A New Kind of Advice for Advice Composition without Interference

干渉のないアドバイスの合成のための新しいアドバイス

by

Fuminobu Takeyama

武山 文信

08M37195

February 26, 2010

A Master's Thesis Submitted to
Department of Mathematical and Computing Sciences
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

In Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Supervisor: Shigeru Chiba

Abstract

Aspect composition often involves aspect interference and this is a crucial problem in aspect oriented programming. When multiple advices are woven at the same join point, the advices often interfere with each other. In this thesis, we handle this interference called *advice interference*.

Since advices are executed one by one in the order of precedence, giving appropriate precedence order among the conflicting advices is a typical solution of advice interference. In AspectJ, declare precedence is provided to control this. More powerful mechanisms that allow fine-grained control have been proposed.

However some combinations of advices have no correct precedence order with which the resulting behavior is acceptable. To resolve the interference among them requires additional implementation of their composition but there is no way to separate the implementation from them. In AspectJ, the composition have to be implemented in each conflicting advices, although they modularize their own concerns.

In this thesis, we propose a novel language extension of AspectJ named *Airia* to address this problem. It provides a new kind of around advice named a *resolver* for resolving interference. Since it is invoked only at the join points when given advices conflict with each other, it overrides the advices and performs merged behavior of them. The resolvers can call an extended version of proceed, which takes as an argument the precedence order among those conflicting advices for reuse; unnecessary advices can be removed. Then not only in the higher precedence than the conflicting advices, an advice for the composition can be invoked by resolver at intermediate precedence. This advice is called a helper advice. Furthermore, the resolvers are composable. They can be used to resolve interference among other resolvers and advices; so we can implement composition of advices in hierarchical manner. We have implemented Airia extending the AspectBench compiler with JastAdd.

Acknowledgments

I would like to express my gratitude to my supervisor, Shigeru Chiba. He took a lot of time to discuss with me every week. The discussion greatly improved this work. He helped me write papers although he was very busy.

I also thank all the members of Chiba Shigeru Group. In particular, Michihiro Horie suggested me what should I do when I started my researches on aspect oriented programming. Shumpei Akai, Takeshi Asumi, and Satoshi Morita have told me about various things including our group, our campus, and our area of research since I came to Tokyo Institute of Technology.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software is composed with various units. A code is generally modularized into some sort of units. One of the purposes of modularization is code reuse, which is a key issues in software engineering. It increases quality of software and decrease cost of development. Programmers reuse the code by the unit. Since bugs of units are removed if they are repeatedly reused, code reuse leads to improvements in quality of applications which use them. Traditionally a unit is a set of functions, which is exported as a library, such as a shared object in UNIX-like operating systems, and large number of modularization techniques including aspect-oriented programming have been proposed. Nowadays object-oriented programming is widely used practically and a code is modularized into classes.

Ideally an application should be composed only by gathering the units without the knowledge about the detailed implementation about the units. However, it is currently infeasible because composition causes conflict and interference. In order to compose an application reusing the units, current approach provides ability to combine units manually to users of the units. For example, in traits inheritance [18], a conflict of methods provided from different traits is manually resolved by users of them. In this thesis, we contribute the improvement of reusability of units in aspect-oriented programming.

1.1 A problem with advice compostion

In aspect-oriented programming, crosscutting concerns are modularized into aspects and an application is composed with classes and aspects. Since programmers can extend an application only by implementing new aspects, it

allows to reuse the whole system of it. For example, we assume an interpreter and extend it by adding new aspects. If problems of the original interpreter is fixed, the modification can be applied to extended intepreter. We can also implement a new interpreter with a combination of features by selecting existing aspects which implemets them.

Composing a new aspect by combining existing aspects is not easy. The aspects often conflict with each other and cause undesirable interference. This problem is called *aspect interference*. In particular, resolving interference that is caused when multiple advices are woven at the same join point is a serious issue and it has been investigated in the research community. This paper addresses this issue called *advice interference*.

A typical solution is to allow programmers to control the precedence order among conflicting advices and linearize them; the advices are executed one by one in the precedence order. For example, AspectJ [13] provides declare precedence to control this statically. Other approaches for more fine-grained control such as depending on dynamic context have been proposed.

However, for some combinations of advices, there is no correct precedence order with which the composed behavior is acceptable. Such combination needs modifying the bodies of the conflicting advices to explicitly implement the merged behavior. This is not desirable since the programmers have to be aware of the composition when they write an individual advice. Furthermore, the composition itself is a crosscutting concern. The implementation of the composition should be described separately from the conflicting advices.

1.2 A Solution for advice composition without interference

In this thesis, we address this problem by extending AspectJ. This novel language extension is named *Airia*. In this language, a new kind of around advices called *resolvers* are available. A resolver is used to implement the composition of conflicting advices. It is invoked at only the join points when the given set of advices conflicts with each other. Since a resolver has higher precedence than those conflicting advices, it overrides the implementation of those advices.

In the body of the resolver, a proceed call takes as an argument the precedence order among those conflicting advices. The unnecessary advices in conflicting advices can be removed at that call. It can thereby control the execution order of the remaining advices. In our language Airia, therefore,

declare precedence is not available. Then not only in the higher precedence than the conflicting advices, an advice for the composition can be invoked by resolver at intermediate precedence. This advice is called a helper advice.

Furthermore, a resolver is composable. It can implement the composition among other resolvers and advices; advices can be combined in a hierarchical manner with existing composition reused. This can be achieved because a resolver is a special around advice and still a base language construct. The total execution order is determined statically and the precedence specified by resolvers are examined by our compiler.

1.3 Contribution

In summary, the contribution of this paper is as the following:

- We find advices that cannot be combined by giving precedence order. To combine correctly, these advices require composition code.
- We propose a new kind of advice named a resolver for advice composition.
- A resolver enables separation of the implementation of advice composition while keeping sufficient expressive power.
- A resolver is composable and thus we can implement composition of several advices in a hierarchical manner.
- We restrict the semantics of resolver so that the execution order of conflicting advices can be determined statically. The precedence order is examined at compile time.
- Our language Airia has been implemented as an extension of the AspectBench compiler [5]. Its frontend is developed with JastAdd [9].

1.4 The structure of this thesis

The rest of this theses is organized as follows. In Chapter 2 we mention AspectJ and related works. Chapter 3 shows an example of advice interference in a simple interpreter. The interference is not resolved by giving precedence order. Then we mention a problem of implementing composition in AspectJ. Chapter 4 presents the design of our language Airia. We also resolve the interference shown in Chapter 3 in Airia. In Chapter 5 we

describe how the total execution order of advice is caluculated. Chapter 6 shows other examples of interference and its resolution. Chapter 7 mentions an overview of implementation of Airia and Chapter 8 concludes this paper.

Chapter 2

Background

2.1 AspectJ

AspectJ is one of aspect oriented language based on Java. In aspect-oriented programming, crosscutting concerns are separated from base code, *i.e.* methods in classes, into aspects. An application consists of classes and aspects.

In AspectJ, this is achieved by *advices* and *pointcuts*. An advice implements the behavior of the concerns. It is a construct like a method but it is not invoked explicitly. Each advice has a pointcut. A pointcut selects points in the execution of program, which is called *join points*. When execution of the application reaches the join point, the advice is invoked implicitly. Advices which implement the same concern compose an *aspect*. In this way, the base code is not aware of the separated concern.

Some pointcuts selects join points with dynamic context. For example, the if pointcut selects join points when its condition is true. The this pointcut checks the actual type of this object. It selects when the type is a subtype of specified one.

An around advice is one kind of advice in AspectJ. It overrides the computation at the join point which its pointcut selects. In its body, a special method proceed can be called. It takes same parameters as the advice and executes the original computation overridden by the advice. It returns a value that the original computation returns.

In this thesis, we explain our approach mainly with around advice. Other kinds of advice, such as before and after, can be considered in the same manner as around advices because they are regarded as syntax sugars of around advices. For example, the next before advice:

```
before(): execution(String C.m(String)) {
   System.out.println("Executing C.m.");
}
```

can be rewritten as the following:

```
Object around(): execution(String C.m(String)) {
   System.out.println("Executing C.m.");
   return proceed();
}
```

2.2 Related works

Resolving aspect interference for aspect composition is a classic research topic and hence there have been a number of proposals.

