A Per-type Instantiation Mechanism for Generic Aspects

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

Tomoyuki Aotani
Japan Advanced Institute of Science and Technology
aotani@jaist.ac.jp

Hidehiko Masuhara
University of Tokyo
masuhara@acm.org

Abstract

We propose a per-type instantiation mechanism for generic aspects. Though AspectJ supports generic aspects, which declare type parameters, we cannot declare aspects that are parametrized over both field types and return types of applied join points without manually concretizing the type parameters. Our mechanism creates automatically an instance of a generic aspect for each type of the applied join points.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Aspect-oriented programming, generic aspects, aspect instantiation

1. Introduction

This paper considers variability of aspects with respect to types. In other words, by making aspects more type-generic, we expect to support aspects compositional with a wider variety of base programs.

We enable aspects that are parametrized over types of advice and fields in aspects. An example is the `Flyweight` [4] (a variant of caching) aspect in Figure 1[1], which minimizes memory use and improves the performance of the program by reusing previously created objects.

The example demonstrates three advantages of our approach. First, the aspect is non-abstract. Therefore the programmers can apply the aspect by merely placing the aspect definition in the build path. Second, the aspect can be applied to join points more than one type. Third, the return type of the around advice and the type of the field are parametrized with `V`, which is declared by the enclosing aspect `Flyweight`. Therefore type safety is guaranteed.

```
1  aspect Flyweight<V> {
2    Map<Object, V> constructedObjects;
3    V around(Object o):
4    call(Course+.new(*)) && args(o) {
5      if (constructedObjects.containsKey(o))
6        return constructedObjects.get(o);
7      V newObject = proceed(o);
8      constructedObjects.put(o, newObject);
9      return newObject;
10   }
11 }
```

Figure 1. A generic and concrete `Flyweight` aspect

AspectJ and its extensions cannot support such an aspect. For example, generics in AspectJ [1], though able to define aspects with parameter types, require the programmer to manually instantiate type parameters. It is not easy to do so when the advice is applied to many join points with different types. Moreover, expressiveness of pointcuts are restricted. Pointcuts which specify join points of more than one return type cause type errors. StrongAspectJ [3] supports type parameters only in advice declarations.

We overcome these limitations by introducing `a per-type instantiation mechanism`, which automatically creates an aspect instance for each type of applicable join points. The rationale behind the approach is that these aspects are type-safe only if the values at the join points of different types are not mixed together.

The contributions of this paper are as follows:

- We point out that AspectJ and StrongAspectJ cannot parametrize an aspect over both field types and return types of advice unless the programmers manually instantiate type parameters. It is not easy to do so in general because the programmers have to exhaustively enumerate all the types of join points where advice is applied. In other words, generic aspects in AspectJ do not allow the programmers to use expressive pointcuts [2, 10].

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.

VariComp'12, March 26, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1101-4/12/03. . . $10.00

[1] We explain this aspect in detail in Sections 2 and 3.

```
1 class Course {
2   Course(Integer id) {/* database access */}
3   ...
4 }
```

Figure 2. A `Course` class definition

- We propose a per-type instantiation mechanism for concrete generic aspects (i.e., non-abstract generic aspects). This mechanism creates an aspect instance for each type of the applicable join points.
- We describe a weaving strategy for concrete generic aspects.

The problem is explained in Section 2. Section 3 proposes the per-type instantiation mechanism. Section 4 explains the weaving strategy for concrete generic aspects. Section 5 discusses the instanciation policy. After discussing related work in Section 6, Section 7 concludes the paper and lists future work.

2. Motivation

AspectJ cannot support aspects parametrized over both field types and return types of the applied join points, without manually instantiating type parameters. We show an aspect that implements the Flyweight pattern [4] as an example of such aspects. The pattern minimizes memory use and improves the performance of the program by reusing previously created objects.

2.1 Example: Flyweight Pattern

Suppose we want to optimize a course registration system in which the constructor of the `Course` class obtains information from a database (see Figure 2). As we found many construction of courses with the same ID, we apply the Flyweight pattern to reduce the number of database accesses.

Figure 3 shows a generic aspect that implements the Flyweight pattern. `constructedObjects` holds constructed objects (line 2), and the around advice takes over constructor calls (lines 3–10). The advice first checks whether an object has ever been created that takes the same constructor parameters (line 5). If so, the advice returns the previously created object (line 6). Otherwise, it creates a new object by calling `proceed`, stores the object with the given arguments into `constructedObjects`, and then returns the object (lines 7–9).

