtual #17 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    14: nop
    15: invokestatic #43 // Method date20120901/AspectjHello/World.aspectOf:()Ldate20120901/AspectjHello/World;
    18: invokevirtual #46 // Method date20120901/AspectjHello/World.ajc$afterReturning$date20120901
        _AspectjHello_World$1$f69f5afa:()V
    21: return
```

The index `0,3` and `15,18` describes the added bytecode to the original class file,which is what our advice `before` and `after` have done.We can also look at in the class file which `ajc` compiles.

Listing 4: aspectOf() in World class file

```
  public static date20120901.AspectjHello.World aspectOf();
  Code:
     0: getstatic #63 // Field ajc$perSingletonInstance:Ldate20120901/AspectjHello/World;
     3: ifnonnull 19
     6: new #65 // class org/aspectj/lang/NoAspectBoundException
     9: dup
    10: ldc #67 // String date20120901_AspectjHello_World
    12: getstatic #15 // Field ajc$initFailureCause:Ljava/lang/Throwable;
    15: invokespecial #69 // Method org/aspectj/lang/NoAspectBoundException."<init>":
        (Ljava/lang/Strin g;Ljava/lang/Throwable;)V
    18: athrow
    19: getstatic #63 // Field ajc$perSingletonInstance:Ldate20120901/AspectjHello/World;
    22: areturn
```

Listing 5: before advice in World class file

```
 public void ajc$before$date20120901_AspectjHello_World$2$f69f5afa();
 Code:
    0: getstatic #56 // Field java/lang/System.err:Ljava/io/PrintStream;
```

3: ldc #59 // *String hi*
5: invokevirtual #48 // *Method java/io/PrintStream.println:(Ljava/lang/String;)V*
8: **return**

2.6 Reduce Advice Kind

In addition, we also reduce the kind of advice to `before` and `after`. In fact,we only consider the `after` advice if the execution of a join point completes normally,and that is to say we only cope with the `after returning` advice,`after throwing` and the simple `after` is neglected.The `around` advice can be modeled as the combination of `before` and `after`,however it is not proven here. The `privilege properties,aspect inheritance` and other aspectj features are also ignored here.

3 Environments and Configurations

3.1 Notations

Since the `ajc` compiler also compiles the aspectj file into java class file(in the above example,the `World.aj` is compiled into `World.class`),we consider that an aspect can be viewed as a class and its advices are represented by the methods of the class. In this case,we transfer our focus into the class files.

Firstly we give some notations that will be needed in later discussions.

- $X \xrightarrow{m} Y$ denotes the set of all `partital functions`(also known as `maps`) from set X to Y. For $m \in X \xrightarrow{m} Y$,it can be extended as $m = [x_0 \mapsto y_0, \ldots x_{n-1} \mapsto y_{n-1}]$,where $x_i \in X$ and $y_i \in Y, i \in [0 \ldots n-1]$.
- For a map $m \in X \xrightarrow{m} Y$,$Dom(m) = X$
- Given a map f,write $f[x \mapsto v]$ to denote the updating operation of f that yields a map that is equivlent to f except that x is from now associated with v
- Given a record space $D = \langle f_1 : D_1, \ldots, f_n : D_n \rangle$(where f_i is a field with the type D_i defined in the specification),and an element e of type D,the access to the filed f_i of e is written as $e.f_i$ and the update of fields $f_{i1}, \ldots, f_{ik}$ is written $e[f_{i1} \leftarrow v_{i1}, \ldots, f_i \leftarrow v_{ik}]$.
- $(\tau) - set$ and $(\tau) - list$ denote the `set` or `list` of type τ respectively.

3.2 Pointcuts, Joinpoint Shadows and Advices

As with the static designators we talk about here,there are 3 cases of join point shadows.

1. The case where the shadow is *exactly one* instruction
2. The case where the shadow is an *entire* method
3. The case that does not by themselves define shadows.They use the shadows defined by the other 6 static pointcuts.

The `before` advice code segment is inserted after the `impdep1` instruction.`impdep1` is a reserved word in java bytecode opcode but can never appear in any class file; we assume the front-end compiler produces the instructioin for us to insert the advice;and after the compilation of the back-end compiler,the advice is woven and `impdep1` is delimitted.The `after` advice code is easy to recognize since it always usually appears with method return opcodes like `return,ireturn,areturn`.

Table 3 refers to the pointcut and the afftected area while the box below includes the syntax of our defined pointcuts.

