On Type Restriction of Around Advice and Aspect Interference [*]

Hidehiko Masuhara

Graduate School of Arts and Sciences, University of Tokyo
`masuhara@acm.org`

Abstract. Statically typed AOP 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. This paper discusses language design issues to support the type relaxed weaving in AOP languages.

1 Advice Mechanism in AspectJ

The advice mechanism in aspect-oriented programming (AOP) languages is a powerful means of modifying behavior of a program without changing the program text. AspectJ[6] is one of the most widely-used AOP languages that support 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 enforcement[1, 3, 10, 12].

One of the unique features of the advice mechanism is the *around advice*, which can change parameter and return values of join points (i.e., specific kinds of events during program execution including method calls, constructor calls and field accesses). With around advice, it becomes possible to define such aspects that directly affect values passed in the program, including caching results, pooling resources and encrypting parameters. In object-oriented programming, around advice is also useful to modify functions of a system that are represented as objects by inserting proxies and wrappers, and by replacing with objects that offer different functionality.

1.1 Example of Around Advice

We first show a typical usage of around advice by taking a code fragment (Fig. 1) in a graphical drawing application[1] that stores graphical data into a file, which

---

[*] In Proceedings of the 3rd International Workshop on Aspects, Dependencies and Interactions (ADI'08), Paphos, Cyprus, July 2008.

[1] The code fragment is taken from JHotDraw version 6.0b1, but simplified for explanatory purpose.

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

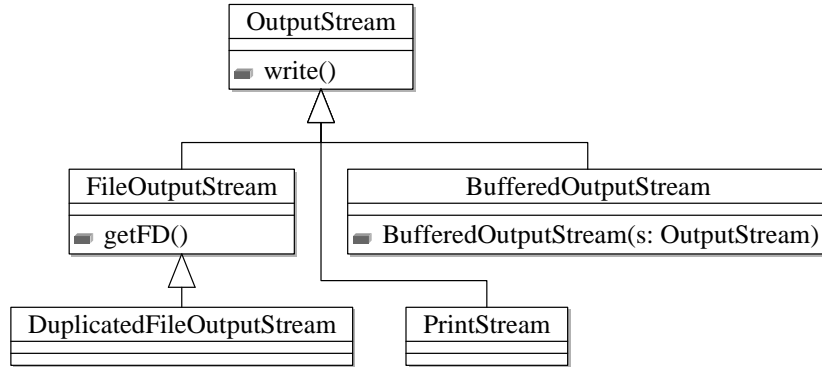**Fig. 1.** A code fragment that stores graphical data into a file.

Fig. 2. UML class diagram of the classes that appear in the examples.

is executed when the user selects the save menu. The hierarchy of the relevant classes is summarized in Fig. 2.

Assume we want to duplicate every file output to the console for debugging purpose. This can be achieved by first defining a subclass of `FileOutputStream` like Fig. 3, and editing the second line of the `store` method (Fig. 1) into the next one.

```
OutputStream s = new DuplicatedFileOutputStream(fileName);
```

Instead of textually editing the `store` method, we can define an aspect that creates `DuplicatedFileOutputStream` objects instead of `FileOutputStream`, as shown in Fig. 4. By using the aspect, we can replace creations of objects that are performed in many places in the program. In fact, there are several classes in JHotDraw that create `FileOutputStream` in order to support different file formats. The aspect definition can therefore achieve more modular modification.

2 The Problem: Restriction on Types of Around Advice

2.1 Restrictions for Type Safety

AspectJ guarantees type safety of around advice by type checking the body of around advice with respect to its parameter and return types, and by checking

```
class DuplicatedFileOutputStream extends FileOutputStream {
  DuplicatedFileOutputStream(String filename) { super(filename); }
  void write(int b) { super.write(b); System.out.write(b); }
  ...override other methods that write data...
}
```

Fig. 3. A class that duplicates all file outputs to the console.

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

Fig. 4. An aspect that replaces all creations of `FileOutputStream` with `DuplicatedFileOutputStream`.

the return type of the around advice with respect to the return types of join points which the advice is woven into.

The type checking rules can be summarized as follows. Given an around advice declaration with return type `T`, pointcut `P`, and a return statement with expression `e`:

```
T around(): P { ... return e; ... }
```

the program must satisfies the next two rules.

(AJ1) The type of `e` must be a subtype[2] of `T`.
(AJ2) For each join point matching `P`, `T` must be a subtype of its return type.

For example, the next around advice is rejected by **AJ1** because the type of `System.out`, namely `PrintStream`, is not a subtype of `FileOutputStream`.