Note that generic aspects in AspectJ must be abstract. Therefore we need to declare a concrete aspect that extends `Flyweight<V>` (Figure 4) in order to implement the flyweight pattern for the `Course` class.

2.2 Problem

Manual type instantiation is not easy in general because the programmers have to exhaustively enumerate all the types of

```
1 abstract aspect Flyweight<V> {
2   Map<Object, V> constructedObjects;
3   V around(Object o):
4   call(V.new(*)) && args(o) {
5     if (constructedObjects.containsKey(o))
6       return constructedObjects.get(o);
7     V newObject = proceed(o);
8     constructedObjects.put(o, newObject);
9     return newObject;
10  }
11 }
```

Figure 3. A generic `Flyweight` aspect in AspectJ 5

```
1 aspect CourseFlyweight
2   extends Flyweight<Course> {}
```

Figure 4. A concrete aspect extending `Flyweight`

join points where advice is applied. In other words, generic aspects in AspectJ do not allow the programmers to use expressive pointcuts [2, 10] if the return type of the advice is parametrized.

Suppose the `Course` class has many subclasses and we want to implement the Flyweight pattern for all subclasses of the `Course` class. It never helps us to replace `call(V.new(*))` within `Flyweight<T>` (line 4 in Figure 3) with `call(V+.new(*))`, i.e.,

```
V around(...): call(V+.new(*)) && args(o){...}
```

This is because the return types of the matched join points and advice do not follow AspectJ's typing rule; the return type of around advice must be a subtype of the return types of the join points where it is applied.

3. Our Approach

To cope with the problem described above, we propose a per-type instantiation mechanism for concrete generic aspects. Our approach is based on *dynamic aspect instance creation* rather than *static code generation*; the mechanism creates an instance for each type of the applied join points. One of its advantages is easy integration with the weaving mechanism in AspectJ. The compiler/weaver does not need to know which types the generic aspect will be concretized, as in AspectJ. Therefore, our weaving algorithm is a small and straightforward extension of AspectJ's weaving algorithm.

In our approach, we can define the generic Flyweight aspect, as shown in Figure 1 without relying on abstract aspects and abstract pointcuts.

The approach introduces instance advice and static advice for concrete generic aspects, which are explained in Sections 3.3 and 3.4.

3.1 Instantiation Mechanism in AspectJ

We explain instantiation mechanisms in AspectJ before we explain the per-type instantiation mechanism. In AspectJ, the way an aspect is instantiated is specified at the aspect declaration. Aspect instances are automatically created. When a piece of advice is run, an aspect instance is automatically selected.

In the case of the pertarget `A` aspect shown below, an instance is created at a join point that is specified by the pointcut `pc()`, and associated with the target object of the join point. When a piece of advice of the `A` aspect is run, an aspect instance that is associated with the target object of the join point is selected.

```
aspect A pertarget(pc()) {...}
```

3.2 Per-type Instantiation Mechanism

Our mechanism creates an instance of a generic aspect for each type. When a piece of instance advice is about to run at a join point, the mechanism selects an instance according to the types of the join point[2].

Instance Creation An instance of a concrete generic aspect is created for each type before the instance is selected. If a type parameter of the aspect has an upper bound, an instance is created for each subtype of the upper bound. If the aspect has more than one parameter, an instance is created for each combination of types.

Instance Selection When a piece of advice runs, the mechanism automatically selects an aspect instance according to the types of the join point. The mechanism first identifies positions of the type parameters in the advice signature. It then obtains the types of the join point that correspond to the type parameters. Finally, it selects an aspect instance created for these types.

When the advice in Figure 1 is applied to a constructor call of the `Course` class, the mechanism confirms that the type parameter `V` of the `Flyweight` aspect is used as the return type of the advice. Since the return type of a constructor call of the `Course` class is `Course`, the mechanism selects the `Flyweight<Course>` instance.

3.3 Instance Advice

The difference between instance advice and advice in AspectJ is that the signature of the instance advice must use all type parameter types of the aspect. As mentioned in Section 3.2, instance selection depends on the positions of the type parameters in the advice signature. Therefore, the mechanism cannot select an instance if all type parameters are not used in the signature of the instance advice. In such a case, the mechanism rejects an instance advice.