Join Point Designator	Join Point shadow
`mcall`	`invokevirtual` i,`invokespecial` i, `invokestatic` i, `invokeinterface` i, n
`get`	`getfield` i ,`getstatic` i
`set`	`putfield` i ,`putstatic` i
`mexecution`	Entire method code
`execution`	Entire advice's code
`staticInit`	Entire "`clinit`" method code

Table 3: pointcut and affected bytecode

$$
\begin{aligned}
BasePcut \models & \texttt{mcall}(MethodPattern) \ | \ \texttt{mexecution}(MethodPattern) \ | \ \texttt{aexecution()} \\
& | \ \texttt{withincode}(MethodPattern) \ | \ \texttt{get}(FieldPattern) \ | \ \texttt{set}(FieldPattern) \\
BooleanPcut \models & Pcut \ \texttt{or} \ Pcut \ | \ \texttt{not} \ Pcut \ | \ Pcut \ \texttt{and} \ Pcut \\
MethodPattern \models & \langle methodModifiers : (Methodmodifier) - \texttt{set}, methodSgnature : MethodSignature, \\
& componentType : ComponentType \rangle \\
FieldPattern \models & \langle fieldModifiers : (FieldModifier) - \texttt{set}, FieldSignature : FieldSignature, \\
& componentType : ComponentType \rangle \\
MethodSignature \models & \langle name : \texttt{Identifier}, argumentsType : (Type) - \texttt{list}, resultType : Type \rangle \\
FieldSignature \models & \langle name : \texttt{Identifier}, type : Type \rangle \\
MethodModifier \models & \texttt{public} \ | \ \texttt{private} \ | \ \texttt{static} \ | \ \texttt{synchronized} \\
FieldModeifer \models & \texttt{public} \ | \ \texttt{private} \ | \ \texttt{static} \\
ComponentType \models & ReferenceType \ | \ AspectType \\
ResultType \models & Type \ | \ \texttt{void} \\
Type \models & PrimitiveType \ | \ ReferenceType \\
ReferenceType \models & ClassType \ | \ InterfaceType \\
ClassType \models & \texttt{Identifier} \\
InterfaceType \models & \texttt{Identifier} \\
AspectType \models & \texttt{Identifier} \\
PrimitiveType \models & \texttt{Identifier}
\end{aligned}
$$

3.3 Environments

Below describes the `environment` before weaving, it contains the `javaEnvironment` and the `advices`.We can denote the environment as:

$$Environment \models \langle javaEnvironment : JavaEnvironment, advices : (AdviceInfo) - \texttt{list} \rangle$$

The `javaEnvironment` is somewhat like the actual jvm environment except that it's reduced and there is an additional instruction `impdep1`. In our model,a class is just a record containing a constant pool, a super-class, a set of interfaces, a list of fields, a map that associates values to static fields, a list of methods, three flags that indicate whether the class is initialized or not, is an interface or not, is an aspect or not and a monitor. We only explain some productions,details can been seen in [3].

- A `constant pool` is a map that associates a set of integers with a set of constant pool entries,which is like this:

  $$ConstantPool \models Int \xrightarrow{m} ConstantPoolEntry$$

  In our example,we can use `javap -verbose Hello.class` to see what contains in the constant pool.

Listing 6: constant pool

```
#1 = Class #2 // date20120901/AspectjHello/Hello
#2 = Utf8 date20120901/AspectjHello/Hello
#3 = Class #4 // java/lang/Object
#4 = Utf8 java/lang/Object
#5 = Utf8 sayHello
#6 = Utf8 ()V
#7 = Utf8 org.aspectj.weaver.MethodDeclarationLineNumber
#8 = Utf8 Code
#9 = Fieldref #10.#12 // java/lang/System.out:Ljava/io/PrintStream;
#10 = Class #11 // java/lang/System
#11 = Utf8 java/lang/System
#12 = NameAndType #13:#14 // out:Ljava/io/PrintStream;
#13 = Utf8 out
...
```

The index first appearing in each line is the *key* `Integer`(#5 for instance) and the content(like `sayHello`) can be viewed as the *value* of the entries.

- A `constant pool entry` can be a class type, a pair of a method signature and a supposed class.Those entries are usually defined by the `type` and the concatenation of String of entries. The recognition and interpretation, however is the task of JVM, which is out of our topic.