2.2.1 Stateful aspect

Douence *et al.* proposed an approach for detecting and resolving conflicts between aspects on their formal framework, Stateful Aspect [7, 8]. In the frame work, aspects are combined with a parallel operator. Their approach is making the operator extensible so that the operator will generate correctly merged behavior when the operands are conflicting with each other. Although this approach is similar to ours, we provide a single composition operator (*i.e.* a resolver) but we do not make the semantics of operator extensible. We also propose an extension of AspectJ based on our approach.

2.2.2 Meta programming for advice composition

Most previous approaches are categorized into meta programming. POPART [6] provides a meta-aspect protocol. Advice composition can be dynamically customized by an instance of MetaAspectManager. Programmers can define an appropriate MetaAspectManager to implement a custom composition policy for resolving conflicts among particular advices. The conflicting advices at a join point is contained in objects that implements the Collection interface and programmers can order them with its methods such as sort. JAsCo [19] also provides a mechanism like this.

Meta programming is often complicated and difficult from its power. Our approach does not need meta programming. Since a resolver is a special around advice, it is still a base-level language construct.

The meta programming provided by the languages above concentrates in changing precedence order but in OARTA [16], the ability for meta programming is restricted to modifying only the pointcut of another advice. It is common to our approach to separate composition into an aspect. For resolving aspect interference, OARTA provides declare pcdisjunct/pcconjunct which replace the pointcut of an advice with the disjunct/conjunct of the original pointcut and specified one. It also allows to extend an existing advice by its extended adviceexecution pointcut which selects execution of the specified advice.

However, declare pcdisjunct/pcconjunct lacks composability. Since it is permanent and can not be removed by some constructs, when composition codes interfere with each other, programmers must remove whole the aspect containing it. For advice interference, only when advices are conflict, the behavior of the advices should be changed. Our resolver is composable; interference of resolvers can be resolved by another resolver. A resolver is executed at the join point when specified advices conflict.

2.2.3 Context-Aware Composition Rules

Context-Aware Composition Rules [14, 15] allows programmers to control precedence order among advices for every join point with a new pointcut advices, which selects join points when advices conflict like our resolver. Since dynamic pointcut is supported, it can control precedence dynamically. It also allows removing an existing advice at the join points. Programmers write only declarative rules such as Prec, First, and Ignore, in a construct similar to an advice named rule. Statements of Java can not be described there. The next code specifies that A has higher precedence than B and removes C when advice A, B, and C conflict:

```
aspect Composition {
   declare rule precedence: advices(A, B, C) {
      Prec(A, B);
      Ignore(C);
   }
}
```

However, as we mentioned in Section 3, some kinds of advice interference cannot be resolved by only reordering advices. On the other hand, our language Airia also provides the ability for adding a new advice when advices conflict.

2.2.4 Reflex

Reflex [20] is an infrastructure for building an aspect system. It provides an application-programming interface (API) for implementing a new policy for advice composition. Programmers can exploit this API for customizing the aspect system to resolve conflicts among particular advices. For example, it supports an implicit cut to share pointcut of the specified advice and the CompositionOperator class for ordering.

Our language Airia provides a base-level language construct for resolving conflicts. The users of Airia do not have to consider the implementation of the language.

2.2.5 Aspect refinement

Aspect refinement and mixin-based aspect inheritance [3, 4] enable programmers to incrementally extend the behavior of an existing advice. JastAdd also supports refinement.

Aspect refinement does not take into consideration extending multiple advices at once. If programmers refine each conflicting advice for composition, composition can be separated but they have to design a protocol to work well with another advice through a proceed call. This is mentioned in Section 3.2. On the other hand, our language Airia enables extending the behavior of a combination of multiple advices.

2.2.6 Interference in object-oriented programming

Interference is common issue in object oriented programming. The relationship between an around advice and the original computation at the join point is similar to the relationship between a mixin and a class. A proceed call corresponds to a super call. The former executes the next advice or the original computation while the latter executes the method in the next mixin or class. While advice interference is a problem, interference among mixins given to the same class is also a problem.

Traits [18] is a solution for mixin interference. For example, two traits provide methods named equals. When a class uses these traits, the two methods conflict. If the composed equals method returns true when both of the conflicting methods return true, then programmers rename the conflicting methods equals1 and equals2 and a value of evaluating equals1() && equals2() is returned. Note that a construct for renaming is provided.

Our approach can be regarded as an application of the idea of traits to aspects. Both approaches allow programmers to define a new advice/method

for overriding advices/methods and resolving their conflicts.

Chapter 3

A Motivating Example

We first show an example of aspect interference that is not resolved in existing approaches.

3.1 A simple interpreter

We present a simple interpreter with a binary operator +, which is written in AspectJ. Listings 3.1 shows classes representing AST (Abstract Syntax Tree) nodes. The Plus class expresses a binary operator +. It has two fields, left and right, representing its operands and it extends the Expression class. We declare a method for evaluating an AST in the EvaluationAspect aspect. Since our interpreter currently does not support any data types, this aspect appends an empty eval method to the Expression class by an inter-type declaration.

We then extend the interpreter to support integer values. We do not have to modify the existing classes. We have only to write a new aspect shown in Listing 3.2. The around advice in the IntegerAspect aspect is invoked when the Plus.eval method is executed; it returns a summation of the two operand. The following code makes an AST representing `1 + 2`. When e.eval is executed on this tree, it returns 3.

```
Expression e =
   new Plus(new Constant(1), new Constant(2));
```

Next, we extend the original interpreter to support character strings. Again, we do not have to modify the existing classes. We implement this extension by the StringAspect in Listing 3.3. Since the operator + now represents concatenation of character strings, the around advice implements

```
public class Expression extends ASTNode { /*...*/ }

public class Plus extends Expression {
    private Expression left;
    private Expression right;

    public Plus(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    public Expression getLeft() {
        return left;
    }

    public Expression getRight() {
        return right;
    }
    // :
}

public class Constant extends Expression {
    Object value;

    public Constant(Object value) {
        this.value = value;
    }
}

aspect Evaluation {
    public Object Expression.eval() {
        return null;
    }

    public Object Constant.eval() {
        return value;
    }
}
```

Listing 3.1: Classes representing the AST

```
aspect IntegerAspect {
    Object around(Plus t):
            target(t) && execution(Object Plus.eval()) {
        return (Integer)t.getLeft().eval() +
                (Integer)t.getRight().eval();
    }
}
```

Listing 3.2: An aspect for integer values

```
aspect StringAspect {
    Object around(Plus t):
            target(t) && execution(Object Plus.eval()) {
        return t.getLeft().eval().toString() +
                t.getRight().eval().toString();
    }
}
```

Listing 3.3: An aspect for character strings

the behavior of the eval method. If we compile the classes and the aspects in Listing 3.1 and 3.3, then the resulting interpreter will correctly handle character strings.

The last step is to build an interpreter supporting both integers and character strings. Some readers might expect that we could easily obtain the interpreter if the two aspects IntegerAspect and StringAspect are compiled and woven together. However, these two aspects conflict with each other, *i.e.* multiple advices are woven at the same join point. The resulting behavior of the eval method is different from our naive expectation. This paper deals with this unexpected behavior of the combined advices, which we call *advice interference*. It is one kind of *aspect* interference; other kinds of aspect interference such as [12] are out of the scope of this paper.

If the two advices are combined, the resulting eval method cannot process all acceptable ASTs. In AspectJ, when the eval method is called, the advice with the highest precedence is executed. Suppose that IntegerAspect has the highest. Then the eval method does not process the AST e constructed by the following code:

```
Expression e = new Plus(
    new Constant("Hello "), new Constant("world!"));
```

It will throw ClassCastException since the operands are character strings but

the around advice in IntegerAspect assumes that the operands are integer values. Changing the precedence order does not solve this problem. In AspectJ, we can explicitly specify precedence. For example,

```
declare precedence: StringAspect, IntegerAspect;
```

this declaration specifies that StringAspect has higher precedence than IntegerAspect. The eval method now returns an unexpected value when the AST e2 shown below is evaluated:

```
Expression e2 =
   new Plus(new Constant(1), new Constant(2));
```

The returned value will be a character string "12" although both operands are integer values.

