Type Relaxed Weaving

Hidehiko Masuhara
University of Tokyo
masuhara@acm.org

Atsushi Igarashi
Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Manabu Toyama
University of Tokyo
touyama@graco.c.u-tokyo.ac.jp

ABSTRACT

Statically typed aspect-oriented programming languages restrict application of around advice only to the join points that have conforming types. Though the restriction guarantees type safety, it can prohibit application of advice that is useful, yet does not cause runtime type errors. To this problem, we present a novel weaving mechanism, called the *type relaxed weaving*, that allows such advice applications while preserving type safety. We formalized the mechanism, and implemented as an AspectJ compatible compiler, called *RelaxAJ*.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Language Constructs and Features—*modules, packages*

Keywords

Aspect-oriented programming, around advice, type safety

1. INTRODUCTION

The advice mechanism is a powerful means of modifying behavior of a base program without changing its original text. AspectJ[10] is one of the most widely used aspect-oriented programming (AOP) languages that support the advice mechanism. It is, in conjunction with the mechanism called the inter-type declarations, shown to be useful for modularizing crosscutting concerns, such as logging, profiling, persistency and security enforcement[1, 3, 16, 19].

One of the unique features of the advice mechanism is the *around advice*, which can change parameters to and return values from operations, or *join points*, in program execution. With around advice, it becomes possible to define such aspects that directly modify values passed in the program, for example caching results, pooling resources and encrypting parameters. Around advice can also modify system's functionalities by inserting proxies and wrappers to objects that implement the functionalities, or by replacing the objects with new ones.

Statically typed AOP languages restrict pieces of around advice so that they will not cause type errors. Basically, given a piece of around advice that replaces a value with a new one, those languages permit the advice if the type of the new value is a subtype of the join point's.

Though the above restriction seems to be reasonable, it is sometimes too strict to support many kinds of advice needed in practice. In fact, AspectJ and StrongAspectJ[6] relax the restriction in order to support generic advice declarations that are applied to join points of different types. The AspectJ's mechanism gives a special meaning to `Object` type in around advice declarations at the risk of runtime cast errors. The StrongAspectJ's mechanism guarantees type safety.

This paper relaxes the restrictions by giving different kind of type genericity to around advice. While existing languages type-check with respect to types of join points, our proposed mechanism checks how the values supplied from around advice are used in subsequent computation. This enables to weave pieces of around advice that have been rejected in existing languages, while guaranteeing type-safety. Examples of such pieces of advice include the one that replaces an object with another object of a sibling class, and the one that inserts a wrapper to an anonymous class.

In the rest of the paper, we first introduce around advice in AspectJ in Section 2. We then, in Section 3, show that several useful around advice declarations are rejected in AspectJ due to its type-checking rules. Section 4 proposes our new mechanism, called *type relaxed weaving*, that accepts such around advice declarations. The section also discusses subtle design decisions to guarantee type-safety. We formalized a core part of the mechanism to show its type soundness in Section 5. We also implemented the mechanism as an AspectJ compatible compiler called *RelaxAJ*, as described in Section 6. After discussing related work in Section 7, Section 8 concludes the paper.

2. AROUND ADVICE IN ASPECTJ

This section introduces the around advice mechanism in AspectJ. Readers familiar with its type checking rules can skip this section.

We first show a method to which we will apply pieces of around advice. We sometimes call it *a base program* or *a base method.* The `store` method (shown in Listing 1) stores graphical data into a file in a graphical drawing application.[1]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'10 March 15–19, Rennes and St. Malo, France
Copyright 2010 ACM 978-1-60558-958-9/10/03 ...$10.00.

[1]The method is taken from JHotDraw (`http://www.`

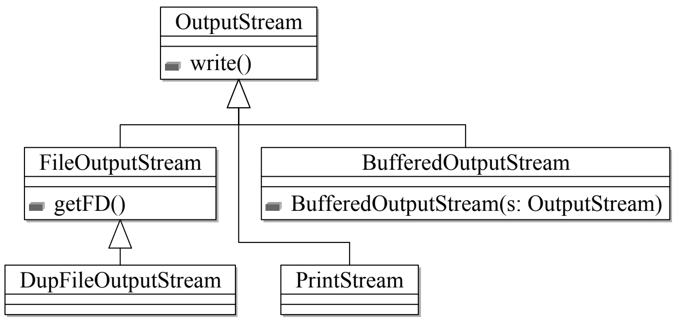

Figure 1: UML class diagram of the classes that appear in the examples.

It is executed when the user selects the save menu. The hierarchy of the relevant classes is summarized in Figure 1.

```
void store(String fileName, Data d) {
  FileOutputStream s
    = new FileOutputStream(fileName);
  BufferedOutputStream output
    = new BufferedOutputStream(s);
  output.write(d.toByteArray());
  output.close();
}
```

Listing 1: A method that stores graphical data into a file.

Assume we want to duplicate every file output to the console for debugging purposes. Without AOP, this can be achieved by replacing "`new FileOutputStream(fileName)`" in the third line of the `store` method with "`new DupFileOutputStream(fileName)`", where `DupFileOutputStream` is a subclass of `FileOutputStream` as defined below.

```
class DupFileOutputStream extends FileOutputStream {
  void write(int b) {
    super.write(b); System.out.write(b);
  }
  ...overrides other write methods as well...
}
```

With AOP, instead of textually editing the `store` method, we can write an around advice declaration that creates `DupFileOutputStream` objects instead of `FileOutputStream`, as shown in Listing 2. In AspectJ, advice declarations are written in aspect declarations, which we omit in the paper for simplicity.

The advice declaration begins with a *return type* (`FileOutputStream`), which specifies a type of values returned by the advice, followed by the `around` keyword. Between the subsequent parentheses, there are formal parameters to the advice (`String n`). The next element is a *pointcut*, which specifies when the advice will run. The pointcut in Listing 2 specifies constructor calls to the `FileOutputStream` class, and also binds the constructor parameter to the variable `n` when it matches. Finally, a *body* of the advice, which consists of Java statements, is written in the braces.

`jhotdraw.org/`) version 6.0b1, but simplified for explanatory purposes.

```
DupFileOutputStream around(String n):
    call(FileOutputStream.new(String)) && args(n) {
  return new DupFileOutputStream(n);
}
```

Listing 2: A piece of advice that creates `DupFileOutputStream` objects instead of `FileOutputStream` objects.

Execution of a base program with pieces of around advice can be explained in terms of *join points*, which represent the operations in program execution whose behavior can be affected by advice. In AspectJ, the join points are the operations about objects, including method and constructor calls, method and constructor executions, field accesses, and exception throwing and catching operations. It should be noted that the operations about local variables are not included.

When a program is to execute an operation, we say the program creates a join point corresponding to the operation. When there is any around advice declaration that has a pointcut specifying the join point, the body of the advice runs instead of the join point. For example, when the `store` method runs, at line 3, it creates a join point corresponding to a constructor call to `FileOutputStream`. Since the join point is specified by the pointcut of the advice in Listing 2, the program creates a `DupFileOutputStream` object instead of `FileOutputStream`. If there is more than one around advice declaration matching a join point, one of them, selected by a certain rule, will run. Calling a pseudo-method `proceed` in the body of around advice lets the next around advice run, if there is one. Otherwise, the operation corresponding to the original join point is executed.

Practical AOP languages *weave* advice declarations into a base program. AspectJ, for example, weaves advice declarations in two steps. First, it compiles class definitions into bytecode as if they are pure Java definitions. It also transforms a body of each advice declaration into a Java method in a bytecode format. Second, it examines bytecode instructions generated at the first step. For each sequence of instructions that creates join points at runtime, which is called a *join point shadow*, if there is an advice declaration that specifies join points created from the shadow, it replaces the instructions with a call instruction to a method translated from the advice body. When the advice has a *dynamic pointcut*, which specifies join points by using runtime information, AspectJ inserts a series of instructions that evaluate the runtime condition, and then branch to either a call to the method translated from the advice body, or a call to the original method.