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

The next advice declaration is rejected by **AJ2**. The pointcut of the advice matches a join point that corresponds to an expression `new FileOutputStream(...)` in `store` (Fig. 1). Since `String` is not a supertype of `FileOutputStream`, the advice violates **AJ2**.

```
String around():                              //weaving error
    call(FileOutputStream.new(String)) {
  return "Hello!";
}
```

---

[2] Here, we assume the subtype and supertype relations are reflexive; i.e., For any `T`, `T` is a subtype and supertype of `T`.

```
aspect Redirection {
  PrintStream around(): call(FileOutputStream.new(String)) {
      return System.out; // of type PrintStream
  }
}
```

Fig. 5. An aspect that returns the `PrintStream` object (i.e., the console) instead of creating `FileOutputStream`. While current AspectJ compilers reject this aspect as type incompatible at weaving, our type relaxed weaving accepts this.

2.2 The Problem: Useful, yet Prohibited Advice

We found that the restrictions prohibit to define some useful advice. Assume we want to redirect the output to the console, instead of duplicating. This can be easily achieved by editing the second line of `store` (Fig. 1) into the next one.

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

This is type safe, because the field `System.out` is of type `PrintStream`, which is a subtype of `OutputStream`.

However, it is not possible to do the same replacement by using AspectJ's around advice. The aspect declaration in Fig. 5 seems to work at first glance, but is actually rejected by **AJ2**. This is because **AJ2** requires the return type of the matching join point (`FileOutputStream`) to be a supertype of the return type of the advice (`PrintStream`). Changing the return type of the advice into `FileOutputStream` does not work, because it will violate **AJ1**.

2.3 Most Specific Usage Type

The problem can be clarified by using the notion of *the most specific usage type* of a value. We define the *usage types* of a value as follows. When a value is used as a parameter or a receiver of a method or constructor, the usage type of the value is its static parameter or receiver type, respectively. When a value is returned from a method, the usage type is the return type of the method. The most specific usage type of a value is such `T` that `T` is a subtype of any usage type of the value, and when `T'` is a subtype of any usage type of the value, `T'` is a subtype of `T`.

For example, the return value from `new FileOutputStream(..)` in `store` (Fig. 1) is used only as a parameter to the `BufferedOutputStream` constructor. Therefore, its most specific usage type is the static parameter type of the constructor, namely `OutputStream`.

The most specific usage type of an expression's return value indicates the upper bound of return types of replacement expression. In other words, replacing the expression with another expression succeeds when the return type of the replacement expression is a subtype of the most specific usage type. In the `store`'s case, the `new FileOutputStream(..)` expression can be replaced with any expression that has a subtype of `OutputStream`.

4

By using the most specific usage type, the problem in the previous section can be stated as follows:

Problem. AspectJ prohibits an around advice declaration when its return type **T** *is not a subtype of the return type of a matching join point, even if* **T** *is a subtype of the most specific usage type of the return value of the join point.*

2.4 More Example of the Problem

The problem is not artificially crafted. It rather can be found more frequently. Another example is around advice that wraps handlers. Java programs frequently use anonymous class objects in order to define event handlers. For example, the next code fragment creates a button object, and then installs a listener object into the button object. The listener object belongs to an anonymous class that implements the `ActionListener` interface. The parameter type of the `addActionListener` method is `ActionListener`.

```
JButton b = new JButton();
b.addActionListener(
  new ActionListener () {
    public void actionPerformed(ActionEvent e) {...}
  }
);
```

Now, assume that we want to wrap the listener object with an object of `Wrapper` that also implements `ActionListener`. While textually inserting a constructor call around the `new` expression is type safe, the next aspect that does the same insertion violates **AJ2**.

```
aspect WrapActionListener {
  ActionListener around(): call(ActionListener+.new(..)) {
    ActionListener l = proceed();
    return new Wrapper(l); // Wrapper implements ActionListener
  }
}
```

Note that the pointcut captures any construction of objects that are of a subtype of `ActionListener` (cf. the plus sign after the class name means any subtype of the class).

2.5 Generality of the Problem

While the examples are about return types of around advice in AspectJ, the problem and proposed solution in the paper are not limited to them.

First, the problem is not limited to the return type of around advice. Since around advice can also replace values of parameters that are captured by `args`

and `target` pointcuts, the same problem arises at replacing those values by using the `proceed` mechanism.

Second, the problem is not limited to AspectJ. Statically-typed AOP languages that support around advice, such as CaeserJ[9] and AspectC++[11], should also have the same problems.

Third, the problem is not limited to AOP languages. The same problem would arise the language mechanisms that can intercept and replace values, such as method-call interception[7] and type-safe update programming[4].

3 Type Relaxed Weaving

3.1 Basic Idea

We propose a weaving mechanism, called *type relaxed weaving*, that solves the problem. The only difference from the original AspectJ is **AJ2**, whose new rule is shown below.

