Is Join Point a Point?

—a pointcut and advice mechanism for making aspects more reusable—

Yusuke Endoh[†] Hidehiko Masuhara[‡] Akinori Yonezawa[†]

[†]Department of Computer Science, University of Tokyo

{mame,yonezawa}@yl.is.s.u-tokyo.ac.jp

[‡]Graduate School of Arts and Sciences, University of Tokyo

masuhara@acm.org

Abstract. We propose a new pointcut and advice mechanism for aspect-oriented languages based on the *point-in-time join points*. Unlike existing pointcut and advice mechanisms, the proposed one regards ends of actions as independent join points from the beginnings of actions, which contributes to more reusable and more robust aspects. This paper gives an overview of an AOP language based on the mechanism, and discusses its formalization and extension.

1 Introduction

Aspect-oriented programming (AOP) is a programming paradigm for modularizing crosscutting concerns, such as logging and security, whose implementations tend to involve with many modules in traditional module mechanisms[7].

One of the key mechanisms in common AOP languages[1, 2, 5, 6, 8–10] are the *pointcut and advice* mechanisms, which are to modularize crosscutting concerns in terms of program behavior. For example, a logging concern is to record all return values from `readLine` method in `Console` class during execution of an application program. The pointcut and advice mechanisms can elegantly implement the concern in a module called an aspect.

A pointcut and advice mechanism is explained by the following three elements:

- *join points* that represent certain kinds of actions during program execution (e.g., a call to `readLine` method, a call to `main` method, and an assignment to `counter` field),
- *pointcuts* that declaratively specify interested join points (e.g., any join point that represents a call to a method whose name begins with `read`), and
- *advice declarations* that add or substitute behavior to the specified join points (e.g., to record the parameter by calling `Log.add` method).

We emphasize that the pointcuts play key role in making aspects more reusable and robust against changes. When we apply an aspect to different base programs, we need to modify the aspect so that advice runs at appropriate points in execution. AOP languages make this task easier by separating pointcuts from advice declarations and by allowing redefining pointcuts later. For example, applying a logging concern to a different application requires to run the advice at different method calls.

The pointcut and advice mechanisms in existing AOP languages, however, have problems in reusability. Our observation is that the problem originates from the design of join points, namely a join point represents a *region-in-time* despite its name. This paper therefore propose an alternative mechanism in which a join point represents a *point-in-time*.

The following sections are organized as follows. The next section presents the problem in reusability. Section 3 overviews our alternative pointcut and advice mechanism. Sections 4 and 5 discuss formal semantics and advanced features of the mechanism. After discussing related work in Section 6, the paper concludes in Section 7.

2 Problems of Pointcut and Advice Mechanism Based on Region-in-Time Join Points

Although the pointcut and advice mechanisms in existing AOP languages are designed to be reusable, there are situations where aspects are less reusable. We argue that this is common to the mechanisms that regard join points as *region-in-time*.

In order to clarify the problem, this section uses a crosscutting concern that is to log user's input received by two base programs. The first base program, called *the console version*, receives input from the console. The second one, called *the hybrid version*, receives input either from the console or GUI components. As for the logging concern, we first implement an aspect for the CUI version.

In order to cope with changes from the console version to the hybrid version, we modify the aspect accordingly. Contrary to our expectation, the changes in the aspect are not trivial; we had to duplicate almost entire declarations in the aspect. At the end of the section, we discuss the problem originates in the design of the join points in the pointcut and advice mechanism.

2.1 Logging Aspect for the Console Version

Figure 1 shows the logging aspect for the console version in AspectJ[6]. We assume that the base program receives user input as return values of `readLine` method in any classes.

Lines 2 and 3 declare a pointcut `userInput` that matches any join point that calls `readLine` method. Lines 4–7 declare advice to log the input. The `after` modifier of the declaration specifies to run

```
1 aspect UserInputLogging {
2   pointcut userInput():
3     call(String *.readLine());
4   after() returning(String str)
5     : userInput() {
6     Log.add(str);
7   }
8 }
```

Figure 1: A logging aspect for the console version

the advice body *after* the matched join points. The `returning(String str)` modifier binds the return value from the join point to `str`. The body of the advice, which is at line 6, records the value.