For the example in Listing 1, the expression `new FileOutputStream(fileName)` is compiled into a sequence of bytecode instructions that creates and initializes an object, which is a join point shadow. Since the shadow creates constructor-call join points that are specified by the pointcut of the around advice in Listing 2, the compiler replaces the instructions with a call to a method that is generated from the body of the around advice, which creates a `DupFileOutputStream` object.

3. TYPE-CHECKING RULES AND THEIR PROBLEMS

3.1 The Rules

AspectJ compilers type-check around advice on the following two regards: (1) each advice body should be consistent with a return type of the advice declaration; and (2) the return type of an advice declaration should be consistent with join point shadows where the advice is woven into. Below we present more detailed rules.[2]

The first rule is similar to the rule for a return statement in a method.

RULE 1 (ADVICE BODY). *Any return statement in the body of around advice must have a subtype[3] of the return type of the advice declaration.*

After checking this rule, we can trust the advice's return type. An advice body that does not match the return type like the one shown below will be rejected.

```
FileOutputStream around():
    call(FileOutputStream.new(String)) {
  return "not a stream";             // type error
}
```

The second rule ensures that around advice will be woven into "right" join point shadows.

RULE 2 (ASPECTJ WEAVING). *The return type of an around advice declaration must be a subtype of the return type of any join point shadow where the advice is woven into.*

For example, AspectJ allows to weave the around advice in Listing 2 into an expression `new FileOutputStream(fileName)` because the return type of the advice (i.e., `DupFileOutputStream`) is a subtype of the return type of the join point shadow (i.e., `FileOutputStream`).

However, AspectJ reports an error when it attempts to weave the following into `new FileOutputStream(fileName)`.

```
String around():                     // weave error
    call(FileOutputStream.new(String)) {
  return "not a stream";
}
```

This is because, the return type of the advice (i.e., `String`) is not a subtype of the return type of the join point shadow (i.e., `FileOutputStream`).

Note that AspectJ rejects the above advice when it weaves the advice into a join point shadow, but not when it checks the advice declaration itself. In other words, AspectJ does not use the pointcut expression when it checks the advice declaration itself, even if it suggests that the advice will be woven into join points of type `FileOutputStream`.

3.2 An Example of the Problem: Replacing with a Sibling

The rules in AspectJ are too restrictive and sometimes prohibit to define useful advice. For example, assume we want to redirect the output to the console, instead of duplicating. At first, it would seem easy to define a piece of advice that returns `System.out`, which represents a stream to the console, at the creation of a `FileOutputStream` object. However, it is impossible to straightly define such advice in AspectJ.

Listing 3 defines a piece of advice, which tries to return `System.out` whenever a program is to create a `FileOutputStream` object.

```
PrintStream around():  // weave error in AspectJ
    call(FileOutputStream.new(String)) {
  return System.out; // of type PrintStream
}
```

Listing 3: A piece of advice that returns the `PrintStream` object (i.e., the console) instead of creating `FileOutputStream`.

Even though the definition looks fine, RULE 2 rejects the advice in the aspect because the return type of the advice (i.e., `PrintStream`) is not a subtype of the join point shadow's return type (i.e., `FileOutputStream`, as shown in Figure 1). Changing the return type in Listing 3 to `OutputStream` does not help because `OutputStream` is not a subtype of `FileOutputStream`, either. Changing the return type in Listing 3 to `FileOutputStream` satisfies RULE 2 but violates RULE 1.

Interestingly, if we can edit the body of the method `store`, it can be easily achieved by replacing the second line of `store` (Listing 1)

```
FileOutputStream s = new FileOutputStream(fileName);
```

with the next one.

```
OutputStream s = System.out;// of type PrintStream
```

3.3 Another Example: Wrapping Anonymous Classes

Another example of the problem can be found when we want to wrap handler objects. Java programs frequently use anonymous class[7, Section 15.9] objects in order to define handler objects, i.e., objects that define call-back functions. For example, the next code fragment creates two button objects in the Swing library and a listener object, and then registers the listener object into the button objects. The listener object belongs to an anonymous class that implements the `ActionListener` interface. The parameter type of the `addActionListener` method is `ActionListener`.

```
JButton b1 = new JButton();
JButton b2 = new JButton();
ActionListener a = new ActionListener () {
  void actionPerformed(ActionEvent e) { ... }
};
b1.addActionListener(a);
b2.addActionListener(a);
```

Now, assume that we want to wrap the listener object with a `Wrapper` object whose definition is shown in the first half of Listing 4.

Even though we want to define a piece of advice as shown at the last half of the Listing 4, RULE 2 rejects it because

[2]The rules are guessed by the authors from behavior of existing compilers, which are consistent with the ones presented by De Fraine, et al.[6], except that we omit the rules for ad-hoc generic typing with `Object`.

[3]In the paper, subtype and supertype relations are reflexive; i.e., for any `T`, `T` is a subtype and supertype of `T`.

```
class Wrapper implements ActionListener {
  Wrapper(ActionListener wrappee) { ... }
  ...
}
ActionListener around()://weave error in AspectJ
    call(ActionListener+.new(..)) {
  ActionListener l = proceed();
  return new Wrapper(l);
}
```

Listing 4: A wrapper class and an advice declaration that wraps `ActionListeners`.

the return type of the advice (i.e., `ActionListener`) is not a subtype of the return type of the join point, which is an anonymous class that implements `ActionListener`. Since the return type of the join point is anonymous, there is no way—except for generic ones—to declare a piece of around advice to that join point!

Note that the advice is applied to the constructions of anonymous class objects thanks to the plus sign in the pointcut description, which specifies any subtype of `ActionListener`.

Again, wrapping can be easily done if we edited the original code fragment as follows.

```
ActionListener a = new Wrapper(
  new ActionListener () {
    void actionPerformed(ActionEvent e) { ... }
  }
);
b1.addActionListener(a);
b2.addActionListener(a);
```

3.4 Relaxation Opportunities

We carried out preliminary assessment to investigate how likely the problem mentioned above can happen in practical application programs. We counted, in practical application programs, the number of such join point shadows whose return value is used merely as its strict supertype of the type of the shadows. As we see in the examples in Sections 3.2 and 3.3, the problem only happens at such join point shadows. In other words, if there are only few such join point shadows in programs, the problem is unlikely to happen. Since the assessment does not consider existence of useful advice in practice, it merely supports chances of existence of the problem.

We examined five medium-sized Java programs, and classified relations of a static type of a join point shadow in the programs and the most general type among static types in the operations that use the value from the shadow. For the ease of examination, we identified the dataflow relations by writing an aspect in AspectJ that dynamically monitors program executions. Note that the results are approximation as we used a dynamic monitoring technique.

The evaluated programs and the results are summarized in Table 1. The 'supertype' row shows the numbers of shadows whose return values are used only as strict supertypes of the return type in subsequent computation. Similarly, the 'subtype,' 'unrelated,' and 'same' rows show the number of shadows classified by the types in the operations using the returned values[4]. The numbers on the 'supertype' row, which are approximately 15–30% in the table, suggest that there are code fragments in practical applications in which the problem presented in the previous sections can happen.

3.5 Generality of the Problem

While the examples are about return values from around advice in AspectJ, the problem is not necessarily limited to those cases.