[2] By types of a join point, we mean the return type and the argument types of the join point

```
1 aspect Flyweight<V> {
2   static before(Object o):
3     call(Course+.new(*))
4       && args(o) {
5     if (o == null)
6       throw ...
7   }
8   ...
9 }
```

Figure 5. An example of static advice

3.4 Static Advice

We introduce static advice in order to declare advice independent of type parameters in a concrete generic aspect. There are cases where modularity is improved by declaring advice independent of type parameters in generic aspects. Static advice can be used to declare such advice. Static advice is not related to instances and cannot use the instance fields, instance methods, and type parameters of the aspect.

Suppose we want to define before advice that checks whether an argument of the constructor calls is null in the `Flyweight<V>` aspect. We cannot declare this advice as instance advice in the `Flyweight<V>` aspect because this advice is independent of the type parameter of the `Flyweight<V>` aspect. We can declare such advice in the `Flyweight<V>` aspect by using static advice (Figure 5).

4. Implementation

In this section, we explain the implementation strategy of our mechanism mentioned in Section 3.

In AspectJ, an aspect instance is selected by the `aspectOf` method when a piece of advice is run. This method is added automatically by a compiler, and the parameter types of this method depend on an instantiation model of the aspect. In our approach, this method receives `Class` objects, and returns an aspect instance whose type is parameterized with the types indicated by these `Class` objects.

When weaving advice in concrete generic aspects, a weaver first identifies the positions of the type parameters in the advice signature. It then obtains the types of the join point corresponding to the type parameters. If a type parameter has an upper bound and the type corresponding to the type parameter is not a subtype of that upper bound, the weaver reports an error. Otherwise, the weaver inserts code such that an instance of the aspect is obtained by the `aspectOf` method and that method that the advice is translated into is invoked.

In the case where a piece of advice in the `Flyweight<V>` aspect is woven into a constructor call of the `Course` class, the base code

```
new Course(...)
```

```
1 aspect Flyweight<V> {
2   private static
3     Map<Class<?>, Flyweight<?>> map;
4   public static <T>
5     Flyweight<T> aspectOf(Class<T> c) {
6     if (map.containsKey(c)))
7       return (Flyweight<T>)map.get(c);
8     Flyweight<T> newInstance
9       = new Flyweight<T>();
10    map.put(c, newInstance);
11    return newInstance;
12  }
13  ... // original definition
14 }
```

Figure 6. `aspectOf` method of `Flyweight<V>` aspect

is translated into

```
Flyweight.aspectOf(Course.class).advice1(...)
```

(`advice1` is the name of the method that the advice is translated into)

4.1 `aspectOf` method definitions

In our approach, the compiler adds not only an `aspectOf` method but also a `Map` field to hold instances of the aspect. The keys of the `Map` field are `Class` objects and the values are instances of the aspect. If an aspect has multiple type parameters, the keys of the `Map` field are lists of `Class` objects.

Figure 6 shows the `aspectOf` method definition of the `Flyweight<V>` aspect. The `map` field holds instances of the `Flyweight<V>` aspect (lines 2–3). The `aspectOf` method first checks whether an instance associated with the given argument is created (line 6). If so, the method returns the instance (line 7). Otherwise, a new instance is created and associated with the given `Class` object in `map`, then the method returns the instance (lines 8–11).

Figure 7 shows an example of an `aspectOf` method of an aspect with multiple type parameters. Such an aspect differs from an aspect with one type parameter in that the key of the `map` is a `List` object and the `aspectOf` method creates an `List` object with the given `Class` objects as the key (lines 6–9).

5. Discussion

In this section, we discuss cases where a type parameter of an aspect is used as an advice parameter type.

The advice of the `A<T>` aspect in Figure 8 is applied to constructor calls of the `BufferedOutputStream` class. We discuss which instance should be selected for a constructor call whose argument is a `FileOutputStream` object.

```
new BufferedOutputStream(
  new FileOutputStream(...))
```

```
1 aspect A<V1, ..., Vn> {
2   private static
3     Map<List, A<?, ..., ?>> map;
4   public static <T1, ..., Tn> A<T1, ..., Tn>
5     aspectOf(Class<T1> c1, ..., Class<Tn> cn) {
6     List l = new ArrayList();
7     l.add(c1);
8     ...
9     l.add(cn);
10    if (map.containsKey(l)))
11      return (A<T1, ..., Tn> )map.get(l);
12    A<T1, ..., Tn> newInstance
13      = new A<T1, ..., Tn>();
14    map.put(l, newInstance);
15    return newInstance;
16  }
17  ...
18 }
```

Figure 7. An `aspectOf` method of an aspect with multiple type parameters

```
1 aspect A<T> {
2   before(T t):
3   call(BufferedOutputStream.new(OutputStream))
4   && args(t) {...}
5 }
```

Figure 8. Using a type parameter as the parameter type of advice

If an instance is selected according to static types (the same as for the case of return types), an instance of type `A<OutputStream>` is selected.