- The `monitor` associated with a class is a record of three components: `threadOwner`, `depth` and a `waitList`, i.e. it is can be represented as a tuple like below:

$$Monitor \models \langle threadOwner : Threadowner, depth : Nat, waitList : WaitingList \rangle$$

  The monitor is set to $\langle None, 0, [\,] \rangle$ if the class is not locked.

Other productions can be seen as follows:

$$
\begin{aligned}
JavaEnvironment \models & ComponentType \xrightarrow{m} Class \\
Class \models & \langle constantPool : ConstantPool, superClass : ClassType | NoneType, \\
& interfaces : (ClassType) - \texttt{set}, fields : (Field) - \texttt{set}, \\
& staticMap : Field \xrightarrow{m} Value, methods : (Method) - \texttt{set}, \\
& initialized : Int, interface : Int, aspect : Int, monitorClass : Monitor \rangle \\
ConstantpoolEntry \models & ClassType \ \mid \ MethodPoolEntry, FieldPoolEntry \\
MethodPoolEntry \models & \langle methodSignature : MethodSignature, supposedClass : ClassType \rangle \\
FieldPoolEntry \models & \langle fieldSignature : FieldSignature, supposedClass : ClassType \rangle \\
ThreadOwner \models & ThreadId | NoneType \\
WaitingList \models & (ThradId) - \texttt{list} \\
ThreadId \models & Nat \\
Field \models & \langle fieldSignagure : FieldSignature, fromClass : ComponentType, \\
& fieldModifiers : (FieldModifier) - \texttt{set} \rangle \\
Method \models & \langle methodSignagure : MethodSignature, fromClass : ComponentType, \\
& methodModifiers : (MethodModifier) - \texttt{set}, code : Code, method \\
& Variables : MethodVariables \rangle \\
AdviceInfo \models & \langle kind : \{\texttt{Before,After}\}, pointcut : Pcut, \\
& fromClass : AspectType, adviceSignagure : MethodSignagure \rangle \\
Code \models & ProgrammCounter \xrightarrow{m} Instruction \\
Instruction \models & JVMLInstruction | \texttt{impdep1} \\
ProgramCounter \models & Nat \\
MethodVariables \models & (Value) - list
\end{aligned}
$$

3.4 Configurations

The operational semantics is based on the evolution of configurations that are defined hereafter. Weaving a class with some aspects is the result of weaving all its methods with the considered aspects. So we only need to describe the weaving inside one method and a configuration will have the following form:

$$\langle \xi, m, pc, ads, nextpc \rangle$$

where:

- ξ represents the environment
- **m** is the related method
- **pc** represents the program counter that contains the address of the instruction to be advised into m
- **ads** represents the advices to consider
- **nextpc** represents the program counter for the next instruction to consider

4 Utility Functions

This section contains the details of the useful utility functions for the final regualation.

4.1 List Processing

The function `head` returns the first element in a given list and `tail` returns the remains.They are just like `LISP`'s `car` and `cdr` operations.

$$\begin{aligned} head:\ (\tau) - list \to \tau \\ head(v :: l) = v, \forall (v, l) \in \tau \times (\tau) - \texttt{list} \end{aligned}$$

$$\begin{aligned} tail:\ (\tau) - list \to \tau \\ tail(v :: l) = l, \forall (v, l) \in \tau \times (\tau) - \texttt{list} \\ tail([\,]) = [\,] \end{aligned}$$

4.2 Constructors and Retriever for Methods and fields

- The function `signatureAspectOf` returns the signature of the method "`aspectO`" of the advice aspect.We can see in listing `sayHello` that there is a line with the `invokestatic` of `date20120901/AspectjHello/World.aspectOf`. This method in the aspect usually calls the actual advice method that we would like to weave(just like the `invokevirtual` of `date20120901/AspectjHello/World.ajc$before$date20120901` in our `Hello` example).

$$\texttt{signatureAspectOf}:\ AdviceInfo \to MethodSignature$$

$$\texttt{signatureAspectOf}(ad) = ms \ \texttt{iff} \begin{cases} ms.name = \text{“}aspectOf'' \\ ms.argumentsType = [\,] \\ ms.resultType = ad.fromClass \end{cases}$$

- `retrieveF` or `retrieveM` searches a method/field in a (method/field)-`list` respectively.