First, the problem is not limited to return values. Since around advice can also replace parameter values that are captured by `args` and `target` pointcuts, the same problem can happen at replacing those values by using the `proceed` mechanism. For example, if the `store` method in Listing 1 takes `FileOutputStream` rather than a file name string, the following around advice requires[5] relaxed weaving rules.

```
void around(OutputStream s): args(s,*) &&
    execution(void store(FileOutputStream,Data)) {
  proceed(System.out);
}
```

In this paper, however, we only consider types about return values, and leave types of parameter values to future work.

Second, the problem is not limited to AspectJ. Statically-typed AOP languages that support around advice, such as CaesarJ[15] and AspectC++[17], should also have the same problems.

Third, the problem is not limited to AOP languages. The same problem would arise with the language mechanisms that can intercept and replace values, such as method-call interception[11] and type-safe update programming[5].

4. TYPE RELAXED WEAVING

4.1 Basic Idea

We propose a weaving mechanism, called *type relaxed weaving*, to address the problem while preserving type safety. Roughly speaking, it replaces the Rule 2 with the following one.

Rule 3 (Type Relaxed Weaving). *When a piece of around advice is woven into a join point shadow, the return type of the advice must be consistent with the operations that use the return value from the join point shadow.*

We here used ambiguous terms such as "consistent with" and "operations," which shall be elaborated in the later sections.

For example, the rule allows the advice in Listing 3 to be woven into the `store` method (Listing 1). The return type of the advice (`PrintStream`) is consistent with the operations that use the return value, namely the `new` expression at line 5 in Listing 1.

4.2 Design Principles

We assumed the following principles in designing our weaving mechanism.

[4]The subtype and unrelated relations appear in programs that have downcasting and more than one interface type, respectively.

[5]Interestingly, an AspectJ compiler (ajc version 1.5.3) compiles the example. However, the generate code does not work as it contains a cast operation to `FileOutputStream`.

program name	Javassist	ANTLR	JHotDraw	jEdit	Xerces
program size (KLoC)	43	77	71	140	205
number of shadows	862	1,827	3,558	8,524	3,490
supertype(%)	177 (21)	315(17)	576 (16)	2,499(29)	650(19)
subtype(%)	37 (4)	70 (4)	170 (5)	974(11)	156 (4)
unrelated(%)	0 (0)	4 (0)	42 (1)	42 (0)	64 (2)
same(%)	648 (75)	1,438(79)	2,770 (78)	5,009(59)	2,620(75)

Table 1: Classification of join point shadows by the usage of the supplied values in practical applications.

Preservation of object interfaces: The mechanism should not change the signatures of methods and fields when weaving advice declarations into a program. Changing method/field signatures would give further freedom to advice, but is problematic with unmodifiable libraries and programs that use the reflection API.

Bytecode-level weaving: The mechanism weaves advice declarations into a bytecode program, in a similar way that most AspectJ compilers do. The mechanism, therefore, can merely use type information available in a bytecode program. At the same time, the mechanism can generate woven programs that cannot be generated from Java source code, as we shall discuss in Section 4.8.

Compatibility with AspectJ: The mechanism should accept a program that is accepted by existing AspectJ compilers, and should yield executable code that has the same behavior with the one compiled by existing AspectJ compilers.

Independence of advice precedence: Given advice declarations, the mechanism judges whether it is safe to weave them, regardless of their precedence. This makes the mechanism simpler, and to have more predictable behavior.

Check before weaving: Given a base program and advice declarations, the mechanism checks whether it is safe to weave them before actually generating woven code. An alternative approach is to generate woven code without checking safety, then type-check the woven code by using a standard bytecode verifier. It, however, does not work well because the mechanism sometimes needs to recompute target types of method invocations, as we will see in Section 4.5. The approach also makes the decision dependent on advice precedence.

4.3 Overview

The type relaxed weaving has the same mechanism as the ones in existing AOP languages, except for the part that type-checks advice to be woven. Our mechanism checks type-safety of advice in the following steps.

First, as explained in Section 2, the weaver processes each method in the base program. It looks for advice declarations that are applicable to join point shadows in the method.

Second, for each join point shadow, it performs dataflow analysis to identify operations that use a return value from the shadow. (The next section will discuss our choice of operations.) The extent of the analysis is within the method; i.e., it is an intra-procedural dataflow analysis. It then gathers type constraints based on the usage of parameters in the identified operations.

Finally, it solves the constraints with respect to the return types of around advice applicable to the shadows. If there is a piece of around advice whose return type that does not meet the constraints, it rejects the advice as a type-error.

Below, we discuss several issues of defining concrete weaving rules.

4.4 Operations that Use Return Values

Basically, operations that use values are determined by the semantics of the Java bytecode language and the principle of the interface preservation.

Below, we summarize the operations that use a value (denoted as `v`) returned from a join point shadow. Since we use a dataflow analysis, `v` actually denotes any variable that has a dataflow relation from a join point shadow.

Method or constructor call parameter: A method call `o.m(v)` uses `v` as of the parameter type in the signature of `m`. Similarly, a constructor call `new C(v)` uses `v` as of the parameter type in the signature of the constructor.

Note that the method and constructor signatures are determined before weaving. In other words, even if a piece of around advice changes types of some values, it will not change the selection of overloaded methods and constructors.

Method call target: A method call `v.m()` uses `v` as of the target type in the signature of `m`. There are subtle issues of selecting a target type. We will discuss it in Section 4.5.

Return value from a method: A return statement `return v` uses `v` as of the return type of the method.

Field access target: A field assignment `v.f=...` or a field reference `v.f` uses `v` as of the class that declares `f`.

Assigned value to a field: A field assignment `o.f = v` uses `v` as of the `f`'s type.

Array access target: An array assignment `v[i]=...` or an array reference `v[i]` uses `v` as an array type. The type of the elements is determined by the results of a dataflow analysis on the assigned or referenced value. For AspectJ compatibility, we regard all the reference types (i.e., classes, interfaces and arrays) as the same type when they appear as an element type of an array.

Exception to throw: A throw statement `throw v;` uses `v` as one of the types in the `throws` clause of the method declaration, or a type in one of `catch` clause around the `throw` statement.

Note that the above operations do not include accesses to a local variable. This gives the mechanism more opportunity to weave advice declarations with more general types. For example, in Listing 1, even when the return value from `new FileOutputStream(fileName)` is stored in a variable of type `FileOutputStream` at lines 2–3, the return value is not considered as used by the assignment. Only the subsequent operation `new BufferedOutputStream(s)` at line 5 is regarded as the operation that uses the value.

Ignoring types of local variables also fits for current AspectJ compilers, which support bytecode weaving. In Java bytecode, types of local variables are merely available as annotations when a program is compiled with a debug option.

4.5 Target Type of a Method Call

When a value is used as a target of a method call, we should examine a type hierarchy to decide the types that the value is used as. This is because, even though there is target type information in a Java bytecode instruction, the static type of the target object can be any subtype of a supertype of the target type, as long as the supertype declares the method.

For example, assume a method call `o.write(...)` whose target type is `BufferedOutputStream`. Since `write` is declared in `OutputStream` (as shown in Figure 1), it is safe to change the target type of the call to `OutputStream`. The static type of `o` can thus be any subtype of `OutputStream`.

Therefore, when a value is used as a target of a method call `m`, we regard the value is used as a value of the most general type that declares `m`, rather than the target type in the signature of `m`. This will give our weaver a chance to accept more advice.

However, when there are interface types, we need to consider that a target object is used as a value of one of several types, rather than a single type. Consider the interfaces and class declarations followed by a code fragment shown in Listing 5.

```
interface Runnable { ... }
interface Task     { void show(); }
class Simulator    { void show() { ... } }
class BkSimulator extends Simulator
                  implements Runnable,Task { ... }
// in a method:
BkSimulator o = new BkSimulator();
new Thread(o).start();
...
o.show(); //signature: void BkSimulator.show()
```

Listing 5: A method call with a class and interface types.