3.2 An incomplete solution in AspectJ

A partial solution of the problem above is to make advices composable by *linearization*. Since AspectJ provides proceed calls, we can reimplement advices and connect them by proceed to make a single chain. If proceed is called in an advice body, it invokes the advice with the next highest precedence in the chain. If there is no other advice, the original computation at the join point is executed.

Listing 3.4 shows the result of the reimplementation for the linearization. The resulting code can be regarded as an AspectJ version of the chain of responsibility pattern. These advices are implemented as possible as they can be combined the advices for other data-types. Now the two around advices call proceed to invoke the next advice when they cannot deal with the given operands. Although the combination of some advices require an explicit declaration of the precedence order for linearization, the two aspects in Listing 3.4 do not require it; they correctly work under any precedence order. The interpreter containing these two aspects can deal with the AST constructed by this code:

```
Expression e =
   new Plus(new Constant("Str"), new Constant("1"));
```

However, this solution is not satisfactory from the software engineering viewpoint. The implementation of each advice body must be aware of the composition with other (maybe unknown yet) advices by the linearization. Programmers have to design a protocol for the advice chain before implementing each advice body. The protocol design is not easy; the advices must

```
aspect IntegerAspect {
    Object around(Plus t):
            target(t) && execution(Object Plus.eval()) {
        Object lvalue = t.getLeft().eval();
        Object rvalue = t.getRight().eval();
        if (lvalue instanceof Integer && rvalue instanceof Integer) {
            // not for composition
            return (Integer)lvalue + (Integer)rvalue;
        } else {
            return proceed(t);
        }
    }
}

aspect StringAspect {
    Object around(Plus t):
            target(t) && execution(Object Plus.eval()) {
        Object lvalue = t.getLeft().eval();
        Object rvalue = t.getRight().eval();
        if (lvalue instanceof String || rvalue instanceof String) {
            // not for composition
            return lvalue.toString() + rvalue.toString();
        } else {
            return proceed(t);
        }
    }
}
```

Listing 3.4: A composable version of the conflicting advices

be able to correctly work with and without other advices. Furthermore, the composition itself is a cross-cutting concern. Note that most statements in Listing 3.4 are for the composition by the linearization. Only the two return statements marked by a comment implement the behavior of the eval method in the Plus class. The composition code scatters over the two advices.

Chapter 4

Resolving interference in Airia

To resolve the problem mentioned above, we propose a novel language extension of AspectJ. This new language named *Airia* allows programmers to separately describe how to resolve advice interference. Instead of the conflicting advices themselves, the resolving code is described in a new kind of around advice called a *resolver*. Hence the implementation of each conflicting advice is independent of the other conflicting advices and their composition protocol.

Listing 4.1 is an example of aspect including a resolver. It resolves the advice interference presented in the previous section. Details of this resolver are mentioned below. Since the resolving code is separated into this resolver, the conflicting advices do not include the code for the composition or resolving the interference. See Listing 4.2, which presents the three conflicting advices written in our language. They are simpler than the composable version of the aspects shown in Listing 3.4. They are the same as the original aspects in Listing 3.2 and 3.3 except that every advice has a unique name. These advice names are used by the resolver.

A resolver is composable. Programmers can write a resolver that resolves interference among other resolvers and normal advices. Suppose that we write a new aspect EvaluationCacheAspect and its advice causes interference with the advices of IntegerAspect and StringAspect. For these three conflicting advices, we can write a new resolver by reusing the existing resolver of IntegerStringAspect. Since the resolver of IntegerStringAspect deals with the advice interference between IntegerAspect and StringAspect, the new resolver will be declared to deal with the interference between the resolver

```
aspect IntegerStringAspect {
    Object resolver plusEvalIntStr(Plus t)
                and(IntegerAspect.plusEvalInt(t),
                    StringAspect.plusEvalStr) {
        Object lvalue = t.getLeft().eval();
        Object rvalue = t.getRight().eval();
        if (lvalue instanceof String || rvalue instanceof String) {
            return [StringAspect.plusEvalStr].proceed(t);
        } else if (lvalue instanceof Integer &&
                    rvalue instanceof Integer) {
            return [IntegerAspect.plusEvalInt].proceed(t);
        } else {
            throw new RuntimeException();
        }
    }
}
```

Listing 4.1: The aspect for combining IntegerAspect and StringAspect

of IntegerStringAspect and the new advice of EvaluationCacheAspect. The implementation of that new resolver will call proceed to execute the resolver of IntegerStringAspect.

4.1 A Resolver

A resolver is a special kind of around advice, which is declared with a keyword resolver instead of around. The syntax of resolver declaration is the following:

```
[Modifiers] RetrunType resolver
    ResolverName(ArgumentType ArgumentName, ...)
    and|or(ConflictingAdviceName[(ArgumentName, ...)], ...)
    [uses HelperAdviceName, ...]
    [throws ExceptionType, ...]
{ Body }
```

The resolver keyword is followed by a resolver name. A parameter list to the resolver follows the resolver name if any. Unlike normal advices in AspectJ, it does not take a pointcut but it takes an and/or clause, which specifies a list of potentially conflicting advices. The resolver is expected to resolve interference among these advices.

Except the resolver keyword, its name, and the and/or clause, a resolver is the same as an around advice. The return type of a resolver is Object if

```
aspect Evaluation {
    public Object Expression.eval() {
        return null;
    }
}

aspect IntegerAspect {
    Object around plusEvalInt(Plus t):
            target(t) && execution(Object Plus.eval()) {
        return (Integer)t.getLeft().eval() +
                (Integer)t.getRight().eval();
    }
}

aspect StringAspect {
    Object around plusEvalStr(Plus t):
            target(t) && execution(Object Plus.eval()) {
        return t.getLeft().eval().toString() +
                t.getRight().eval().toString();
    }
}
```

Listing 4.2: The aspects written in our language

the join points bound to the resolver have different return types. The body of the resolver may include a proceed call.

In Listing 4.1, a resolver is named plusEvalIntStr and takes an and clause, which lists the names of the around advices in the two aspects IntegerAspect and StringAspect. Note that an advice also has a unique name. See Listing 4.2. The IntegerAspect aspect has an advice named plusEvalInt and the StringAspect has an advice named plusEvalStr. IntegerAspect.plusEvalInt and StringAspect.plusEvalStr are their fully-qualified names.

The join points when a resolver is executed are specified by an and/or clause. Since the resolver in Listing 4.1 has an and clause, it is executed at the join points that all the given advices are bound to, that is, when the eval method in the Plus class is executed. Note that those advices of the two aspects IntegerAspect and StringAspect conflict at those join points. A resolver has higher precedence than the advices specified by its and/or clause. Hence, it *overrides* all the conflicting advices at the join points. In our example, when the eval method in the Plus class is called, the body of the resolver is executed first.

The advices given to the and/or clause of a resolver work as pointcuts. Thus, a resolver can take parameters and pass them to those advices. For example, the resolver in Listing 4.1 takes a parameter t and passes it to the advice in the IntegerAspect. The parameter t is bound to the value that this advice binds its parameter to, that is, the target object of the call to the eval method.

A resolver may have an or clause. This specifies that the resolver is executed at the join points that at least one advice given to the or clause is bound to. For example, the next resolver is executed at the join points that only the two advices A and B are bound to but C is not:

```
Object resolver precedence() or(A, B, C) {
   return [A, B, C].proceed();
}
```

The or clause can be used for specifying a precedence order among advices as we do with declare precedence in AspectJ. The resolver shown above specifies that the precedence order is A, B, and C. `[A, B, C].proceed()` executes the three advices in that order (we below mention this proceed call again).