The aspect is robust against some minor changes in the base program thanks to separation of pointcut and advice declarations. When we modify the base program to take user input from environment variable—by calling `System.getenv` method—as well, we only need to extend the pointcut declaration as follows:

```
2   pointcut userInput():
3     call(String *.readLine()) ||
4     call(String System.getenv(String));
```

It is even possible to linguistically declare an aspect that can subsume changes in pointcut with the aid of the mechanisms like abstract aspects, abstract pointcuts and aspect inheritance.

2.2 Modifying the Aspect to the Hybrid Version

The aspect is not reusable when the base program changes its programming style. In other words, pointcuts no longer can subsume changes in certain kinds of programming style.

Consider the hybrid version of the base program that receives user input from GUI components as well as from the console. In the version, when a user inputs a string, the GUI framework used here calls `onSubmit(String)` on a listener object in the base program with the string as an argument.

Figure 2 shows the logging aspect for the hybrid version. Note that the pointcuts can not subsume the changes from the console version; i.e., we have to define an additional advice declaration with an additional pointcut. This is because the hybrid version receives user input from GUI components as *parameters to some methods* in addition to the input from the console as *return values from some methods*.

```
 1 aspect UserInputLogging {
 2   pointcut userInput():
 3     call(String *.readLine());
 4   pointcut userInput2(String str):
 5     call(String *.onSubmit(String))
 6     && args(str);
 7   after() returning(String str)
 8     : userInput() {
 9     Log.add(str);
10   }
11   before(String str)
12     : userInput2(str) {
13     Log.add(str);
14   }
15 }
```

Figure 2: A logging aspect for the hybrid version

Figure 2 has an additional pointcut declaration at lines 4–6 and an additional advice declaration at lines 11–14. The pointcut specifies the join points that calls to method `onSubmit`, and binds actual parameters to the method to formal parameter `str`. The advice logs the user input just before method calls to `onSubmit` method.

2.3 Analysis of the Problem

By generalizing the problem observed above, we argue that pointcuts in existing AOP languages can not subsume differences between the beginnings of actions and the ends of actions.

The difference between the console version and the hybrid version is the timing to receive user input. The console version receives at the ends of actions (i.e., upon returning from `readLine`), while the GUI one receives at the beginnings of actions (i.e., upon a call to `onSubmit` from the framework).

Such a difference is not unique to the logging concern, but can also be seen in changes of minor programming styles. For example, differences between event-driven and polling (e.g., non-blocking I/O and blocking I/O), between returning an error value and throwing an exception to represent a failure and between direct style and continuation-passing style in functional programming.

Our claim is that the inability of pointcuts to subsume such differences roots from the design of join points in which a join point represents a *region-in-time* during program execution. For example, in AspectJ, a method call join point represents a region-in-time from the beginning of the method call until the end of the call. This design in turn requires advice declarations to select either the beginnings or the ends of the join points that are selected by pointcut.

3 Point-in-Time Join Points

3.1 Overview

We propose a new pointcut and advice mechanism based on the *point-in-time* join points. The mechanism differs the one based on the *region-in-time* join points in the following ways:

- A join point represents a point-in-time during execution, rather than a region-in-time. Consequently, there are no such notions like "beginning of a join point" or "end of a join point".
- There are new kinds of join points that represent terminations of actions. For example, returns from methods are independent join points of call join points.
- There are new pointcut constructs that match those new kinds of join points. For example, `return(f)` is a pointcut that selects a return from method `f`.
- Advice declarations no longer take modifiers like `before` to specify timing of execution.

Figures 3 and 4 illustrate the difference between the region-in-time join points and the point-in-time

ones. As in Figure 3, a call join point in existing AOP languages represents a whole life time of a method call. Our proposed mechanism, a method call is represented by two independent join points; one is a call join point that represents an action that begins a method call and the other is a return join point that represents an action that returns from a method call.