At the bottom line, we should regard that `o` is used as a value of *either* `Simulator`, `BkSimulator`, or `Task`, regardless of the target type in `show`'s signature. Since there is no subtype relation between `Simulator` and `Task`, we should at least consider these two types to maximize weaving opportunity.

Note that *covariant return types*[7, Section 8.4] in Java, which allows an overriding method in a subclass to have a more specific return type than the one in the overridden method, can complicate the safety condition. Our current approach here is to rely on the types in compiled bytecode, where signatures of two methods—an overriding method with a covariant return type and the method to be overridden—are regarded as different (i.e., not overriding).

4.6 Usage as a Value of Multiple Interface Types

Existence of interface types also requires to consider that an object is used as a value of a subtype of several unrelated types.

Listing 5, the constructor `new Thread(o)` has a parameter of type `Runnable`. Therefore, we should consider that `o` is used as a value of a subtype of *both* `Runnable` and `Task`, in order to increase a weaving opportunity.

This means that, when there is a class that implements both `Runnable` and `Task`, we can replace the return value from `new BkSimulator()` with an object of such a class even if it is not a subtype of `BkSimulator`. The following declarations demonstrate the case.

```
class RunnableTask implements Runnable,Task { ... }
RunnableTask around(): call(BkSimulator.new()) {
  return new RunnableTask();
}
```

Note that application of the above around advice to Listing 5 could generate woven code that cannot be generated from the Java source language when the pointcut contains a dynamic condition, such as `if(debugging)`. In this case, the woven code for the line

```
BkSimulator o = new BkSimulator();
```

will become the one like below, if translated back into the source language.

```
if (debugging) o = new RunnableTask();
else           o = new BkSimulator();
```

Even though it is not a valid source program because we cannot give a static type for `o`, the woven code is valid at the bytecode language level. We refer readers to formalizations of the Java bytecode language[12] for a detailed discussion.

4.7 Multiple Advice Applications at a Join Point

When the mechanism checks usage of a return value from a join point, it should also check the operations in advice bodies that are applied to the same join point. This is because, when there is more than one piece of advice applied to a join point, one of the pieces can use a return value from another piece by executing a `proceed` operation. Since our principle is not to rely on advice precedence, the mechanism should take all pieces of advice applied to the same join point into account.

For example, assume that the following advice is applied to the `store` method (in Listing 1) in conjunction with the advice in Listing 3.

```
FileOutputStream around():
    call(FileOutputStream.new(String)) {
  FileOutputStream s = proceed();
  s.getFD().sync();//uses s as BufferedOutputStream
  return s;
}
```

Our weaver then has to reject the combination of the above advice and the one in Listing 3. This is because, when the above advice runs prior to the advice in Listing 3, the former advice receives a return value from the latter, and uses the value as a `FileOutputStream` object by calling the `getFD` method (see Figure 1).

4.8 Advice Interference

When there are advice declarations applicable to more than one join point shadow in a method, it should check consistency among all those shadows in a method at once. In other words, it is not possible to employ an approach to safety checking in existing weavers, which check safety for each join point shadow.

The following example demonstrates a case in which two advice declarations cannot coexist, even though each of them can be safely applied without the other.

The case consists of two interfaces `I` and `J`, two classes `C` and `D` that implement both interfaces, followed by a code fragment that assigns objects in different classes into one variable.

```
interface I { C m(); }
interface J { C m(); }
class C implements I,J { C m(){ ... } }
class D implements I,J { C m(){ ... } }
// in a method:
I x;
if (...) x = new C(); else x = new D();
x.m(); // uses x as a value of I or J
```

Now consider the following class and advice declarations that replace creations of `C` and `D` objects with those of `E` and `F` objects, respectively.

```
class E implements I   { C m(){ ... } }
class F implements   J { C m(){ ... } }
E around(): call(C.new()) { return new E(); }//CtoE
F around(): call(D.new()) { return new F(); }//DtoF
```

Application of both advice declarations is not correct because we cannot give a single target type for `x.m()`. Applications of either one of above two advice declarations are, however, safe.

Even though the inconsistency of the above advice declarations can be detected by checking and weaving for one by one join point shadow, there are cases where different processing orders of shadows will lead to different results. Consider the following declarations and the same base method as above.

```
interface K { C m(); }
class G implements I,K { C m(){ ... } }
class H implements   K { C m(){ ... } }
G around(): call(C.new()) { return new G(); }//CtoG
H around(): call(D.new()) { return new H(); }//DtoH
```

It is safe to apply the advice marked as `CtoG` to the base program. The woven code is effectively equivalent to the following code fragment.

```
if (...) x = new G(); else x = new D();
x.m(); // uses x as a value of I
```

Then, it is now safe to apply the advice marked as `DtoH` to the woven code as we can use `K` as the target type for `x.m()`. However, it is not safe to apply the advice marked `DtoH` first because we cannot give a single target type for `x.m()`, which is required in the Java bytecode language.

Therefore, our mechanism should check all join point shadows in a method at once, in order to detect inconsistency between shadows.

5. FORMAL CORRECTNESS OF TYPE RELAXED WEAVING

We formalize a core of type relaxed weaving to confirm its type soundness. Rather than formalizing the entire weaving mechanism, our approach here focuses on replacement of several expressions in a method body with the ones that may have different types. We believe that it sufficiently covers major issues in the type relaxed weaving because around advice can be basically considered as substitution of expressions. Further issues that should be addressed in practical weavers are discussed at the end of this section.

Below, we first set up a simple object-oriented language called Featherweight Java for Relaxation (FJR), which is an extension of Featherweight Java (FJ)[9]. Its type system, which we believe sound with respect to operational semantics, corresponds to bytecode verification in the Java Virtual Machine. Then, we give an algorithm $\mathcal{G}$ to generate subtyping constraints from a given expression and a type environment—types for free variables in the expression. Finally, we prove that, if the subtyping constraints have a solution, then the given expression is well typed under the given type environment.

Our intention is to split the type checking process into two phases in future so as to fit the load-time weaving mechanism available in many AOP systems: i.e., gathering typing information at compilation time, and then use the information later at weaving time.

Based on that intention, we designed that $\mathcal{G}$ will be given a method body where each join point shadow that a piece of around advice is woven into is replaced by a (fresh) variable. Each variable is given the advice's return type, which may be different from the type of the join point shadow. When the subtyping constraints that $\mathcal{G}$ generates have a solution, we can safely replace the variables with the bodies of the advice, since substitution of expressions for variables of the same types preserves well-typedness.

We show only a main part of definitions and the statements of theorems here; the full definition of the language is found in the full version of this paper[6].

5.1 Featherweight Java for Relaxation

5.1.1 Syntax

Featherweight Java for Relaxation has, in addition to most of the features in FJ, interface types (without interface extension), let expressions, and non-deterministic choice. However, for simplicity reasons, we remove typecasts and fields from objects, which can be easily restored without difficulty. Unlike FJ, where every expression is assigned a class name as its type, FJR has a restricted version of the union type system[8], which allows more liberal use of values as discussed in the last section.

[6]Available at `http://www.graco.c.u-tokyo.ac.jp/ppp/projects/typerelaxedweaving.en`.