We introduced an or clause for reducing the number of necessary resolvers. If we could not use an or clause, we would have to define many resolvers for all possible combinations of potentially conflicting advices. Suppose that we have three advices A, B, and C. If we use only and clauses, we must define resolvers for every combination: A and B, B and C, C and A, and all the three, if they conflict at different join points. Since we expect that most combinations share the same advice body, using an or clause will reduce the number of necessary resolvers.

To be precise, the join points selected by an and/or clause are the intersection/union of the join point *shadow* [17] selected for the advices given to that and/or clause, respectively. Dynamic pointcuts such as cflow and target are ignored. Thus, a resolver may be executed at the join points that the advices in its and/or clause are not bound to.

4.2 A Proceed call with precedence

Like a normal advice, a resolver can call proceed to invoke another advice with the next highest precedence. The proceed call from a resolver explicitly specifies the precedence order of the advices given to the and/or clause, which will be invoked by the proceed call. Note that unlike AspectJ our language Airia does not provide declare precedence. The precedence order is described between brackets preceding .proceed.

Suppose that there are two advices A and B, which conflict at the join point selected by a pointcut pc(). We assume that there is no other advices. Then a resolver AorB can call proceed twice with different precedence order:

```
void resolver AorB() or(A, B) {
   [A, B].proceed();
   [B, A].proceed();
}

pointcut pc(): ...;

void around A(): pc() {
   proceed();
}

void around B(): pc() {
   proceed();
}
```

When [A, B].proceed() is called, the advice A is invoked. The advice B is invoked by the proceed call in the advice A. The proceed call in the advice B execute the original computation at the join point. On the other hand, when [B, A].proceed() is called, the advice B is invoked. The advice A is the next.

Note that [A, B].proceed() does not mean that the advice A and then B is executed. It means that the advice A has higher precedence than B; if there was the advice C shown below, it might be invoked by the proceed calls in AorB:

```
void around C(): pc() {
   proceed();
}
```

A or B may not be executed when their pointcuts do not match the current join point.

The proceed call can remove advices from the set of the remaining advices, which will be invoked by later proceed calls. If the advice list between brackets does not include an advice given to the and/or clause, the advice is removed. In Listing 4.1, both proceed calls remove one advice. The former removes IntegerAspect.plusEvalInt and the other removes StringAspect.plusEvalStr. For example, [IntegerAspect.plusEvalInt].proceed() invokes the plusEvalInt advice in IntegerAspect and then, if it calls proceed again, the original eval method is invoked. The plusEvalStr advice in StringAspect is never invoked.

```
aspect EvaluationCacheAspect {
    Object Expression.cachedValue;
    boolean Expression.isChanged = false;
    void around plusEvalCache(Expression t):
            execution(Object Plus.eval()) && args(t) {
        if (t.isChanged) {
            cachedValue = proceed(t);
            isChanged = false;
        }
        return cachedValue;
    }
    after changed(): ... {
        isChanged = true;
    }
}
```

Listing 4.3: The EvaluationCacheAspect

```
aspect IntegerStringCacheAspect {
    Object resolver evalIntStrCache():
            and(IntegerStringAspect.evalIntStr,
                EvaluationCacheAspect.plusEvalCache)
        return [EvaluationCacheAspect.plusEvalCache,
                IntegerStringAspect.evalIntStr].proceed();
    }
}
```

Listing 4.4: A resolver resolving conflicts between a normal advice and another resolver

4.3 Composability of resolvers

A resolver, which is a special around advice, may also conflict with other resolvers or normal advices. This conflict can be also resolved by another resolver; a resolver is composable. An advice given to an and/or clause may be a resolver. A proceed call with precedence specifies precedence order among conflicting advices and/or resolvers.

Let us consider a new advice shown in Listing 4.3. The join point of this advice is the execution of the eval method. Thus, this advice conflicts with the two advices in IntegerAspect and StringAspect shown in Listing 3.2 and 3.3. Since the conflict between these two advices has been already resolved by the resolver in IntegerStringAspect, we reuse this resolver to resolve the

conflicts among the new advice and these two advices. See Listing 4.4. This resolver in IntegerStringCacheAspect has an and clause, which lists the new advice in EvaluationCacheAspect and the resolver in IntegerStringAspect. It resolves conflicts between the advice and the resolver.

The behavior of a resolver for another resolver is the same as normal resolvers. When the eval method is called, this resolver in IntegerStringCacheAspect is invoked first since it has higher precedence than the other advices and resolver. When this resolver calls proceed with precedence, the advice with the next highest precedence is executed, which is the advice in EvaluationCacheAspect. After that if the advice calls proceed, the resolver in IntegerStringAspect is executed. Note that this resolver does not explicitly describe how the conflicts between IntegerAspect and StringAspect are resolved. It is encapsulated in the resolver of IntegerStringAspect. The composition of IntegerStringCacheAspect is hierarchical.

A resolver for another resolver can remove not only an advice but also a resolver given to its and/or clause. When the resolver in IntegerStringCacheAspect calls proceed, it could remove the resolver in IntegerStringAspect if necessary. For example, if [EvaluationCacheAspect.plusEvalCache].proceed() is called, then the remaining advices are only the normal advices in EvaluationCacheAspect, StringAspect, and IntegerAspect. The resolver in IntegerStringAspect is not included any longer.

Unlike declare precedence in AspectJ, a resolver can flexibly modify precedence order among conflicting advices even during runtime by a proceed call with precedence. Thus, declare precedence is not available in our language Airia. The precedence order must be explicitly specified; there is no default precedence order unlike AspectJ.

4.4 A helper advice

A resolver can add a new advice for helping composition. Since a resolver has higher precedence than conflicting advices, the added advice is normally given intermediate precedence among those conflicting advices.

Suppose that we have two logging aspects shown in Listing 4.5. The advice in the TraceLogging aspect records executed methods during program execution. The ArgumentLogging aspect records the values of arguments when a method is invoked. If the precedence order specifies that TraceLogging is executed before ArgumentLogging, then a printed method name is followed by argument values. However, if a program is multi-threaded, the two advices must be synchronized. Otherwise, printed log messages will be

```
aspect TraceLogging {
    before log(): ... {
        Logger.getInstance()
            .debug(thisJoinPointStaticPart.toString());
    }
}

aspect ArgumentLogging {
    before log(): ... {
        Object[] args = thisJoinPoint.getArgs();
        StringBuilder msg = new StringBuilder("Arguments: ");
        for (int i = 0; i < args.length; i++) {
            msg.append(args[i].toString());
            if (i != args.length - 1) {
                msg.append(',');
            }
        }
        Logger.getInstance().debug(msg.toString());
    }
}
```

Listing 4.5: Two aspects for logging

interleaved as the following:

```
[DEBUG] execution(Object Main.run(String))
[DEBUG] execution(void Test.test())
[DEBUG] Argument:
[DEBUG] Argument:  --debug
```

Here, the forth line shows the value of the arguemnt to the run method.

Listing 4.6 shows a resolver for synchronizing the two logging advices. This resolver uses two helper advices lock and unlock. Note that this resolver

```
[Modifiers] RetrunType around|before|after
    AdviceName(ArgumentType ArgumentName, ...)
    [returning[(Type Name)]]
    [throwing[(Type Name)]]
    [throws ExceptionType, ...]
{ Body }
```

Figure 4.1: The syntax of helper advice declaration

```
aspect LoggingWithSync {
    before lock() {
        Logger.getInstance().lock(); //reentrant lock
    }

    before unlock() {
        Logger.getInstance().unlock();
    }

    void resolver logWithSync()
            and(TraceLogging.log, ArgumentLogging.log)
            uses lock, unlock {
        [lock, TraceLogging.log,
         ArgumentLogging.log, unlock].proceed();
    }
}
```

Listing 4.6: A resolver for synchronizing two aspects

has a uses clause, which specifies the helper advices for that resolver. The pointcut of a helper advice is not explicitly specified as its syntax shown in Listing 4.1; a helper advice is bound to the same join points that the resolver using that helper advice is bound to. The helper advices are included in the precedence order of proceed. In Listing 4.6, the lock advice is given the highest precedence while the the unlock advice is given the lowest precedence among the four advices. Thus, the lock advice acquires a lock, the logging advices print messages, and then the unlock releases before the method logged by the aspects is executed.