$$\texttt{retrieveF}:\ FieldSignature \times Fields \to Field$$

$$\texttt{retrieveF}(fs, l) = \begin{cases} \texttt{head(l) if } \ \texttt{head}(l).FieldSignature = ms \\ \texttt{retrieveF}(fs, tail(l) \ \ \texttt{otherwise} \end{cases}$$

$$\texttt{retrieveM}:\ MethodSignature \times Methods \to Method$$

$$\texttt{retrieveM}(ms, l) = \begin{cases} \texttt{head(l) if head}(l).MethodSignature = ms \\ \texttt{retrieveM}(ms, tail(l) \ \texttt{otherwise} \end{cases}$$

- `newPoolEntry` returns a method that has the given method signature ms and its class c.

$$\texttt{newPoolEntry}:\ MethodSignature \times Class \to ConstontPoolEntry$$

$$\texttt{newPoolEntry}(ms, c) = c \ \texttt{iff} \begin{cases} c.methodSignagure = ms \\ c.supposedClass = c \end{cases}$$

4.3 Functions for Checking Shadows

- `isReturn` judges if it comes to the end of some method and it checks whether the current instruction is one of those *return* opcodes.

$$\texttt{isReturn}:\ Method \times ProgramCounter \to Boolean$$

$$\texttt{isReturn}(m, pc) = true \text{ iff } m.code(pc) = return \lor areturn \lor ireturn$$

- `isBeforeOrAfterShadow` tests the possibility of being a shadow(`before` or `after`).

$$\begin{aligned} &\texttt{isBeforeOrAfterShadow}(m, pc) = true \\ &\quad \texttt{iff } m.Code(pc) = \texttt{invokevirtual } i | \texttt{invokespecial } i | \texttt{invokestatic} | \ \texttt{invokeinterface } i, n \\ &\quad\quad | \texttt{getstatic } i | \texttt{getfield } i | \texttt{putstatic } i | \texttt{putfield } i \end{aligned}$$

- `isBeforeShadow` additionally check if the current instruction is the reserved `impdep1`;while `isAfterShadow` check if the method would exit.

$$\texttt{isBeforeShadow}: \ Method \times ProgramCounter \rightarrow Boolean$$

$$\texttt{isBeforeShadow}(m, pc) = \texttt{true iff} \begin{cases} \texttt{isBeforeOrAfterShadow}(m, pc) \\ m.Code(pc) = \texttt{impdep1} \end{cases}$$

$$\texttt{isAfterShadow}: \ Method \times ProgramCounter \rightarrow Boolean$$

$$\texttt{isAfterShadow}(m, pc) = true \ \texttt{iff} \begin{cases} \texttt{isReturn}(m, pc) \\ \texttt{isBeforeOrAfterShadow}(m, pc) \end{cases}$$

- `isMpatternMatched` and `isFpatternMatched` decides whether the special method or field matches the pattern.

$$\texttt{isMpatternMatched}(mp, m) = true \quad \texttt{iff} \begin{cases} mp.methodSignature = m.methodSignature \\ mp.componentType = m.fromClass \\ mp.methodModifiers = m.methodModifiers \end{cases}$$

$$\texttt{isFpatternMatched}(fp, f) = true \quad \texttt{iff} \begin{cases} fp.fieldSignature = f.fieldSignature \\ fp.componentType = f.fromClass \\ fp.fieldModifiers = f.fieldModifiers \end{cases}$$

4.4 After Advice Matches

- `matchAfterAexec` checks whether it is inside an aspect and reaches the end of some method.

$$\texttt{matchAfterAexec}: \ Method \times ProgramCounter \rightarrow Boolean$$

$$\texttt{matchAfterAexec}(m, pc) = true \ \texttt{iff} \begin{cases} (m.fromClass).aspect = 1 \\ \texttt{isReturn}(m, pc) \end{cases}$$

- `matchAfterMexec` should check whether it matches the current method pattern.Note that the methods in those aspects are also tested.

$$\texttt{matchAfterMexec}: \ MethodPattern \times Method \times ProgramCounter \rightarrow Boolean$$

$$\texttt{matchAfterMexec}(mp, m, pc) = true \ \texttt{iff} \begin{cases} \texttt{isMpatternMatched}(mp, m) \\ \texttt{isReturn}(m, pc) \end{cases}$$

- `matchAfterStaticInit` is called when the advice is an `After` advice and its pointcut is `staticInit`(ct).