The syntax rules of FJR are given as follows.

$$
\begin{array}{rcl}
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C\ implements\ \overline{I}\ \{\ \overline{M}\ \}} \\
\mathtt{M} & ::= & \mathtt{T\ m(\overline{T}\ \overline{x})\ \{\ return\ e;\ \}} \\
\mathtt{IF} & ::= & \mathtt{interface\ I\ \{\ \overline{\mathcal{S}}\ \}} \\
\mathcal{S} & ::= & \mathtt{T\ m(\overline{T}\ \overline{x});} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.m(\overline{e})} \mid \mathtt{new\ C(\overline{e})} \\
 & & \mid \mathtt{let\ x\ =\ e\ in\ e} \mid \mathtt{(?e{:}e)} \\
\mathtt{S,T} & ::= & \mathtt{C} \mid \mathtt{I} \\
\mathtt{U,V} & ::= & \mathtt{T} \mid \mathtt{U} \cup \mathtt{U}
\end{array}
$$

Following the convention of FJ, we use an overline to denote a sequence and write, for example, $\overline{\mathtt{x}}$ as shorthand for $\mathtt{x}_1,\ldots,\mathtt{x}_n$. The metavariables C and D range over class names; I and J range over interface names; m ranges over method names; and x and y range over variables, which include the special variable this.

CL is a class declaration, consisting of its name, a superclass name, interface names that it implements, and methods $\overline{\mathtt{M}}$; IF is an interface declaration, consisting of its name and method headers $\overline{\mathcal{S}}$. In addition to class declarations, FJR has interface declarations, which are needed to discuss the issues of target types (Sections 4.5 and 4.6) and advice interference (Section 4.8).

The syntax of expressions is extended from that of FJ, which already includes variables, method invocation, and object instantiation (and also field access and typecast, which are omitted here). We introduce let expressions to illustrate the cases when a value returned from around advice is used as values of different types. let is the only binding construct of an expression and the variable x in $\mathtt{let\ x\ =\ e_1\ in\ e_2}$ is bound in $\mathtt{e}_2$. We also introduce non-deterministic choice (?e:e) to handle the cases when a variable contains values of different types. Although a non-deterministic choice is not useful in practical programming, it is sufficient to express such a case. We define the set of free variables in an expression by a standard manner. We will denote a capture-avoiding substitution of expressions $\overline{\mathtt{e}}$ for variables $\overline{\mathtt{x}}$ by $[\overline{\mathtt{e}}/\overline{\mathtt{x}}]$.

S and T stand for simple types, i.e., class and interface names, and will be used for types written down in classes and interfaces. U and V stand for union types. For example, a local variable of type $\mathtt{C} \cup \mathtt{D}$ may point to either an object of class C or that of D.

An FJR program is a pair of a class table CT, which is a mapping from class/interface names to class and interface declarations, and an expression, which stands for the body of the main method. We denote the domain of a mapping by $dom(\cdot)$. We always assume a fixed class table, which is assumed to satisfy the following sanity conditions: (1) $CT(\mathtt{C}) = \mathtt{class\ C}\ \cdots$ for every $\mathtt{C} \in dom(CT)$; (2) $CT(\mathtt{I}) = \mathtt{interface\ I}\ \cdots$ for every $\mathtt{I} \in dom(CT)$; (3) $\mathtt{Object} \notin dom(CT)$; (4) for every simple type T (except Object) appearing anywhere in CT, we have $\mathtt{T} \in dom(CT)$; and (5) there are no cycles formed by extends clauses.

5.1.2 Lookup Functions

As in FJ, we use the functions, whose definitions are omitted, to look up method signatures and bodies in the class table. $mtype(\mathtt{m},\mathtt{T})$ returns a pair (written $\overline{\mathtt{S}}{\rightarrow}\mathtt{S}_0$) of the sequence of the argument types $\overline{\mathtt{S}}$ and the return type $\mathtt{S}_0$ of method m in simple type T (or its supertypes). We also use the function $mtype^{C}(\mathtt{m},\mathtt{C})$, which also returns the signature of m in C; unlike *mtype*, it looks up only superclasses and do not consider interfaces that the given class implements. $mbody(\mathtt{m},\mathtt{C})$ returns a pair (written $\overline{\mathtt{x}}.\mathtt{e}$) of the formal parameters $\overline{\mathtt{x}}$ and the body e of method m in class C. Since a method is declared only in classes, *mbody* takes only class names as its second argument. We assume Object has no methods and so neither $mtype(\mathtt{m},\mathtt{Object})$ nor $mbody(\mathtt{m},\mathtt{Object})$ is defined.

5.1.3 Type System

The subtyping relation is written U <: V. It includes extends and implements relations, and, for union types, we have the following rules:

$$
\mathtt{U}_1 \mathrel{<:} \mathtt{U}_1 \cup \mathtt{U}_2 \qquad\qquad \mathtt{U}_2 \mathrel{<:} \mathtt{U}_1 \cup \mathtt{U}_2
$$

$$
\frac{\mathtt{U}_1 \mathrel{<:} \mathtt{U}_3 \qquad \mathtt{U}_2 \mathrel{<:} \mathtt{U}_3}{\mathtt{U}_1 \cup \mathtt{U}_2 \mathrel{<:} \mathtt{U}_3}
$$

They mean that the union type $\mathtt{U}_1 \cup \mathtt{U}_2$ is the least upper bound of $\mathtt{U}_1$ and $\mathtt{U}_2$. Formally, the subtyping relation is given as the least set including the reflexive transitive closure of the union of extends and implements relations, closed by the three rules above.

The type judgment is of the form $\Gamma \vdash \mathtt{e} : \mathtt{U}$, read "expression e is given type U under type environment Γ." A type environment Γ, also written $\overline{\mathtt{x}}{:}\overline{\mathtt{U}}$, is a finite mapping from variables $\overline{\mathtt{x}}$ to types $\overline{\mathtt{U}}$. The typing rules are given below.

$$
\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-VAR)}
$$

$$
\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{U}_0 \qquad \mathtt{U}_0 \mathrel{<:} \mathtt{T}_0 \qquad mtype(\mathtt{m},\mathtt{T}_0) = \overline{\mathtt{T}}{\rightarrow}\mathtt{T} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} \mathrel{<:} \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{T}} \qquad \text{(T-INVK)}
$$

$$
\Gamma \vdash \mathtt{new\ C()} : \mathtt{C} \qquad \text{(T-NEW)}
$$

$$
\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{U}_1 \qquad \Gamma, \mathtt{x}{:}\mathtt{U}_1 \vdash \mathtt{e}_2 : \mathtt{U}_2}{\Gamma \vdash \mathtt{let\ x\ =\ e_1\ in\ e_2} : \mathtt{U}_2} \qquad \text{(T-LET)}
$$

$$
\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{U}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{U}_2}{\Gamma \vdash \mathtt{(?e_1{:}e_2)} : \mathtt{U}_1 \cup \mathtt{U}_2} \qquad \text{(T-CHOICE)}
$$

All rules are straightforward. In T-INVK, the simple type $\mathtt{T}_0$ corresponds to the receiver's static type recorded in the bytecode instruction invokevirtual or invokeinterface. Note that a simple name has to be chosen here. So, even if two classes C and D that extend Object and implement no interfaces happen to have a method m of the same signature, m cannot be invoked on the receiver of type $\mathtt{C}\cup\mathtt{D}$. T-LET allows local variables to have union types, whereas the method typing rule, given below, allows method parameters to have only simple types. In T-CHOICE, the choice expression is given the union of the types of the two subexpressions since its value is that of either $\mathtt{e}_1$ or $\mathtt{e}_2$.