Without these helper advices, the resolver could not implement synchronization. since it had to release a lock between the logging advices and the logged method but the resolver automatically obtains higher precedence than the logging advices. For example, the next code can not provide the behavior of logWithSync in Listing 4.6:

```
void resolver badLogWithSync()
       and(TraceLogging.log, ArgumentLogging.log) {
    Logger.getInstance().lock();
    [TraceLogging.log, ArgumentLogging.log].proceed();
    Logger.getInstance().unlock();
}
```

This is because the original computation at the join points is also syncronized.

```
aspect IntegerStringAspect {
    Object around(Plus t):
            execution(Object Plus.eval()) && target(t) {
        Object lvalue = t.getLeft().eval();
        Object rvalue = t.getRight().eval();
        if (lvalue instanceof String && rvalue instanceof Integer ||
            lvalue instanceof String && rvalue instanceof String) {
            return lvalue.toString() + rvalue.toString();
        } else if (lvalue instanceof Integer &&
                   rvalue instanceof Integer) {
            return (Integer)lvalue + (Integer)rvalue;
        } else {
            throw new RuntimeException();
        }
    }

    declare precedence:
        IntegerStringAspect, IntegerAspect, StringAspect;
}
```

Listing 4.7: Another incomplete solution in AspectJ

Multiple resolvers may use the same helper advice. If those resolvers are bound to the same join points, that helper advice is executed only once at every join point (shadow). The following code adds the AnotherLogging aspect with a resolver, which uses unlock again:

```
  void resolver logWithSync1()
         and(LoggingWithSync.logWithSync,
             AnotherLogging.log
             ArgumentLogging.log)
         uses(unlock) {
       [LoggingWithSync.logWithSync, ArgumentLogging.log,
        AnotherLogging.log, unlock].proceed();
  }
```

If a resolver removes another resolver using a helper advice, that helper advice is not removed together. It must be explicitly removed.

4.5 Discussion

4.5.1 Needs of our language constructs

A resolver does not take a normal pointcut but an and/or clause — a list of conflicting advices. It can call proceed with precedence. These are unique features of our language Airia. To clarify their benefits, we show another aspect in Listing 4.7. Like the aspect written in Airia, this aspect does not require us to modify the conflicting aspects in Listing 3.2 and 3.3. We wrote this aspect in AspectJ to be similar to the aspect written in Airia shown in Listing 4.1. The aspect has a normal around advcie. We manually translated the and clause of the resolver into a normal pointcut for this around advice. In the body of this around advice, we also manually inlined the body of the conflicting advices since a proceed call with precedence is not available.

This aspect has two drawbacks. First, the pointcut of the advice is fragile. We will have to modify the pointcut of this advice when the pointcuts of the conflicting advices are modified. Second, the body of this advice contains code duplication since we manually inlined the body of the conflicting advices. We will also have to modify the advice body when the bodies of the conflicting advices are modified. The aspect written in Airia does not have these problems.

4.5.2 Join points when a resolver is executed

A resolver is executed at the join point shadow where the specified advices conflict. We adopted this language design since it is extremely difficult to detect conflicts among advices even at runtime. This is because an advice in AspectJ can change the dynamic contexts.

A trivial definition is that the pointcut of a resolver is conjunction/disjunction of all advices and resolvers specified in the and/or clause but it is not satisfactory. For example, there are the following advices and resolvers:

```
pointcut pc(): ...;
void around A(): pc() && if(flag) {...}
void around B(): pc() && if(flag) {...}
void resolver AandB() and(A, B) {
    [A, B].proceed();
}
void around C(): pc() {
    flag = true;
    proceed();
}
void resolver AandBandC and(AandB, C, A) {
    flag = false;
    [AandB, C, A].proceed();
}
```

First, the resolver AandBandC is executed. The resolver is set flag to false. The resolver with the next highest precedence is AandB. However since its pointcut is pc() && if(flag) && pc() && if(flag) and does not match the current context, it is not executed. Then the advice C is executed and it change flag to true. Now the advice A and B is executable and conflict but they have not been combined by AandB.

Another definition is evaluating pointcuts first at the join point shadow and collecting executable advices. However, this prevent advices to change context dynamically. In AspectJ, advices can change values of this object or arguments bound to the this or the args pointcut by specifying new values as arguments of proceed. This feature is an advantage of AspectJ but cannot be provided in this definition.

Chapter 5

The execution order of advices

Our compiler statically calculates execution order of advices from precedence declared by resolvers. We restrict the semantics of a resolver to achieve this. We show an operational definition of execution order.

5.1 Precedence order in Airia

Advices are executed in the order that fulfills all declared precedence relation. In our language Airia, constructs that affect precedence are only a resolver and a proceed call with precedence as summarized in Table 5.1. A resolver has higher precedence than conflicting advices that are specified in its and/or clause. Our proceed call declares precedence in the order written between its brackets. This precedence is effective only in advices which is executed through this proceed call until it returns. We use a binary relation $\prec$ here; $a \prec b$ represents that a has higher precedence than b. This relation is transitive, *i.e.*, if $a \prec b$ and $b \prec c$ then $a \prec c$.

If advices or resolvers conflict, they have to be explicitly combined by

Table 5.1: Summary of precedence declared by constructs in Airia

construct	precedence
Type resolver R() and/or(A, B, C)	R $\prec$ A, R $\prec$ B, R $\prec$ C
[A, B, C].proceed()	A $\prec$ B, B $\prec$ C

at least a resolver. Otherwise there is no precedence relation between these advices. For example, the advice A and B conflict, there are two possible sequences A, B and B, A. Since the total execution order can not be determined, a compile error is reported.

5.2 Static calculation of the execution order

In Airia the execution order of conflicting advices is determined statically at every join point shadow. The errors of precedence declared by resolvers are also checked. This is an advantage over other approaches using meta programming, which cannot check consistency at compile time.

Like AspectJ, our definition of conflict is static; if the join point shadows of two advices are the same, they conflict. For example, two advices conflict if one's pointcut is call(int C.m()) && if (C.f) and the other's pointcut is call(int C.m()) && if (!C.f). Dynamic pointcut if (C.f) and if (!C.f) are ignored. At runtime, since C.f and !C.f do not become true at the same time, the two advices are never executable at the same join point unless C.f is changed.

The more complicated case is pointcuts that checks actual types of values such as the this pointcut. They are not completely ignored. For example, two advices have pointcuts call(int C.m()) && this(T) and call(int C.m()) && this(S) respectively. Only if apparent type of this which calling C.m() is a super type of both T and S, these advices conflict. Note that resolvers never have dynamic pointcuts. They are executed when specified advices conflicts.

We restrict semantics of Airia so that total execution order can be determined statically without loosing composability. First, precedence from resolver definition, which is shown in the first row in Table 5.1, is permanent and cannot removed. If removing the next resolver using our proceed call, since the resolver is not executed, precedence specified by the next proceed is not take effect:

```
Object resolver R() and(A, B) {
    [A, B].proceed();
}
```

However the precedence $R \prec A$ and $R \prec B$ still effective. This rule is introduced to prohibits resolvers that combine resolvers circularly like this:

```
void resolver R() and (S, ...)  {
   [].proceed();
}
void resolver S() and (R, ...)  {
   [].proceed();
}
```

Each resolver try to remove the other resolver. Without this rule, the resolver executed first can not be determined.

With the restriction above, the execution order of advices can be statically determined for every control path of proceed from the join point (shadow) to the original computation. Our compiler of Airia performs abstract computation at every occurrence of a join point shadow. If the body of the executed advice contains proceed calls, it computes which advice will be executed next. Since resolvers which combines the next advice has higher precedence than it, relations that affect the calculation is declared by proceed calls and resolvers that is already executed. Therefore if the precedence is specified correctly, it should be determined. While executing body of resolver, all control statements are ignored and all possible paths is executed. If the compiler cannot determine the next advice, it reports compile error.