(AJ2′) For each join point matching P, T must be a subtype of its *most specific usage type* of its return value.

The type relaxed weaving accepts the aspect `Redirection` in Fig. 5 and `WrapActionListener` in Section 2.4. For the `Redirection`'s case, the return type of the advice (`PrintStream`) is a subtype of the most specific usage type of the join point (`OutputStream`). For the `WrapActionListener`'s case, the return type of the advice (`ActionListener`) is a subtype of the most specific usage type of the join point (`ActionListener`).

3.2 Design Issues

There are several design issues that need to be addressed in order to make the type relaxed weaving into a concrete language implementation.

Operations that "use" values. We define the next eight operations *use* a value. (1) Calling a method or a constructor with the value as a parameter. (2) Calling a method with the value as a target. (3) Returning from a method with the value. (4) Down-casting the value. (5) Accessing (i.e., either reading from or writing to) a field of the value. (6) Assigning the value into a field. (7) Assigning the value into an array. (8) Throwing an exception with the value.

Note that the operations do not include assignments to local variables. This is because, types of local variables may not be available at bytecode weaving. Also, excluding local variable assignments can give more opportunities to type relaxed weaving. For example, when local variable assignments are *not* considered as use of values, the most specific usage type of the constructor call to `FileOutputStream` in the next code fragment is `OutputStream`, which is otherwise `FileOutputStream`.

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

6

Usage type of an overridden method call. When a value is used as a target object of a method call, we regard it as a usage with type `T` where `T` is the most general supertype of the static type of the value that defines the method.

Note the usage type of a target object can be different from the target type in the method signature that the Java compiler gives. For example, to the method call `output.write(...)` in `store` (Fig. 1), recent Java compilers chooses `BufferedOutputStream.write(byte[])` as its method signature because `BufferedOutputStream` is the static type of `output`.

The usage type of the value of `output` in the call is, however, `OutputStream` because it is the most general type that defines `write`.

Extent of value tracking. When computing the most specific usage type of a value, we merely chase dataflow of values within a method. This is a connotation of our previous decision that regards parameter passing as the usage of a value.

There could be further type relaxation opportunities if we took dataflow across method calls into account. Assume that the return value from `new FileOutputStream(..)` is passed as a parameter to the constructor `BufferedOutputStream(OutputStream)` (as shown in Fig. 1), but the constructor used the parameter as a value of type `Object`. Then, it is theoretically safe to replace the value with the one of any subtype of `Object`.

We however did not choose this level of relaxation because its implementation requires inter-method dataflow analysis as well as changes of method signatures, which are not easy.

When a usage type does not match. There are two options for the weaver when it detects that the return type of around advice is not a subtype of the usage type of a matching join point. The one is to raise an error as the current AspectJ compilers do. The other is not to weave the advice at the join point; i.e., we will weave advice to only join points that have more general usage types than the return type of the advice. Our tentative choice is the former. We believe further experiences will reveal the advantages and disadvantages of those options.

3.3 Aspect Interaction

When more than one aspect interacts, i.e., more than one around advice declaration is applicable to one join point, care must be taken about type safety.

Assume there are two around advice declarations: the one is in the `Redirection` aspect (Fig. 5), and the other is as follows.

```
FileOutputStream around(): call(FileOutputStream.new(String)) {
  FileOutputStream s = proceed();
  ... s.getFD() ... // getFD() is defined only in FileOutputStream
  return s;
}
```

Note that the advice uses the return value from `proceed` as the one of type `FileOutputStream`.

When those advice declarations are applied to a constructor call to `FileOutputStream` (e.g., the second line of `store` in Fig. 1), it is not safe depending on the advice precedence. If the latter advice precedes the former, the latter receives a value that is replaced by the former: i.e., a `PrintStream` object. The method call at the third line in the latter advice declaration then fails because the method is defined only in `FileOutputStream`. If the former precedes the latter, there is no problem because the former does not proceed further.

To cope with this interaction problem, we need to take the usage in advice bodies into account. Given a join point, we regard that its return value is used by advice bodies in addition to the operations listed in Section 3.2. In the above example, the latter advice uses the return value from `proceed` as `FileOutputStream`. Therefore, the most specific usage type of the join point becomes `FileOutputStream`, which correctly detects the problem of combining those two advice declarations.

3.4   Implementation

We are implementing a system that allows the type relaxed weaving. The implementation is based on existing AspectJ compilers extended with a modified weaving algorithm. Since the difference is only in between **AJ2** and **AJ2′**, the modification should be minimal.

The implementation consists of two parts, namely an analyzer that computes the most specific usage type of a given expression, and a judgment routine that decides whether a weaver can apply an around advice body to a join point.