The next two advice declarations based on the region-in-time join points:

```
before(): call(* *.onSubmit(String))
after() returning: call(String *.readLine())
```

become the following ones based on the point-in-time join points:

```
advice(): call(* *.onSubmit(String))
advice(): return(String *.readLine())
```

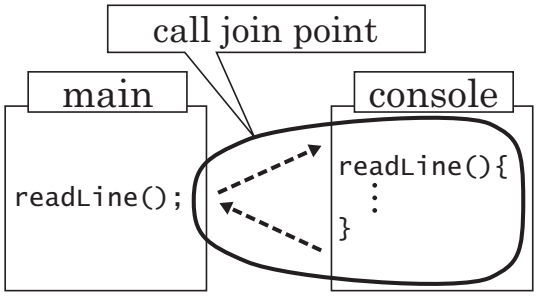

Figure 3: A call join point in existing mechanisms

We also extend the meaning of `args(x)` to bind return values to `x` upon return join points. Consequently, the next two advice declarations based on the region-in-time join points:

```
before(String str):
  call(* *.onSubmit(String)) && args(str)
after() returning(String str):
  call(String *.readLine())
```

become the following ones based on the point-in-time join points:

```
advice(String str):
  call(* *.onSubmit(String)) && args(str)
advice(String str):
  return(String *.readLine()) && args(str)
```

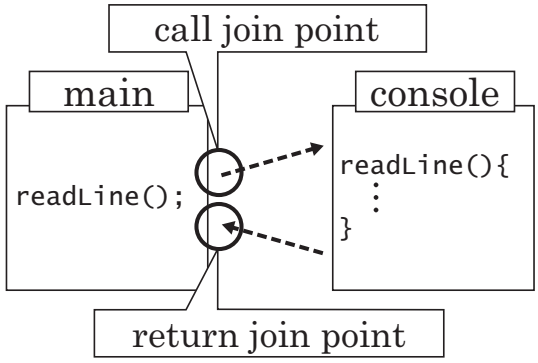

Figure 4: call and return join points in our proposed mechanism

3.2 Logging Aspect with Point-in-Time Join Points

With the point-in-time join points, we can define the logging aspects for the two versions of the base program as in Figures 5 and 6.

Figure 5 is not different from Figure 1 except in the pointcut that specifies returns from `readLine`, and binds return values to a variable. The advice declaration at lines 5–7 no longer has the `after` modifier because the pointcut specifies the timing.

In order to cope with the changes in the base program, we merely need to change the pointcut declaration as in Figure 6. The modified pointcut `userInput` matches returns from method `readLine` as well as calls to method `onSubmit`. Note that the advice declaration is not changed at all.

As we see, differences in the timing of advice execution as well as the way of passing parameters can be subsumed by pointcuts with the point-in-time based join points. This ability would foster to define more reusable aspect libraries by using abstract pointcuts because the library user can fully control the join points to apply aspect.

4 Formal Semantics

We give a formal semantics to a pointcut and advice mechanism based on the point-in-time join points. The details of the semantics are presented in the other

```
1 aspect UserInputLogging {
2   pointcut userInput(String str):
3     return(String *.readLine()) &&
4     args(str);
5   advice(String str): userInput(str) {
6     Log.add(str);
7   }
8 }
```

Figure 5: Logging aspect for the console version with the point-in-time join points

```
2   pointcut userInput(String str):
3     (return(String *.readLine()) ||
4      call(* *.onSubmit(String))) &&
5     args(str);
```

Figure 6: Modification to the pointcut for the hybrid version with the point-in-time join points

literature[4]. The semantics contributes to clarify the detailed behavior of the mechanism especially when integrated with other features such as excepting handling and first class continuations. It also helps to compare expressiveness of the mechanism against existing ones.

A denotational semantics is given in a continuation passing style (CPS). Since a return from a function is denoted as application of a value to a continuation in CPS, we can symmetrically model call and return join points.