$$\texttt{matchAfterStaticInit}: \ ComponentType \times Method \times ProgramCounter \rightarrow Boolean$$

$$\texttt{matchAfterStaticInit}(ct, m, pc) = true \ \texttt{iff} \begin{cases} m.fromClass = ct \\ (m.methodSignature).name = "clinit" \\ \texttt{isReturn}(m, pc) \end{cases}$$

- `matchAfterPcut` can be viewed as a summary of the above,it returns true if the given `After` advice is applicable in the method m at the program counter pc:

$$\texttt{matchAfterPcut}: \ Environment \times Pointcut \times Method \times ProgramCounter \rightarrow Boolean$$

It supports the logical operators `and`,`or` and `not`.

$$\begin{aligned} \texttt{matchAfterPcut}(\xi, pcut1 \ \texttt{and} \ pcut2, m, pc) &= \texttt{matchAfterPcut}(\xi, pcut1, m, pc) \wedge (\xi, pcut2, m, pc) \\ \texttt{matchAfterPcut}(\xi, pcut1 \ \texttt{or} \ pcut2, m, pc) &= \texttt{matchAfterPcut}(\xi, pcut1, m, pc) \vee (\xi, pcut2, m, pc) \\ \texttt{matchAfterPcut}(\xi, \texttt{not} \ Pcut, m, pc) &= \neg\texttt{matchAfterPcut}(\xi, pcut, m, pc) \end{aligned}$$

Note the relationship between matchAfterPcut and the functions above:

$$\begin{aligned}
\texttt{matchAfterPcut}(\xi, mexecution(mp), m, pc) &= \texttt{matchAfterMexec}(mp, m, pc)\\
\texttt{matchAfterPcut}(\xi, staticInit(ct), m, pc) &= \texttt{matchAfterStaticInit}(ct, m, pc)\\
\texttt{matchAfterPcut}(\xi, aexecution(), m, pc) &= \texttt{matchAfterAexec}(m, pc)
\end{aligned}$$

In the other cases,we have:

$$\texttt{matchAfterPcut}(\xi, pcut, m, pc) = \texttt{matchPcut}(\xi, pcut, m, pc) \ \texttt{iff} \begin{cases} pcut \in BasicPcut \\ pcut \neq \texttt{mexecution}(mp) \\ pcut \neq \texttt{staticInit}(ct) \\ pcut \neq \texttt{aexecution}() \end{cases}$$

4.5 Before Advice Matches

- matchBeforeExec returns true if the given pointcut *pcut* of a before advice matches as a *method* or *advice execution* of the given method.

$$\texttt{matchBeforeExec}: \ Pointcut \times Method \rightarrow Boolean$$

$$\begin{aligned}
\texttt{matchBeforeExec}(mcall(mp), m) &= false\\
\texttt{matchBeforeExec}(get(fp), m) &= false\\
\texttt{matchBeforeExec}(set(fp), m) &= false\\
\texttt{matchBeforeExec}(withincode(mp), m) &= \texttt{isMpatternMatched}(mp, m)\\
\texttt{matchBeforeExec}(within(ct), m) &= (m.fromClass = ct)\\
\texttt{matchBeforeExec}(mexecution(mp), m) &= \texttt{isMpatternMatched}(mp, m)\\
\texttt{matchBeforeExec}(staticInit(ct, m)) &= ((m.fromClass = ct) \wedge ((m.methodSignature).name = "clinit"))\\
\texttt{matchBeforeExec}(aexecution(), m) &= ((m.fromClass).aspect = 1)
\end{aligned}$$

And the logical operations are similar:

$$\begin{aligned}
\texttt{matchBeforeExec}(pcut1 \ and \ pcut2, m) &= \texttt{matchBeforeExec}(pcut1, m) \wedge \texttt{matchBeforeExec}(pcut2, m)\\
\texttt{matchBeforeExec}(pcut1 \ or \ pcut2, m) &= \texttt{matchBeforeExec}(pcut1, m) \vee \texttt{matchBeforeExec}(pcut2, m)\\
\texttt{mtchBeforeExec}(not \ pcut, m) &= \neg\texttt{matchBeforeExec}(pcut, m)
\end{aligned}$$

- matchBeforeOtherExec returns true if the given pointcut *pc* of a before advice matches with the instruction at the position *pc* in the method *m* *not* as method or advice execution!(The corresponding and,or,not operations are the same and is not listed.)