5.3 Operational definition of the execution order

The execution order of advices at a given static join point is determined as the following. First, we collect all the (potentially) conflicting advices and resolvers at the given static join point. During this collection, the dynamic part of the pointcut (*i.e.* the pointcut residue) of an advice is ignored. Then we make a directed graph $D(V, E)$ where a vertex $v \in V$ represents an advice or a resolver and an edge $u \prec v \in E$ represents that an advice u has higher precedence than another advice v. If a resolver r has an and/or clause including a, b, ..., then $r \prec a$, $r \prec b$, ... We also make an empty set M, whose elements are edges $v \in V$.

Now(*), we find a top advice t, where $t \prec v$ for any $v \in V \setminus M$. Here, $V \setminus M$ represents a set of the elements contained in V but not in M. Note that $\prec$ is transitive. If $V \setminus M$ is empty, there is no other advice. We execute the original computation at the join point. If there is no unique t or if the graph D has a cycle, an error is reported. Otherwise, we execute the body of the top advice t.

During the execution of t, suppose that we execute proceed in the body of t. Then we add t into M. If the proceed has precedence order [a, b, c, ...],

we add $t \prec a$, $a \prec b$, $b \prec c$, ... to the graph D. If an advice a is included in the and/or clause of t and the precedence order does not include a, then a is added to M. Finally, we go back to * to find a top advice t for D and M again as we did above.

Recall that a top advice t is a potentially conflicting advice at the current static join point. Hence t might be unexecutable since the join point residue of t does not match the current runtime contexts. If it does not match, the body of t is not executed. t is added into M and we find and execute a top advice for D and the new M.

Now we show steps of this algorithm using an example. We assume that normal advices A, B, C, and D conflicts at a join point and then combine these advices with the next resolvers:

```
void R() and(A, B) {
    [A, B].proceed();
}
void S() and(R, C, B) {
    [R, C, B].proceed();
}
void T() and(R, S, D, B, C) {
    [D, R, B, C].proceed();
}
```

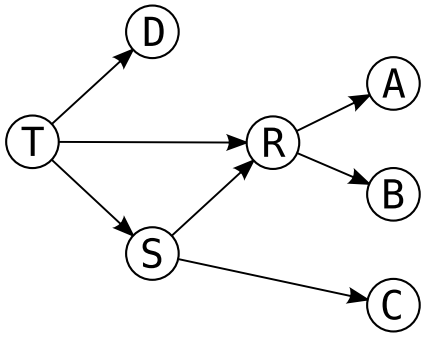

Figure 5.1: A graph representing the precedence declared by the resolver definitions

First, we create a directed graph shown Figure 5.1 with edges that represents precedence relations declared by the resolver definitions. Since the relation is transitive, some edges such as T $\prec$ B are omitted.

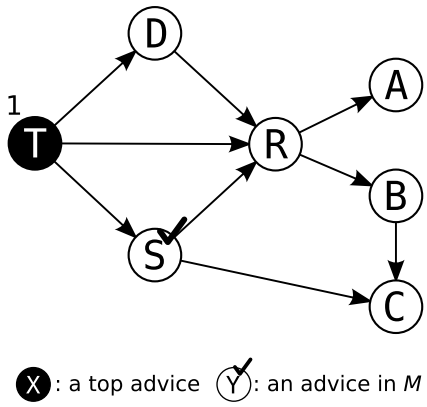

Figure 5.2: Adding edges specified in the resolver T

Next the current top resolver is T in Figure 5.2. The precedence relations specified by the proceed call in T are added to the graph. Since the resolver S is removed by the proceed call, we add S in M. The next top advice is D, which has higher precedence than all advices and resolvers that are not contained in M.

After the advice D calls a proceed, the resolver R is top in Figure 5.3. It appends the relation A $\prec$ B specified by its proceed call to the graph. Through calling the proceed, a top advice transits from A to C via B. There is only the proceed path [D, R, B, C].proceed, [A, B].proceed(). The total execution order of the advices for this path is T, R, A, B, C.

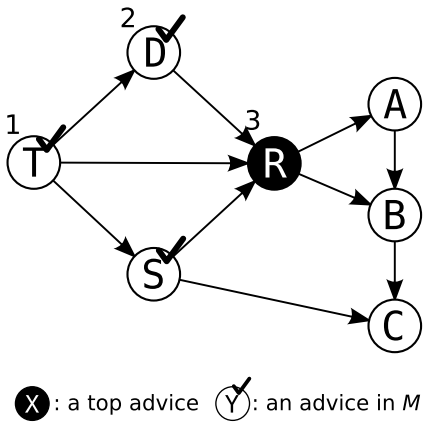

Figure 5.3: Adding edges specified in the resolver R

Chapter 6

Other examples

We show two other examples here. The first is aspects applied to a shape editor. The other is two authorization aspects. We permit to access our web application only when either of the aspects permits.

6.1 A figure editor

We show a shape editor as an example. Listing 6.1 shows the Shape class, which represents a shape on this editor. Shape has methods, setWidth/setHeight and getWidth/getHeight to set and get its width/height. The scale method change its width and height to the specified scale.

We extend this shape editor to hold values of the width and the height in double precision. This is implemented in the DoubleCoordinateAspect shown in Listing 6.2. Since the original scale method rounds values of the width/height to integers, when the method is called repeatedly, the precision of the width/height is lost. DoubleCoordinateAspect addresses this problem. Now we append double-typed fields, dblWidth and dblHeight into Shape with inter-type declarations. To store values of the width/height in dblWidth/dblHeight, a scale method is overridden by the scaleDbl advice. The original fields of shape class are not used any more. The getWidth/getHeight and the setWidth/setHeight method are also overridden. They get values from and set values to new fields.

Next Listing 6.3 shows the ReallyChanged aspect and the ObseverProtocol aspect. When setWidth is called, if the values of width is not changed after this call, ReallyChanged execute nothing. It also prevents execution of ObserverProtocol. ObserverProtocol is an aspect that implement the observer pattern [11]. It updates a screen of the shape editor when a shape is

```
public class Shape {
    int width;
    int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public int getWidth() {
        return width;
    }

    public void setHeight(int width) {
        this.height = height;
    }

    public int getHeight() {
        return height;
    }

    public void scale(double scale) {
        width = (int) Math.round(width * scale);
        height = (int) Math.round(height * scale);
    }
}
```

Listing 6.1: The Shape class for the shape editor

changed. ObserverProtocol must have higher precedence than ReallyChanged, otherwise ObserverProtocol is executed at every time.

These advices conflict at execution of Shape.setWidth, for instance. Suppose that ReallyChanged has higher precedence than DoubleCoordinate. Now there is a shape and its width is 4.2, that is to say, dblWidth has 4.2. Then we call setWidth(4), which should change its width to 4. When ReallyChanged is executed, a new value 4 is compared to current value that getWidth returns. Since getWidth round the value of dblWidth 4.2 to 4, the ReallyChanged aspect decides that the value is not to be changed. Thus the value of dblWidth remains 4.2 but this is undesirable behavior.

There is no precedence order among these advices. If DoubleCoordinate has higher, ReallyChanged is not executed because DoubleCoordinate does not call a proceed. Like our example in Chapter 3, this interference is not resolved with declare precedence.