5 Advanced Features

With the aid of the clarified semantics, we further investigated integration of advanced language features with the point-in-time join points. Thus far, the following two features are integrated into the mechanism:

Exception handling: Exception handling is also a good candidate of join points. In AspectJ, advice declarations have to distinguish exceptions by adding modifiers like `after throwing`. With the point-in-time join points, 'throwing an exception' can be simply regarded as an independent join point.

Around advice and proceed: One of the most notable drawbacks in the point-in-time join points is that it can not fully support all the features of `around` advice with the region-in-time join points. In AspectJ, an `around` advice has abilities to:

1. continue execution without running the join point,
2. proceed to the join point with different parameters,
3. continue execution with a different return value, and
4. run the join point more than once.

We observe that those abilities can also be simulated by the advice with point-in-time join points by adding a construct to modify the parameter to the join point (for the abilities 2 and 3), and by adding a construct to continue execution without running the join point (for the ability 1). For the last ability, we are investigating a mechanism similar to *partial continuation*[3].

6 Related Work

There are different ways to formalize the pointcut and advice mechanism. Wand et al. gave a denotational semantics to an AOP language that is a subset of AspectJ[12]. Since the language is based on the region-in-time join points, the formalization have to give different semantic equations to different kinds of advice declarations.

Walker et al. gave a semantics of an AOP language as a translation to a language of labeled terms and advice[11]. Although the surface AOP language is based on the region-in-time join points, the translated language is based on the point-in-time join points. A function call and a return from a function are uniformly treated as jumps to labels.

7 Conclusion

We proposed a new pointcut and advice mechanism based on the point-in-time join point. The mechanism treats ends of actions, such as returns from methods, as independent join points, which in turn gives more power to pointcuts. As a consequence, the mechanism contributes to define aspects more reusable and robust against changes.

We gave the semantics of the pointcut and advice mechanism based on the point-in-time join points in a continuation passing style. It uniformly represents beginnings and ends of actions as join points. We also investigated integration of advanced features with the proposed join point model.

Our future work includes the following topics. We will integrate other advanced features, such as `cflow` pointcut and tail-call elimination. We will also design and implement compilers for AOP languages based on the point-in-time join points.

Acknowledgments We would like to thank Kenichi Asai, the members of the Principles of Programming Languages Group at University of Tokyo, and the members of the Kumiki Project for their valuable comments.

References

[1] A. Bryant, and R. Feldt.
AspectR, http://aspectr.sourceforge.net/

[2] B. Burke, A. Chau, M. Fleury, A. Brock, A. Godwin, and H. Gliebe, JBoss Aspect Oriented Programming, The JBoss Group, http://www.jboss.org/developers/projects/jboss/aop, 2003

[3] O. Danvy and A. Filinski. Abstracting control. In Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, pages 151–160, Nice, France, June 1990.

[4] Y. Endoh, H. Masuhara, and A. Yonezawa, A Model for Aspect-Oriented Programming that Regards Applications to Continuations as Join Points. in Proceedings of Annual Conference of Japan Society for Software Science and Technology, September 2005. to appear.

[5] R. Hirschfeld, AspectS – Aspect-Oriented Programming with Squeak. in Proceedings of NODe 2002, LNCS 2591, pages 216-232, Springer-Verlag, 2003

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18–22 June 2001.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In ECOOP'97—Object-Oriented Programming, 11th European Conference, LNCS 1241, pages 220–242, 1997.

[8] H. Kim. AspectC#: An AOSD implementation for C#. MSc. Thesis, Comp.Sci, Trinity College, Dublin, Dublin, 2002.

[9] O. Spinczyk, D. Lohmann, and M. Urban, AspectC++: an AOP Extension for C++. in Software Developer's Journal, pages 68-76, 05/2005.

[10] A. Vasseur. Dynamic AOP and Runtime Weaving for Java – How does AspectWerkz Address it? In AOSD 2004, Dynamic AOP WorkShop, March 2004.

[11] D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. In International Conference on Functional Programming, 2003.

[12] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. ACM TOPLAS. 2003.