$$\texttt{matchBeforeOtherExec}: \ Environment \times Pointcut \times Method \times ProgramCounter \rightarrow Boolean$$

$$\begin{aligned}
\texttt{matchBeforeOtherExec}(\xi, mexecution(mp), m, pc) &= false\\
\texttt{matchBeforeOtherExec}(\xi, staticInit(ct), m, pc) &= false\\
\texttt{matchBeforeOtherExec}(\xi, aexecution(), m, pc) &= false
\end{aligned}$$

$$\texttt{matchBeforeOtherExec}(\xi, pcut, m, pc) = matchPcut(\xi, pcut, m, pc) \quad \texttt{iff} \begin{cases} pcut \in BasicPcut \\ pcut \neq mexecution(mp) \\ pcut \neq staticInit(ct) \\ pcut \neq aexecution() \end{cases}$$

4.6 matchPcut

`matchPcut` is called from the functions `matchBeforeOtherExecut`, `matchAfterPcut`. The pointcut argument is one of the the following base pointcuts: mcall(mp),get(fp), set(fp), withincode(mp) and within(ct).

$$\texttt{matchPcut} \models Environment \times Pointcut \times Method \times ProgramCounter \rightarrow Boolean$$

$$\texttt{matchPcut}(\xi, pcut, m, pc) = true \ \texttt{if}$$

$$① \begin{cases} pcut = mcall(mp); mp \in MethodPattern \\ m.code(pc) = \texttt{invokevirtual}|\texttt{invokestatic}|\texttt{invokeinterface}\ i, n|\texttt{invokespecial}\ i \\ \Gamma\xi = (\Gamma.javaEnvironment) \\ mPoolEntry = \Gamma\xi(m.fromClass).constantPool(i) \\ msign = mPoolEntry.methodSignature \\ calledm = \texttt{retrieveM}(msign, \Gamma\xi(mPoolEntry.supposedClass).methods) \\ m.code(pc) = invokespecial\ i) \Rightarrow (\texttt{isPrivate}(calledm) \wedge calledm.methodSignature.name <> "init") \\ \texttt{isMpatternMatched}(mp, calledm) \end{cases}$$

$$② \begin{cases} pcut = get(fp); fp \in FieldPattern \\ m.code(pc) = getfield\ i \vee m.code(pc) = getstatic\ i \\ \Gamma\xi = (\xi.javaEnvironment) \\ fPoolEntry = \Gamma\xi(m.fromClass).constantPool(i) \\ fsign = fPoolEntry.fieldSignagure \\ wedgegetF = \texttt{retriveF}(fsign, \Gamma\xi(fPoolEntry.supposedClass).fields) \\ \texttt{isFpatternMatched}(fp, getF) \end{cases}$$

$$③ \begin{cases} pcut = set(fp); fp \in FieldPattern \\ m.code(pc) = \texttt{putfield}\ i \vee m.code(pc) = \texttt{putstatic}\ i \\ \Gamma\xi = (\xi.javaEnvironment) \\ fPoolEntry = \Gamma\xi(m.fromClass).constantPool(i) \\ fsign = fPoolEntry.fieldSignature \\ setF = \texttt{retrieveF}(fsign, \Gamma\xi(fPoolEntry.supposedClass).fields) \\ \texttt{isFpatternMatched}(fp, setF) \end{cases}$$

$$④ \begin{cases} pcut = withincode(mp); mp \in MethodPattern \\ \texttt{isBeforeOrAfterShadow}(m, pc) \\ \texttt{isMpatternMatched}(mp, m) \end{cases}$$

$$⑤ \begin{cases} pcut = within(ct); ct \in ComponentType \\ \texttt{isBeforeOrAfterShadow}(m, pc) \\ m.fromClass = ct \end{cases}$$

4.7 Advice Applicable to Insert

- `isBeforeAdviceApplicable` returns true if the given `before` advice is applicable in the method m at the program counter pc. We distinguish between the case of *execution* shadow and the other shadows.

  $$\texttt{isBeforeAdviceApplicable} : \ Environment \times Method \times ProgramCounter \times AdviceInfo \rightarrow Boolean$$

  $$\begin{aligned} \texttt{isBeforeAdviceApplicable}(\Gamma, m, pc, ad) = & true\ \texttt{iff} \\ & (m.Code(pc) = \texttt{impdep1} \wedge \texttt{matchBeforeExec}(ad.pointcut, m)) \\ & |\texttt{matchBeforeOtherExec}(\xi, ad.pointcut, m, pc) \end{aligned}$$

- `isAfterAdviseApplicable` returns true if the given `after` advice is applicable in the method m at the program counter pc.