We can resolve this interference by a resolver in Listing 6.4. It is a new implementation of ReallyChanged which is aware of DoubleCoordinate. When setWidth is called, the resolver is executed first. Now its argument is casted to the double type and compared to a value of dblWidth. If it is changed, ObserverProtocol and DoubleCoordinate are executed. ReallyChanged is removed because it is reimplemeted in the resolver.

```
aspect DoubleCoordinate {
    double Shape.dblWidth;
    double Shape.dblHeight;

    void around setWidthDbl(double width, Shape shape):
            execution(void Shape.setWidth(int)) &&
            args(width) && target(shape) {
        shape.dblWidth = width;
    }

    int around getWidthDbl(Shape shape):
            execution(int Shape.getWidth()) && target(shape) {
        return (int)Math.round(shape.dblWidth);
    }

    void around setHeightDbl(double height, Shape shape):
            execution(void Shape.setHeight(int)) &&
            args(height) && target(shape) {
        shape.dblHeight = height;
    }

    int around getHeightDbl(Shape shape):
            execution(int Shape.getWidth()) && target(shape) {
        return (int)Math.round(shape.dblHeight);
    }

    void around scaleDbl(double scale, Shape shape):
            execution(void Shape.scale(double)) &&
            args(scale) && target(shape) {
        shape.dblWidth *= scale;
        shape.dblHeight *= scale;
    }
}
```

Listing 6.2: The DoubleCoordinate aspect

```
aspect ReallyChanged {
    around setWithRC(int width, Shape shape):
            execution(void Shape.setWidth(int)) &&
            args(width) && target(shape) {
        if (shape.getWidth() != width) {
            proceed(width, shape);
        }
    }

    around setHeightRC(int height, Shape shape):
            execution(void Shape.setHeight(int)) &&
            args(height) && target(shape) {
        if (shape.getHeight() != height) {
            proceed(height, shape);
        }
    }

    around scaleRC(double scale):
            execution(void Shape.scale(scale) && args(scale) {
        if (scale != 1) {
            proceed(scale);
        }
    }
}

aspect ObserverProtocol {
    after onChanged():
            execution(void Shape.setWidth(int)) || ... ||
            execution(void Shape.scale(double)) {
        //observer.notify();
    }
}
```

Listing 6.3: The ReallyChanged and the ObserverProtocol aspect

```
aspect DoublCoordinateReallyChanged {
    void resolver setWidth(int w, Shape s)
           and(DoubleCoordinate.setWidthDbl(w, s),
               ReallyChanged.setWidthRC,
               ObserverProtocol.onChanged) {
        if (s.dblWidth != (double)w) {
            [ObserverProtocol.onChanged,
             DoubleCoordinate.setWidthDbl].proceed(w, s);
        }
    }
    //:
}
```

Listing 6.4: The aspect for combining DoubleCoordinate and ReallyChanged

```
abstract class CGI {
    void run() {
        String content;
        try {
            content = generate();
        } catch(RuntimeException e) {
            content = generateErrorMessages(e);
        }
        System.out.println("Content-Type: text/html");
        // :
        System.out.println(content);
    }

    abstract String generate();
    abstract String generateErrorMessages(RuntimeException e);
    // :
}

class Index extends CGI {
    String generate() {
        return /* a generated html string */;
    }

    String generateErrorMessages(RuntimeException e) {
        return /* a generated html string */;
    }
    // :
}
```

Listing 6.5: The classes of our web application

6.2 Combination of two authorization aspects

The next example is two authorization aspects applied to a web application which is executed through CGI (common gateway interface). Listing 6.5 shows classes of the application. CGI is the base class of the application. When index.cgi is accessed from a web browser, the Index.run method is executed. It invokes generate method to generate content of this page. If an error occurs during generating a page, an exception is thrown. The exception is caught in the CGI.run method and the content of its error message is generated. The content is output to the client through the standard output stream.

```
aspect SessionAuth {
    void around auth(CGI cgi):
            execution(String Index.generate()) && target(cgi) {
        if (!getSession().isLogined()) {
            throw new AccessDeniedException();
        }
        return proceed();
    }
    // :
}

aspect HostAuth {
    void around auth():
            execution(String Index.generate()) && target(cgi) {
        if (!isPermitted(cgi.getHost())) {
            throw new AccessDeniedException();
        }
        return proceed();
    }
    // :
}
```

Listing 6.6: Two authorization aspects

Two authorization aspects are shown in Listing 6.6. First, the SessionAuth aspect permit to access this page only if a user have already logged in with his or her password. The advice is executed before a page is generated. If the user have not logged in, it throw AccessDeniedException and error messages are displayed to the user. Otherwise, it calls the proceed and the content of the page is displayed. Then, the HostAuth aspect permit to access only from specified hosts in the same way as SessionAuth.

It is easy to make the application so that users can access it only from specified hosts after they have logged in. If we specify the next precedence between the aspects, the expected behavior can be obtained:

```
String resolver auth()
       and(SessionAuth.auth, HostAuth.auth) {
   return [SessionAuth.auth, HostAuth.auth].proceed();
}
```

This composition can be also described with declare precedence. Since the two advices are commutative, the other precedence order is also acceptable.

However, disjunction of the conditions of the two advices is difficult.

```
aspect SessionOrHostAuth {
    boolean isAllowed;
    String resolver auth() and(SessionAuth.auth, HostAuth.auth) {
        try {
            return [SessionAuth.auth].proceed();
        } catch(AccessDeniedException e) {
            return [HostAuth.auth].proceed();
        }
    }
    // :
}
```

Listing 6.7: The resolver the two aspects with an or operator

Now we want to permit to access the application without passwords if the user accesses it from specified host, such as a host on a local network. The authorization like this can not be achieved by combining the advice with any precedence order. This composition require additional composition code.

We can implement the composition code for the authorization with a resolver. Listing 6.7 shows the resolver. Since the advices throw an exception if access is denied, the resolver catches the exception and retries another authorization. First, it invokes the advice in SessionAuth and if access is not permitted, it invokes HostAuth in the catch clause.

Chapter 7

Implementation of Airia

Airia have been implemented as an extension of the AspectBench compiler (abc) 1.3.0, which is an implementation of AspectJ compiler. It is designed for extensibility; a language extension of AspectJ, such as Airia, can be implemented as its extension (plugin). It is separated in two part: the frontend, which is mentioned below, and the backend.

The backend receives ASTs of Java and information about aspects from the frontend. The ASTs are translated to intermediate codes by Soot [21]. The code of the aspects are inserted into them by the weaver. The data structures and the weaver is described in pure Java and they lacks extensibility. In order to extend a part of weaver as a plugin, we need duplicating some classes from original ones and modifying them.

7.1 The front end of the compiler

The front end have been extended without modification of the existing code unlike the backend. The front end of this version of abc is implemented by extending the JastAddJ [10] extensible compiler, which is an Java compiler developed by JastAdd. JastAdd is Java-based compiler compiler system. Since concerns such as type checking and code generation is crosscuts over classes which represent ASTs, in JastAdd, they are separated into aspects. Like our example in Chapter 3, the front end of Airia consist of aspects. JastAddJ provides aspects for Java 1.4 and for Java 1.5. The aspects for AspectJ are provided from abc. We have implemented aspects for Airia. The classes for ASTs to which the aspects is woven is automatically generated by the parser generator Beaver [1]. Only by describing the difference of the syntax for the generator, we can extend the ASTs.

```
refine ImplicitVariables public void AdviceSpec.nameCheck() {
    if (adviceName == null) {
        //error
    } else {
        AspectDecl aspectDecl = (AspectDecl)hostType();
        SimpleSet decls = aspectDecl.lookupMemberAdvice(adviceName);
        if (decls.iterator().next() != this.getParent()) {
            error(
                "An advice with name `" + adviceName +
                "` have already defined in `" +
                aspectDecl.getID() + "`"
                );
        }
    }
    //invoke refined method
    ImplicitVariables.AdviceSpec.nameCheck();
}
```

Listing 7.1: The refinement of existing aspect

In the aspects of Airia, only two methods that is declared with inter-type declarations conflict with existing methods. The methods implement code generation and name analysis of resolvers. The methods are extended by the technique called advice refinement. See Listing 7.1. This method implements the name analysis of the AdviceSpec node and refines the nameCheck method declared in ImplicitVariables aspect. Like our resolver and method override, the new method is executed first. The method invokes refined method at the second line from the bottom. Note that our aspects are not separated completely. If so, more methods might conflict.

Pointcuts of resolvers and helper advices is created implicitly. Thus, we could make the best reuse of the matcher provided by abc. The matcher searches join point shadows that a pointcut is bound to. A pointcut of a resolver is conjunction of all pointcut that the advices specified in the and clause of the resolver. It is translated in the backend so that a resolver is executed at the shadow. A pointcut of a helper advice is created in the same way.