The type judgment for methods is of the form M OK IN C, read "method M is well typed in class C." The typing rule is given as follows:

$$
\frac{\overline{\mathtt{x}}{:}\overline{\mathtt{T}}, \mathtt{this}{:}\mathtt{C} \vdash \mathtt{e} : \mathtt{U} \qquad \mathtt{U} \mathrel{<:} \mathtt{T}_0 \qquad \mathtt{class\ C\ extends\ D\ implements\ \overline{I}\ \{\cdots\}} \qquad override(\mathtt{m},\mathtt{D},\overline{\mathtt{T}}{\rightarrow}\mathtt{T}_0)}{\mathtt{T}_0\ \mathtt{m(\overline{T}\ \overline{x})\ \{\ return\ e;\ \}\ OK\ IN\ C}} \qquad \text{(T-METH)}
$$

The method body $\mathtt{e}$ has to be well typed under the type declarations of the parameters $\overline{\mathtt{x}}$ and the assumption that $\mathtt{this}$ has type $\mathtt{C}$. As mentioned above, the parameter types and return type have to be simple types. The predicate *override*, whose definition is omitted, checks whether $\mathtt{M}$ correctly overrides the method of the same name in the superclass $\mathtt{D}$ (if any).

Finally, the type judgment for classes is written $\mathtt{C\ OK}$, meaning "class $\mathtt{C}$ is well typed," and the typing rule is given as follows:

$$\frac{\begin{array}{c}\overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}\\ \forall \mathtt{m},\mathtt{I}\in\overline{\mathtt{I}}.(\mathit{mtype}(\mathtt{m},\mathtt{I})=\overline{\mathtt{T}}{\rightarrow}\mathtt{T}_0)\Longrightarrow(\mathit{mtype}^{C}(\mathtt{m},\mathtt{C})=\overline{\mathtt{T}}{\rightarrow}\mathtt{T}_0)\end{array}}{\mathtt{class\ C\ extends\ D\ implements\ }\overline{\mathtt{I}}\ \{\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}}\quad\text{(T-CLASS)}$$

It means that all methods have to be well typed and all methods declared in $\overline{\mathtt{I}}$ have to be implemented (or inherited) in $\mathtt{C}$ with the correct signature. A program is well typed when all classes in the class table are well typed and the main expression is well typed under the empty type environment.

5.1.4 Operational Semantics

The reduction relation is of the form $\mathtt{e}\longrightarrow\mathtt{e}'$, read "expression $\mathtt{e}$ reduces to expression $\mathtt{e}'$ in one step." We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. Reduction rules are given below:

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C})=\overline{\mathtt{x}}.\mathtt{e}_0}{\mathtt{new\ C().m(}\overline{\mathtt{e}}\mathtt{)}\longrightarrow[\overline{\mathtt{e}}/\overline{\mathtt{x}},\mathtt{new\ C()/this}]\mathtt{e}_0}$$

$$\mathtt{let\ x\ =\ e_1\ in\ e_2}\longrightarrow[\mathtt{e}_1/\mathtt{x}]\mathtt{e}_2 \qquad (\mathtt{?e_1:e_2})\longrightarrow\mathtt{e}_i$$

We omit congruence rules, which allow a subexpression to reduce.

5.1.5 Properties

FJR enjoys the standard type soundness properties consisting of subject reduction and progress[20]. We show their statements with Substitution Lemma, which states that substitution preserves typing. Since the language is mostly a subset of FJ∨[8], their proofs, not shown here, are very similar to (and, in fact, easier than) those for FJ∨. Here, we write $\Gamma\vdash\overline{\mathtt{e}}:\overline{\mathtt{U}}$ for the sequence of type judgments $\Gamma\vdash\mathtt{e}_1:\mathtt{U}_1,\ldots,\Gamma\vdash\mathtt{e}_n:\mathtt{U}_n$. Similarly for $\overline{\mathtt{U}}$ <: $\overline{\mathtt{V}}$.

LEMMA 1 (SUBSTITUTION LEMMA). *If* $\Gamma,\overline{\mathtt{x}}{:}\overline{\mathtt{U}}\vdash\mathtt{e}:\mathtt{U}_0$ *and* $\Gamma\vdash\overline{\mathtt{e}}:\overline{\mathtt{V}}$ *and* $\overline{\mathtt{V}}$ <: $\overline{\mathtt{U}}$*, then there exists some type* $\mathtt{V}_0$ *such that* $\Gamma\vdash[\overline{\mathtt{e}}/\overline{\mathtt{x}}]\mathtt{e}:\mathtt{V}_0$ *and* $\mathtt{V}_0$ <: $\mathtt{U}_0$.

THEOREM 1 (SUBJECT REDUCTION). *If* $\Gamma\vdash\mathtt{e}:\mathtt{U}$ *and* $\mathtt{e}\longrightarrow\mathtt{e}'$*, then there exists some type* $\mathtt{U}'$ *such that* $\Gamma\vdash\mathtt{e}':\mathtt{U}'$ *and* $\mathtt{U}'$ <: $\mathtt{U}$.

THEOREM 2 (PROGRESS). *If* $\Gamma\vdash\mathtt{e}:\mathtt{U}$*, then* $\mathtt{e}$ *is either* $\mathtt{new\ C()}$ *for some* $\mathtt{C}$ *or there exists some expression* $\mathtt{e}'$ *such that* $\mathtt{e}\longrightarrow\mathtt{e}'$.

5.2 Constraint Generation

We give the algorithm called $\mathcal{G}$ to generate subtyping constraints from a given pair of an expression and a type environment, after giving preliminary definitions.

First, we extend the syntax of types so that they include type variables.

$$\mathtt{S},\mathtt{T} ::= \cdots \mid \mathtt{X}$$

$\mathcal{G}(\Gamma,\mathtt{x})=(\emptyset,\Gamma(\mathtt{x}))$

```
$\mathcal{G}(\Gamma,\mathtt{let\ x\ =\ e_1\ in\ e_2})=$
    let $(\mathcal{R}_1,\mathtt{U}_1)=\mathcal{G}(\Gamma,\mathtt{e}_1)$ in
    let $(\mathcal{R}_2,\mathtt{U}_2)=\mathcal{G}((\Gamma,\mathtt{x}{:}\mathtt{U}_1),\mathtt{e}_2)$ in
    $(\mathcal{R}_1\cup\mathcal{R}_2,\mathtt{U}_2)$

$\mathcal{G}(\Gamma,\mathtt{e}_0\mathtt{.m(e_1,}\cdots\mathtt{,e}_n\mathtt{)})=$
    let $(\mathcal{R}_0,\mathtt{V})=\mathcal{G}(\Gamma,\mathtt{e}_0)$ in
    let $(\mathcal{R}_1,\mathtt{U}_1)=\mathcal{G}(\Gamma,\mathtt{e}_1)$ in
       $\vdots$
    let $(\mathcal{R}_n,\mathtt{U}_n)=\mathcal{G}(\Gamma,\mathtt{e}_n)$ in
    let $\overline{\mathtt{T}}\rightarrow\mathtt{T}=\mathit{mtype}(\mathtt{m},\mathit{typeOf}(\mathtt{e}_0))$ in
    $(\mathcal{R}_0\cup\mathcal{R}_1\cup\cdots\cup\mathcal{R}_n\cup\{\overline{\mathtt{U}}$ <: $\overline{\mathtt{T}}\}$
      $\cup\{\mathtt{V}$ <: $\mathtt{X},\mathtt{X}$ <: $\bigcup\mathit{mdeftypes}(\mathtt{m},\mathit{typeOf}(\mathtt{e}_0))\},$
    $\mathtt{T})$
    (for fresh $\mathtt{X}$)

$\mathcal{G}(\Gamma,\mathtt{new\ C()})=(\emptyset,\mathtt{C})$

$\mathcal{G}(\Gamma,(\mathtt{?e_1:e_2}))=$
    let $(\mathcal{R}_1,\mathtt{U}_1)=\mathcal{G}(\Gamma,\mathtt{e}_1)$ in
    let $(\mathcal{R}_2,\mathtt{U}_2)=\mathcal{G}(\Gamma,\mathtt{e}_2)$ in
    $(\mathcal{R}_1\cup\mathcal{R}_2,\mathtt{U}_1\cup\mathtt{U}_2)$
```

$\bigcup\{\mathtt{T}_1,\ldots,\mathtt{T}_n\}$ stands for the union type $\mathtt{T}_1\cup\cdots\cup\mathtt{T}_n$.