However, advice parameter types indicate dynamic types of arguments. For example, the advice shown below is executed if a dynamic type of an argument is a subtype of `FileOutputStream`.

```
before(FileOutputStream t):
call(BufferedOutputStream.new(OutputStream))
&& args(t) {...}
```

From this point of view, an instance should be selected according to dynamic types. In this case, an instance of type `A<FileOutputStream>` is selected.

Our tentative choice is that the programmer can choose static types or dynamic types. In the case of the aspect shown below, an instance of type `A<OutputStream>` is selected.

```
aspect A<static T> {
  before(T t):
  call(BufferedOutputStream.new(OutputStream))
  && args(t) {...}}
```

In the case of the aspect shown below, an instance of type `A<FileOutputStream>` is selected.

```
aspect A<dynamic T> {
  before(T t):
  call(BufferedOutputStream.new(OutputStream))
  && args(t) {...}}
```

6. Related Work

As mentioned in Section 1, generics are introduced in AspectJ [1] and StrongAspectJ [3].

Several studies introduce generic advice similar to that in StrongAspectJ. Jagadeesan et al. introduced generic advice in AFGJ [8], which is an extension of Featherweight Generic Java [7]. Lohmann et al. introduced generic advice in AspectC++ [9].

Eos [11] and Association aspects [12] extend the instantiation mechanism of AspectJ. These mechanisms associate an instance of an aspect with values. Our mechanism associates an instance of an aspect with types.

AspectJ has a pertypewithin instance model. In this model, an instance of an aspect is selected according to the type that a join point is within. In our approach, an instance of an aspect is selected according to the types of join points.

Hannemann and Kiczales showed that implementing design patterns in AspectJ improves modurality [5]. They implemented the Flyweight pattern using an aspect as a factory; therefore, with their approach it is difficult to add a Flyweight pattern to a system. With our approach, it is easy to add a Flyweight pattern to a system.

There is an exception to the rule mentioned in Section 2.2. When the return type of around advice is `Object`, the advice is not rejected regardless of the return type of the join points. This exception allows advice to be applied to join points of different types but breaks type safety. Our approach allows advice to be applied into join points of different types while preserving type safety.

7. Conclusion and Future Work

This paper proposed a per-type instantiation mechanism for generic aspects. This mechanism allows the concrete generic aspects by creating an instance of a generic aspect for each type. When a piece of advice is executed, the mechanism (creates if necessary and) selects an instance according to the types of the join points.

We are currently finalising the language design and gathering more use cases to evaluate possible design choices.

We then implement our mechanism by extending ajc [6]. The only difference between the syntax of our approach and that of AspectJ is that concrete generic aspects and static advice are allowed, and the only difference between our weaving strategy and AspectJ's weaving strategy is that mentioned in Section 4. Therefore, we presume that the extension is straightforward.

Formal discussion on type safety is also future work. We plan to do this by extending Featherweight Java [7].

References

[1] The AspectJ 5 Development Kit Developer's Notebook. `http://www.eclipse.org/aspectj/doc/released/adk15notebook/`, 2005.

[2] T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *Proceedings of the 6th International Conference on Aspect-oriented Software Sevelopment (AOSD'07)*, pages 161–172, 2007.

[3] B. D. Fraine, M. Südholt, and V. 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, 2008.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[5] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 161–173, 2002.

[6] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 26–35, 2004.

[7] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 132–146, 1999.

[8] R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63 (3):267–296, 2006.

[9] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic Advice: On the combination of AOP with generative programming in AspectC++. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE2004)*, pages 55–74, 2004.

[10] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 214–240, 2005.

[11] H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. In *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'03)*, pages 297–306, 2003.

[12] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Design and implementation of an aspect instantiation mechanism. *Transactions on Aspect-Oriented Software Development*, 3880:259–292, 2006.