The analyzer can be implemented as a bytecode verifier for the Java bytecode language that checks type safety of a method given in a bytecode format. The difference from standard bytecode verifier is that it performs type inference with the return type of a specified method call unknown. Since we decided to track dataflow merely within a method, it is not difficult to implement the analysis.

The judgment routine is invoked when a weaver is to apply a set of applicable advice declarations to a join point. When the set does not include around advice, it proceeds as usual. When it does, it first computes the most specific usage type of the join point, as well as the most specific usage type of the body of each applicable advice. It then finds the greatest type `T` that is a subtype of all the most specific usage types. Finally, it allows advice weaving when the return type of the around advice is a subtype of `T`.

4   Preliminary Feasibility Assessment

Before implementing the system, we estimated the number of opportunities in practical programs that can benefit from the type relaxed weaving. We monitored executions of five medium-sized Java programs, and classified the join point

shadows (i.e., source code locations) based on their most specific usage types. We used AspectJ for monitoring program executions.

The evaluated programs and the results are summarized in Table 1. The 'more general' row shows the numbers of shadows whose most specific usage type are strict supertypes of the return types or parameter types of the shadows. They approximate the numbers of shadows that can benefit from the type relaxed weaving. As can be seen in the table, we found that approximately 15–30% are such shadows. Even though the numbers merely represent potential benefits, they suggest that the type relaxed weaving can be useful in practice.

5   Related Work

There are several researchers that work on the types of the advice mechanism in AspectJ-like languages.

Clifton and Leavens formalized the proceed mechanism of around advice and its type safety[2]. They directly work on the type system of around advice, but for formalizing the existing mechanism. Our work purposes to deregulate existing mechanism for enabling more useful around advice declarations.

StrongAspectJ offers an extended mechanism and a type system for around advice in AspectJ[5]. The purpose of the work is to support generic around advice without compromising type safety. It would be useful to define an around advice declaration applicable to many different join points with different types, as long as the advice body does not access the values obtained from join points. We believe our proposed mechanism complements their mechanism, as ours aims at supporting advice declarations that explicitly changes types of values exchanged between advice and join points.

Aspectual Caml is an AOP extension to the functional language Caml[8] that supports polymorphic pointcuts. Similar to StrongAspectJ, polymorphic pointcuts enable to weave advice declarations into join points with different types as long as the advice body does not access the values in join points.

Table 1. Characteristics of the measured programs and the results. The bottom four rows show the numbers (and their percentages in parentheses) of join point shadows whose return values or parameter values are used as the values of more general, more specific, incompatible or same types of the shadows.

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
more general(%)	177 (21)	315(17)	576 (16)	2,499(29)	650(19)
more specific(%)	37 (4)	70 (4)	170 (5)	974(11)	156 (4)
incompatible(%)	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)

6 Conclusion

This paper presented a problem of type restriction to around advice that prevents the programmer defining advice that replaces values into of different types, while the replacement is type safe when it is achieved by textual modification. To the problem, we proposed the notion of *the most specific usage type* and a novel weaving mechanism called the *type relaxed weaving*, which permits around advice to replace values with the ones of the most specific usage type. Our preliminary assessment showed that practical programs have 15–30% join points (per source code location basis) that can benefit from the type relaxed weaving.

We are currently implementing an extended AspectJ compiler that supports type relaxed weaving. The implementation will consist of the type inference algorithm in Java bytecode verifier, and a small modification to the weaving mechanism in an existing AspectJ compiler.

We also plan to formalize the mechanism so that we can prove type safety with the type relaxed weaving. We believe this is needed for ensuring the correctness of the algorithm in complicated cases, especially when more than one aspect interacts.

References

1. Ron Bodkin. Performance monitoring with AspectJ: A look inside the Glassbox inspector with AspectJ and JMX. AOP@Work, September 2005.
2. Curtis Clifton and Gary T. Leavens. MiniMAO1: Investigating the semantics of proceed. *Science of Computer Programming*, 63(3):321–374, December 2006.
3. Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 56–65, March 2004.
4. Martin Erwig and Deling Ren. Type-safe update programming. In *ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*, pages 269–283, 2003.
5. Bruno De Fraine, Mario Südholt, and Viviane Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 60–71, April 2008.
6. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. June 2001.

7. Ralf Lämmel. A semantical approach to method-call interception. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 41–55. April 2002.

8. Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proceedings of International Conference on Functional Programming (ICFP 2005)*, pages 320–330, September 2005.

9. Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. March 2003.

10. Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD2003)*, pages 120–129. March 2003.

11. Olaf Spinczyk, Andreas Gal, and Wolfgang Schroder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 18–21, February 2002.

12. Daniel Wiese, Regine Meunier, and Uwe Hohenstein. How to convince industry of AOP. In *Proceedings of Industry Track at AOSD.07*, March 2007.