  $$\texttt{isAfterAdviseApplicable} : \ Environment \times Method \times ProgramCounter \times AdviceInfo \rightarrow Boolean$$

  $$\texttt{isAfterAdviseApplicable}(\xi, m, pc, ad) = true\ \texttt{if} \begin{cases} \texttt{isAfterShadow}(m, pc) \\ \texttt{matchAfterPcut}(\xi, ad.pointcut, m, pc) \end{cases}$$

- `insertBeforeAdvice` takes an environment, a method, a program counter,and an advice as arguments and returns a new environment, a new method, and a new program counter after injecting the advice in the method.*cpool1* refers to the new constant pool added by the advice;incase there is mathes added to the advice methods,`cpool2` is also added to define the advice's advice(This may lead to endless loop if the advice is not defined properly).The `changeMethods` map all possible methods and advice into a new set of methods.$pc' = pc + 2$ since `invokestatic` and `invokestatic` consumes two instructions.Hence a new environment ξ' is returned.

$$\begin{aligned} \texttt{insertBeforeAdvice} \quad : \quad & Environment \times Method \times ProgramCounter \times AdviceInfo \\ \rightarrow \quad & Environment \times method \times ProgramCounter \times ProgramCounter \end{aligned}$$

$$\texttt{insertBeforeAdvice}(\xi, m, pc, ad) = (\xi', m', pc')\ \texttt{if}$$

$$\begin{cases} \Gamma\xi = \xi.javaEnvironment \\ c = \Gamma\xi(m.fromClass) \\ cpool = c.constantPool \\ ms = \texttt{signatureAspectOf}(ad) \\ cpool1 = cpool[i \mapsto \texttt{newPoolEntry}(ms, ad.fromClass)], i \notin Dom(cpool) \\ cpool2 = cpool1[j \mapsto \texttt{newPoolentry}(ad.adviceSignature, ad.fromClass)], j \notin Dom(cpool1) \\ code1 = m.code[k+2 \mapsto m.code(k)], \forall k \in Dom(m.code) \\ code2 = code1[pc+1 \mapsto \texttt{invokestatic}\ i] \\ code3 = code2[pc+2 \mapsto \texttt{invokevirtual}\ j] \\ m' = m[code \leftarrow code3] \\ c1 = c[constantPool \leftarrow cpool2, methods \leftarrow \texttt{ChangeMethods}(c1.methods, m, m')] \\ \Gamma\xi_1 = \Gamma\xi[m.fromClass \mapsto c1] \\ pc' = pc + 2 \\ \xi = \xi[javaEnvironment \leftarrow \Gamma\xi_1] \end{cases}$$

- `insertAfterAdvice` also takes *nextpc* into consideration since it always forward 2 instructions during the advice. There are 2 options for this insertion as with whether *pc* reaches the end of the method.

$$\begin{aligned} \texttt{insertAfterAdvice} \quad : \quad & Environment \times Method \times ProgramCounter \times AdviceInfo \times ProgramCounter \\ \rightarrow \quad & Environment \times method \times ProgramCounter \times ProgramCounter \end{aligned}$$

$$\texttt{insertAfterAdvice}(\xi, m, pc, ad, nextpc) = (\xi', m', pc', nextpc')\ \texttt{iff}$$

$$① \begin{cases} \neg\texttt{isReturn}(m, pc) \\ \Gamma\xi = \xi.javaEnvironment \\ c = \Gamma\xi(m.fromClass) \\ cpool = c.constantPool \\ ms = \texttt{signatureAspectOf}(ad) \\ cpool1 = cpool[i \mapsto \texttt{newPoolEntry}(ms, ad.fromClass)], i \notin Dom(cpool) \\ cpool2 = cpool1[j \mapsto \texttt{newPoolentry}(ad.adviceSignature, ad.fromClass)], j \notin Dom(cpool1) \\ code1 = m.code[k+2 \mapsto m.code(k)], \forall k \in Dom(m.code) \\ code2 = code1[pc+1 \mapsto \texttt{invokestatic}\ i] \\ code3 = code2[pc+2 \mapsto \texttt{invokevirtual}\ j] \\ m' = m[code \leftarrow code3] \\ c1 = c[constantPool \leftarrow cpool2, methods \leftarrow \texttt{changeMethods}(c1.methods, m, m')] \\ \Gamma\xi_1 = \Gamma\xi[m.fromClass \mapsto c1] \\ pc' = pc \\ nextpc' = nextpc + 2 \\ \xi = \xi[javaEnvironment \leftarrow \Gamma\xi_1] \end{cases}$$

$$② \begin{cases} \texttt{isReturn}(m, pc) \\ \wedge(\xi', m', pc') = \texttt{insertBefore}(\xi, m, pc, ad) \\ \wedge nextpc' = nextpc + 2 \end{cases}$$

5 Weaving Semantics

With the help of utility functions described above,We conclude 5 rules in the operational semantics of static pointcuts based on aspectj and JVMLI.