7.2 The weaver for resolvers

Airia supports dynamic control of precedence but advices can be woven into base code statically because execution order of advice is determined

statically for each proceed path as mentioned in Section 5. We customized weaving process of advices and resolvers. Now they are woven for each proceed path.

Since a resolver is a new kind of around advice, weaving of a resolver is similar to that of an around advice. An around advice is basically implemented by a closure method. First, original computation overridden by a advice is moved into the closure. Then, the code for the advice is generated as a normal method, every invocation of super is replaced with invocation of the closure. Finally, the invocation of the advices are inserted to where the original computation was. If another advice is woven at the same join point, the invocation of that advice is replaced with an invocation of this advice.

In Airia, a closure is created for each proceed calls. Suppose that two around advices, which call a proceed, and the following resolver is woven at the same join point:

```
void resolver AandB() and(A, B) {
    [A, B].proceed();
    [B, A].proceed();
}
```

This code creates seven closures. For [A, B].proceed(), a closure containing original computation, one containing invocation of the advice B, and one containing invocation the advice A are created. For [B, A].proceed(), closures are created in similar way but exchanged A for B. A closure which invoke the resolver AandB is also created. Although the size of generated code can be large, we expect that it would not be critical size in realistic programs.

Chapter 8

Conclusion

8.1 Summary

We presented a language extension of AspectJ to achieve advice composition without interference. Some interference could not satisfactory resolved in original AspectJ because the conflicting advices have no correct precedence order. We need to modify the advices and implement resolving code in them. Furthermore the composition code is also a crosscutting concern.

This language named Airia can resolve interference among conflicting advices. Airia enables programmers to separate resolving code into an independent resolver. A resolver is a new kind of advice executed at join point when the specified advices conflict. In a resolver, we can execute conflicting advice with our extended proceed call. At that call, programmers declare precedence order among them and remove unnecessary advices. In our example, the interference is resolved by invoking only the selected advice with the proceed call depending on the dynamic context, concretely, actual types of the operands.

A resolver is composable. It can resolve interference between another resolver and an advice. Thus we can implement composition of advices in a hierarchical manner as IntegerAspect, StringAspect, and EvaluationCacheAspect are resolved with IntegerStringAspect and IntegerStringCacheAspect shown in Section 4.3.

A resolver is a special around advice and base-level construct in spite of its expressive power. The precedence specified by resolvers is examined and the total order of conflicting advices calculated statically. We have implemented Airia in the JastAdd version of the AspectBench compiler

8.2 Future work

8.2.1 Improvement of Airia

Our future work includes improving the expressive power of proceed calls. There are some proposals such as [2], which are used to detect whether or not advices are commutative, *i.e.* whether or not their combined behavior is independent of their precedence order. Stateful Aspect and Reflex supports to declare a combination of advices is commutative. In the current design of Airia, programmers have to explicitly specify the precedence order among advices even though they are commutative.

This is annoying and to make the matters worse, this might reduce composability. For example, there are two commutative advices A and B at first. After declaring the explicit precedence with the advices like [A, B].proceed() in a resolver, we add a new advice C and a resolver. If C is not commutative with A and C, they might require precedence like [B, C, A].proceed(). This precedence causes interference between these resolvers and a compile error is reported. In this case, we have to remove the former resolver.

A mechanism for directly accessing the instances of the aspects is another future work. An instance of aspect keep values in its fields like a normal class. Some interference might need to get and set the value from a resolver. In OARTA, since the whole body of an advice is overridden by around advice using the extended `adviceexecution` pointcut, the advice for composition can access the instance of aspects to which overridden advices belong through the this pointcut or the thisJoinPoint object. However in Airia, we can modify the values only by conflicting advices through calling a proceed.

8.2.2 Application to other languages

Not only to AspectJ, our idea of a proceed call with precedence can be applied to other languages. Our approach is more suitable to aspect-oriented language in which *hooks* (where) and *activations* (when) is separated, such as Reflex and JastAdd. JastAdd does not have advices but methods. The semantics of resolver, in particular the join points when a resolver is executed, is complicated but in those languages, the semantics might be more simple.

Non aspect-oriented languages are also included in the languages. In traits inheritance, in order to linearize methods of traits, programmers must define a class for each trait. For example, for the traits T1, T2, programmers need to define the class C1 which uses T1 and the class C2 which extends

C1 and uses T2. Since precedence is declarative and composable, we expect that this idea might achieve more flexible method composition.

Bibliography

[1] : Beaver - a LALR Parser Generator, `http://beaver.sourceforge.net/`.

[2] Aksit, M., Rensink, A. and Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points, *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 39–50 (2009).

[3] Apel, S., Leich, T. and Saake, G.: Aspect Refinement and Bounding Quantification in Incremental Designs, *Asia-Pacific Software Engineering Conference*, Vol. 0, pp. 796–804 (2005).

[4] Apel, S., Leich, T. and Saake, G.: Mixin-Based Aspect Inheritance, *Technical Report Number 10*, Germany, Department of Computer Science, University of Magdeburg (2005).

[5] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J.: abc: an extensible AspectJ compiler, *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 87–98 (2005).

[6] Dinkelaker, T., Mezini, M. and Bockisch, C.: The art of the meta-aspect protocol, *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 51–62 (2009).

[7] Douence, R., Fradet, P. and Südholt, M.: A Framework for the Detection and Resolution of Aspect Interactions, *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, London, UK, Springer-Verlag, pp. 173–188 (2002).

[8] Douence, R., Fradet, P. and Südholt, M.: Composition, reuse and interaction analysis of stateful aspects, *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 141–150 (2004).

[9] Ekman, T. and Hedin, G.: The JastAdd extensible Java compiler, *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, New York, NY, USA, ACM, pp. 1–18 (2007).

[10] Ekman, T. and Hedin, G.: The JastAdd extensible Java compiler, *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, New York, NY, USA, ACM, pp. 773–774 (2007).

[11] Hannemann, J. and Kiczales, G.: Design pattern implementation in Java and AspectJ, *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, ACM, pp. 161–173 (2002).

[12] Havinga, W., Nagy, I., Bergmans, L. and Aksit, M.: A graph-based approach to modeling and detecting composition conflicts related to introductions, *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 85–95 (2007).

[13] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, London, UK, Springer-Verlag, pp. 327–353 (2001).

[14] Marot, A. and Wuyts, R.: Composability of aspects, *SPLAT '08: Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies*, New York, NY, USA, ACM, pp. 1–6 (2008).

[15] Marot, A. and Wuyts, R.: A DSL to declare aspect execution order, *DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages*, New York, NY, USA, ACM, pp. 1–5 (2008).

[16] Marot, A. and Wuyts, R.: Composing Aspects in an Aspect, *AOSD '10: Proceedings of the 9th ACM international conference on Aspect-*

oriented software development, New York, NY, USA, ACM (2010). To appeared.

[17] Masuhara, H., Kiczales, G. and Dutchyn, C.: Compilation Semantics of Aspect-Oriented Programs, *Proc. of Foundations of Aspect-Oriented Languages Workshop*, AOSD 2002, pp. 17–26 (2002).

[18] Schrli, N., Ducasse, S., Nierstrasz, O. and Black, A. P.: Traits: Composable Units of Behaviour, *ECOOP 2003 – Object-Oriented Programming*, pp. 327 – 339 (2003).

[19] Suvée, D., Vanderperren, W. and Jonckers, V.: JAsCo: an aspect-oriented approach tailored for component based software development, *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp. 21–29 (2003).

[20] Tanter, E.: Aspects of Composition in the Reflex AOP Kernel, *Software Composition*, Lecture Notes in Computer Science, Vol. 4089, Springer Berlin / Heidelberg, pp. 98–113 (2006).

[21] Vallée-Rai, R., Gagnon, E., Hendren, L. J., Lam, P., Pominville, P. and Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?, *Compiler Construction, 9th International Conference (CC 2000)*, pp. 18–34 (2000).