Figure 2: Constraint generation algorithm $\mathcal{G}$.

(The syntax rule for union types remains the same.) We will denote a substitution of simple types $\overline{\mathtt{T}}$ for type variables $\overline{\mathtt{X}}$ by $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$ and use the metavariable S for such type substitutions. We say a type substitution is *ground*, if every type variable in its domain is mapped to either a class or interface name (not a type variable).

A subtyping constraint, or simply a constraint, is an inequality of the form $\mathtt{U}$ <: $\mathtt{V}$. A ground type substitution S is called a solution of the set $\{\overline{\mathtt{U}}$ <: $\overline{\mathtt{V}}\}$ of subtyping constraints if and only if for each constraint $\mathtt{U}_i$ <: $\mathtt{V}_i$, the subtyping relation $(S\mathtt{U}_i)$<:$(S\mathtt{V}_i)$ is derivable.

Now, we give the definition of $\mathcal{G}$, which takes a type environment Γ and an expression $\mathtt{e}$ and returns a set $\mathcal{R}$ of constraints and a type $\mathtt{U}$. The expression is assumed to be well typed under some type environment, which may or may not be the same as Γ, in order to know the signatures of methods invoked in $\mathtt{e}$. We denote the type of a subexpression $\mathtt{e}$ by *typeOf*($\mathtt{e}$). We also need an auxiliary function *mdeftypes*($\mathtt{m},\mathtt{T}$) to collect the set of $\mathtt{T}$'s supertypes that have $\mathtt{m}$. A formal definition is given as follows:

$$\mathit{mdeftypes}(\mathtt{m},\mathtt{T})=\{\mathtt{S}\mid\mathtt{T}\ \text{<:}\ \mathtt{S}\ \text{and}\ \mathit{mtype}(\mathtt{m},\mathtt{S})\ \text{defined}\}$$

Then, $\mathcal{G}$ is defined in Figure 2. The algorithm is obtained basically by reading typing rules in a bottom-up manner. The only interesting case is one for method invocations. The type variable $\mathtt{X}$ stands for the receiver type, which has to be a supertype of the expression $\mathtt{e}_0$ at the receiver position, hence we have an inequality $\mathtt{V}$ <: $\mathtt{X}$. $\mathtt{X}$ also has to have method $\mathtt{m}$ of the same signature $\overline{\mathtt{T}}\rightarrow\mathtt{T}$ as in the original typing. The inequation $\mathtt{X}$ <: $\bigcup\mathit{mdeftypes}(\mathtt{m},\mathit{typeOf}(\mathtt{e}_0))$ represents this condition.

If the constraints generated by $\mathcal{G}(\Gamma,\mathtt{e})$ has a solution, $\mathtt{e}$

can be well typed also under Γ, which may be different from e's original type environment:

THEOREM 3. *If* $\Gamma \vdash \mathtt{e} : \mathtt{U}$ *and* $dom(\Gamma) \subseteq dom(\Gamma')$ *and* $\mathcal{G}(\Gamma', \mathtt{e}) = (\mathcal{R}, \mathtt{U}')$ *and* $\mathcal{R}$ *has a solution* S*, then* $S\Gamma' \vdash \mathtt{e} : S\mathtt{U}'$.

The theorem is proved by easy induction on e.

5.3 Soundness of Type Relaxed Weaving

We finally argue that type relaxed weaving is sound, that is, woven programs do not break type safety, using the machinery introduced above. Here, we consider a rather simple case, where advice neither take parameters nor call `proceed()` and for each join point at most one piece of advice is applied, so that weaving is expressed by simple replacement of subexpressions with other closed expressions. We discuss about parameters and `proceed()` later and leave a more rigorous treatment to future work. (Our implementation supports parameter substitution and `proceed()`, though.)

Given a method definition $\mathtt{T}_0$ `m(`$\overline{\mathtt{T}}$ `x) { return e; }` in class C and advice bodies $\overline{\mathtt{e}}$ (which have type $\overline{\mathtt{S}}'$, respectively, under the empty type environment), we first replace subexpressions where weaving occurs with fresh variables $\overline{\mathtt{y}}$ and obtain $\mathtt{e}'$. Then, the method body after weaving will be expressed as $[\overline{\mathtt{e}}/\overline{\mathtt{y}}]\mathtt{e}'$. The intermediate expression $\mathtt{e}'$ can be typed as

$$\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}, \mathtt{this:C}, \overline{\mathtt{y}}\mathtt{:}\overline{\mathtt{S}} \vdash \mathtt{e}' : \mathtt{U}_0$$

where $\mathtt{U}_0$ <: $\mathtt{T}_0$. Then, we generate constraints by computing $\mathcal{G}((\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}},\ \mathtt{this:C},\ \overline{\mathtt{y}}\mathtt{:}\overline{\mathtt{S}}'), \mathtt{e}') = (\mathcal{R}, \mathtt{U}_0{}')$. Notice that the types for $\overline{\mathtt{y}}$ have been changed from $\overline{\mathtt{S}}$ to $\overline{\mathtt{S}}'$. Then, we check if $\mathcal{R} \cup \{\mathtt{U}_0{}' <: \mathtt{T}_0\}$ has a solution. If it has a solution S, then type relaxed weaving is allowed to proceed. Indeed, by Theorem 3, we have

$$\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}, \mathtt{this:C}, \overline{\mathtt{y}}\mathtt{:}\overline{\mathtt{S}}' \vdash \mathtt{e}' : S\mathtt{U}_0{}'$$

and $S\mathtt{U}_0{}'$ <: $\mathtt{T}_0$ (notice that $\overline{\mathtt{T}}$, C, and $\overline{\mathtt{S}}'$ do not contain type variables, so S does not change them). Finally, by Lemma 1, the method body $[\overline{\mathtt{e}}/\overline{\mathtt{y}}]\mathtt{e}'$ and so the whole method after weaving are also well typed.

We can easily extend this argument to the case in which more than one advice body is woven. Since we do not take advice precedence into account, we simply take the union of all the return types of possible advice bodies as $\mathtt{S}'_i$ in the argument above. Similarly, for advice with a dynamic pointcut, the type of the original expression $\mathtt{S}_i$ should be added to the union.

As discussed in Section 4.7, advice bodies have to be checked in order to deal with `proceed()`. The constraint generation for advice bodies will be similar to the one for base methods. It suffices to give `proceed()` the union of the return types of all other pieces of advice and the base method when generating constraints.

Finally, parameters to advice are also easy to deal with, since, in the above type soundness argument, we can regard the expressions $\overline{\mathtt{e}}$ substituted for $\overline{\mathtt{y}}$ as the advice bodies whose parameters are already replaced with appropriate actual arguments. This replacement is also type preserving thanks to Substitution Lemma.

6. IMPLEMENTATION

We implemented an AspectJ compatible compiler that supports the type relaxed weaving, called *RelaxAJ*, which is publicly available[7]. The implementation is based on an existing AspectJ compiler (`ajc` version1.6.1), and modified `ajc`'s weaving algorithm. Since the difference is only in between RULE 2 and RULE 3, we merely needed to modify a few methods in the original implementation.