5.1 Neither Advice Applicable

This rule of the semantics describes the case where the instruction in the method m at the program counter pc is not a shadow or the advice list is empty(all the advices have been treated for this instruction). In such cases, the current instruction is skipped and the list of advices is reset to its initial value (all the advices of the environment).

$$\frac{(\texttt{isBeforeShadow}(m, pc) \wedge \texttt{isAfterShadow}(m, pc)) \vee ads = [\,]}{\langle \xi, m, pc, ads, nextpc \rangle \longrightarrow \langle \xi, m, \xi.advices, nextpc + 1 \rangle}$$

5.2 Before Advice not Applicable

This rule fires in the case where the head of the advice list is a Before advice but is not applicable to the current instruction. This advice is then re-moved and we reconsider the possibility of weaving the same instruction with the remaining list of advices.

$$\frac{\texttt{head(ads).kind=before} \wedge \neg\texttt{isBeforeAdviceApplicable}(\xi, m, pc, \texttt{head}(ads))}{\langle \xi, m, pc, ads, nextpc \rangle \longrightarrow \langle \xi, m, tail(ads), nextpc \rangle}$$

5.3 Before Advice Applicable

This rule of the semantics represents the case where the head of the advice list is a Before advice and is applicable to the current instruction. In this case, the method and the environment are changed because of the Before advice merging. The program counter of the current instruction will change also because of the Before advice injection.Denote

$beforeApplicable := \texttt{head(ads).kind=before} \wedge \texttt{isBeforeAdviceApplicable}(\xi, m, pc, \texttt{head}(ads))$

$$\frac{beforeApplicable \wedge (\xi', m', pc') = \texttt{insertBeforeAdvice}(\xi, m, pc, \texttt{head}(ads))}{\langle \xi, m, pc, ads, nextpc \rangle \longrightarrow \langle \xi', m', pc', \texttt{tail}(ads), pc' + 1}$$

5.4 After Advice not Applicable

This rule depicts the case where the head of the advice list is an After advice but is not applicable to the current instruction. The head of the advice list is then removed and we reconsider the possibility of weaving with the remaining list of advices.

$$\frac{\texttt{head(ads).kind=after} \wedge \neg\texttt{isAfterAdviceApplicable}(\xi, m, pc, \texttt{head}(ads))}{\langle \xi, m, pc, ads, nextpc \rangle \longrightarrow \langle \xi, m, tail(ads), nextpc \rangle}$$

5.5 Before Advice Applicable

This rule of the semantics represents the case where the head of the advice list is an After advice and is applicable to the current instruction. In this case, the method and the environment are changed because of the After advice merging. The program counters of the current instruction and next instruction will change also because of the After advice injection.

$afterApplicable := \texttt{head(ads).kind=after} \wedge \texttt{isAfterAdviceApplicable}(\xi, m, pc, \texttt{head}(ads))$

$$\frac{afterApplicable \wedge (\xi', m', pc') = \texttt{insertAfterAdvice}(\xi, m, pc, \texttt{head}(ads))}{\langle \xi, m, pc, ads, nextpc \rangle \longrightarrow \langle \xi', m', pc', \texttt{tail}(ads), pc' + 1}$$

6 Conclusions

This review discusses about the operational semantics of aspectj pointcuts with a reduced JVMLI set.It divides the procedure of weaving into two parts:the front-end and the back-end.We view the aspect woven as a normal java byte code file and analyze the mapping from the mixed environment of `javaEnvironment` and `aspect` into pure javaEnvironment.It also induces five rules related to the two kinds of advice.

References

[1] N. Belblidia and M. Debbabi. Towards a formal semantics for aspectj weaving. In *JMLC'06*, pages 155–171, 2006.

[2] M. K. Gupt1a. *Mastering AspectJ Aspect Oriented Programming in Java*. Wiley, 2003.

[3] Oracle. The Java Virtual Machine Specification. `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`, June 2012.

[4] Wikipedia. Aspect-oriented programming. `http://en.wikipedia.org/wiki/Aspect-oriented_programming`, September 2012.