A Fine-Grained Join Point Model for More Reusable Aspects

Hidehiko Masuhara[1], Yusuke Endoh[2,⋆], and Akinori Yonezawa[2]

[1] Graduate School of Arts and Sciences, University of Tokyo
masuhara@acm.org
[2] Department of Computer Science, University of Tokyo
{mame, yonezawa}@yl.is.s.u-tokyo.ac.jp

Abstract. We propose a new join point model for aspect-oriented programming (AOP) languages. In most AOP languages including AspectJ, a join point is a time interval of an action in execution. While those languages are widely accepted, they have problems in aspects reusability, and awkwardness when designing advanced features such as tracematches. Our proposed join point model, namely the point-in-time join point model redefines join points as the moments both at the beginning and end of actions. Those finer-grained join points enable us to design AOP languages with better reusability and flexibility of aspects. In this paper, we designed an AspectJ-like language based on the point-in-time model. We also give a denotational semantics of a simplified language in a continuation passing style, and demonstrate that we can straightforwardly model advanced language features such as exception handling and `cflow` pointcuts.

1 Introduction

Aspect-oriented programming (AOP) is a programming paradigm that addresses problems of crosscutting concerns[11, 15], such as exception handling, security mechanisms and coordinations among modules. Since implementations of crosscutting concerns without AOP have to involve with many modules, AOP improves maintainability of programs by making those concerns into separate modules.

One of the fundamental language mechanisms in AOP is the *pointcut and advice* mechanism, which can be found in many AOP languages including AspectJ[15]. As previous studies have shown, design of pointcut language and selection of join points are key design factors of the pointcut and advice mechanisms in terms of expressiveness, reusability and robustness of advice declarations [4,14, 16–18, 21].

A pointcut serves as an abstraction of join points in the following senses:

– It can give a name to a set of join points (e.g., by means of *named pointcuts* in AspectJ).

⋆ Currently with Toshiba Corp.

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 131–147, 2006.

– Differences among join points, such as join point kinds and parameter positions, can be subsumed. For example, when we define a logging aspect that records the first argument to `runCommand` method and the second argument to `debug`, different parameter positions are subsumed by the next pointcut:

```
pointcut userInput(String s):
    (call(* Toplevel.runCommand(String)) && args(s))
 || (call(* Debugger.debug(int,String)) && args(*,s));
```

– It can separate concrete specifications of interested join points from advice declarations (e.g., by means of *abstract pointcuts* and *aspect inheritance* in AspectJ). In other words, we can parameterize interested join points in an advice declaration.

There have been several studies on advanced pointcut primitives for accurately and concisely abstracting join points[4, 16, 17, 21].

In order to allow pointcuts to accurately abstract join points, the pointcut and advice mechanisms should also have a rich set of join points. If an interested event is not a join point, there is not way to advise it at all. Several studies have investigated to introduce new kinds of join points, such as loops[14], conditional branches[18], and local variable accesses[19] into AspectJ-like languages. In other words, the more kinds of join points the pointcut and advice mechanism has, the more opportunities advice declarations can be applied to.

This paper focuses on a language with finer grained join points for improving reusability of advice declarations. The join point model can be compared with traditional join point model in AspectJ-like languages as follows:

– In the join point model in AspectJ-like languages, a join point represents duration of an event, such as a call to a method until its termination. We call this model the *region-in-time* model because a join point corresponds to a region on a time line.
– In our proposing join point model, a join point represents an instant of an event, such as the beginning of a method call and the termination of a method call. We call this model the *point-in-time* model because a join point corresponds to a point on a time line.

The contributions of the paper are:

– We demonstrate that the point-in-time join point model can improve reusability of advice.
– We present an experimental AOP language called PitJ based on the point-in-time model. PitJ's advice is as expressive as AspectJ's in most typical use cases even though the advice mechanism in PitJ is simpler than the one in AspectJ-like languages.
– We give a formal semantics of the point-in-time model by using a small functional AOP language called Pitλ. Thanks to affinity with continuation passing style, the semantics gives a concise model with advanced features such as exception handling.