The modified compiler works in the following ways. After compiling class and aspect declarations into bytecode class formats, it visits all methods in all classes provided. For each method, our weaver first accumulates all pieces of around advice applicable to any join point shadows within the method. When there are any piece of advice that violates original compiler's type-checking rule (i.e., RULE 2), our weaver performs its own type-checking based on RULE 3.

When it type-checks, the bodies of advice are woven into the bytecode instructions same as done by the original weaver. Finally, when the type-checker identifies changes in target types (as discussed in Section 4.5), it changes the method invocation instructions and signatures appropriately. It also removes runtime type-checking (i.e., cast) instructions.

Our type-checker implements the typing rules and the constraint generating function $\mathcal{G}$ presented in Section 5. In order to cope with full-set of Java bytecode instructions, which include branching ones, we implemented the algorithm as abstract interpretation.

The implementation is approximately 2,200 lines of additional code to the original AspectJ complier. The additional type-checking adds relatively small amount of time to compilation time of a practical application program. When we compiled JHotDraw version 7.1, which consists of 441 classes or 93,000 lines of code, with a wrapper inserting aspect that has advice declarations similar to the one in Listing 4, the compilation time increased 2.41 percent (from 6.411 seconds to 6.566 seconds on the Sun HotSpot VM version 1.5.0 executed by two 2.8 GHz Quad-Core Intel Xeon processors with 4 GB memory, under Mac OS X version 10.5) from the case compiled by `ajc` with a dummy advice[8]. In this experiment, the advice body was woven to 51 join point shadows. Of course, the overheads would become larger when a piece of around advice that requires type relaxed weaving were woven to more methods.

7. RELATED WORK

7.1 StrongAspectJ

StrongAspectJ is an extension to AspectJ that supports generic advice declarations in a type-safe manner[6]. As mentioned in the first section, the genericity offered by StrongAspectJ is similar to, but different from the one offered by the type relaxed weaving.

Let's see similarity and difference by using the example in Listing 6, which is taken from (but slightly modified for explanatory purposes) StrongAspectJ's paper[6]. In Java, `Integer` and `Float` are both subtypes of the `Number` interface, which requires the `intValue` method.

Interestingly, the advice declaration, which is rejected by AspectJ, is accepted both by StrongAspectJ (modulo slight modifications to the advice declaration) and the relaxed type

[7]`http://www.graco.c.u-tokyo.ac.jp/ppp/projects/typerelaxedweaving.en`

[8]We declared the return type of the advice as `Object` in order to get the advice compiled by AspectJ. With this advice, `ajc` can generate woven code, which cause runtime cast errors.

```
Integer calcInteger() { return new Integer(...); }
Float   calcFloat()   { return new Float  (...); }

int compute() {
  Integer i = calcInteger();
  Float   f = calcFloat  ();
  return i.intValue() + f.intValue();
}

Number around():call((Integer||Float) calc*(..)) {
  Number n = proceed();
  while (n.intValue() > 100)
    n = proceed();
  return n;
}
```

Listing 6: An around advice declaration that is applied to join points of different types.

weaving. With StrongAspectJ, we can redefine the advice by using a type variable, so that the type system can check correctness of the advice body with respect to *the type of each join point* where the advice is applied to. With the type relaxed weaving, the advice happens to be accepted because *the return values from the join points are used* merely as the `Number` objects.

Difference becomes apparent when we modify either the advice or the `compute` method. If we modify the body of the advice so that it returns, for example, `new Double(0)`, StrongAspectJ cannot accept such a piece of advice. Alternatively, if we modify the `compute` method so that it calls a method that is not defined in `Integer` class on `f`, the type relaxed weaving cannot accept the advice any longer.

We believe that the type systems of StrongAspectJ and the type relaxed weaving complement each other, and are currently designing an AspectJ language extension that supports both mechanisms.

7.2 Type Systems for Around Advice

As far as the authors know, all formalizations of around advice are based on RULE 2; i.e., advice types are checked against types in join points, if they formalized type systems. Wand, Kiczales and Dutchyn formalized behavior of around advice without types[18]. Clifton and Leavens formalized the proceed mechanism of around advice and its type safety[2]. Their type system is based on the one in AspectJ, which is based on RULE 2.

AspectML[4] and Aspectual Caml[14] are AOP extensions to functional languages. Those languages support *polymorphic advice*, which can be applied to join points that have polymorphic types. Although polymorphic types might give similar genericity, we believe that our work first pointed out the problem in practice, and formalized in a language with a subtyping relation.

8. CONCLUSION

This paper presented the type relaxed weaving, a novel weaving mechanism for around advice in statically typed AOP languages. With the type relaxed weaving, we can define around advice that replaces a value in a join point with the one of a different type, as long as the usage of the value in subsequent computation agrees.

Our contributions are: we (1) pointed out that the problem of the type-checking rules on around advice in existing AOP languages, (2) proposed the type relaxed weaving, which resolves the problem in a type safe manner, (3) formalized the core part of the mechanism in order to show type soundness, and (4) implemented the weaving mechanism as an AspectJ compatible compiler, which is publicly available.

We are extending our compiler so that it will allow around advice to provide values of different types to `proceed()`, as it allows to provide values to return from the advice. As for formalization, we plan to incorporate an advice weaving mechanism into our formalization so that we can express subtleties in `proceed()`. We are also designing a language *StrongRelaxAJ*, which is a hybrid of StrongAspectJ and RelaxAJ by taking advantages of both mechanisms.

Acknowledgements

An earlier version of the paper is presented at the ADI'08 workshop[13] and the Workshop on Programming and Programming Languages (PPL'09), at which the authors received valuable input and feedback. The authors would like to thank the members of the PPP Group at the University of Tokyo, the members of the Kumiki 2.0 Project, and the anonymous reviewers for their comments.

9. REFERENCES

[1] R. Bodkin. Performance monitoring with AspectJ. AOP@Work, Sept. 2005.

[2] C. Clifton and G. T. Leavens. MiniMAO1: Investigating the semantics of proceed. *Science of Computer Programming*, 63(3):321–374, 2006.

[3] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proc. of AOSD'04*, pp.56–65, 2004.

[4] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *TOPLAS*, 30(3):1–60, 2008.

[5] M. Erwig and D. Ren. Type-safe update programming. In *Proc. of ESOP'03*, pp.269–283, 2003.

[6] B. D. Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In *Proc. of AOSD'08*, pp.60–71, 2008.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Prentice Hall, 3rd ed., 2005.

[8] A. Igarashi and H. Nagira. Union types for object-oriented programming. In *Proc. of SAC'06*, pp.1435–1441, 2006.

[9] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *Proc. of OOPSLA'99*, pp.132–146, 1999.

[10] G. Kiczales, et al. An overview of AspectJ. In *Proc. of ECOOP'01*, pp.327–353. 2001.

[11] R. Lämmel. A semantical approach to method-call interception. In *Proc. of AOSD'02*, pp.41–55. 2002.

[12] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[13] H. Masuhara. On type restriction of around advice and aspect interference. In *Proc. of Workshop on Aspects,*

Dependencies and Interactions (ADI'08), 2008.

[14] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proc. of ICFP'05*, pp.320–330, 2005.

[15] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proc. of AOSD'03*, pp.90–99, 2003.

[16] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proc. of AOSD'03*, pp.120–129. 2003.

[17] O. Spinczyk, A. Gal, and W. Schroder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proc. of TOOLS Pacific'02*, pp.18–21, 2002.

[18] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, Sept. 2004.

[19] D. Wiese, R. Meunier, and U. Hohenstein. How to convince industry of AOP. In *Proc. of Industry Track at AOSD'07*, 2007.

[20] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.