to appear in Proc. of The Fourth ASIAN Symposium on Programming Languages and Systems (APLAS 2006), Nov. 2006

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.

– 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.

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

Fig. 1. Logging aspect for the console version

2 Reusability Problem of Region-in-Time Join Point Model

Although languages that are based on the region-in-time join point model are designed to be reusable, there are situations where aspects are not as reusable as they seem to be. This section explains such situations, and argues that this is common problem to the region-in-time join point model.

In order to clarify the problem, this section uses a crosscutting concern that is to log user's input received by the following two versions of base program:

a console version that receives user input from the console.
a hybrid version, evolved from the console version, that receives user input from both the console and GUI components.

2.1 Logging Aspect for the Console Version

Figure 1 shows a logging aspect for the console version in AspectJ[15]. We assume that the base program receives user input as return values of `readLine` method in several classes.

Line 2 declares a pointcut `userInput` that matches any join point that represents a call to `readLine` method. Lines 3–5 declare advice to log the input. `after() returning(String s)` is an advice modifier of the advice declaration that specifies to run the advice body *after* the action of the matched join points with binding the return value from the join point to variable `s`. The body of the advice, which is at line 4, records the value.

It is possible to declare a generic aspect in order to subsume changes of join points to be logged in different versions. For example, Figure 2 shows a generic logging aspect that uses abstract pointcut `userInput` in an advice declaration, and a concrete logging aspect for the console version that concretizes `userInput` into `call(String *.readLine())`.

The generic logging aspect is reusable to log user's input from environment variables by changing `userInput()` pointcut in `ConsoleLogging` in Figure 2 to `call(String *.readLine()) || call(String System.getenv(String))`. Note that we do not need to modify the generic logging aspect.

```
1 abstract aspect UserInputLogging {
2   abstract pointcut userInput();
3   after() returning(String s): userInput() {
4     Log.add(s);
5   }
6 }

7 aspect ConsoleLogging extends UserInputLogging {
8   pointcut userInput(): call(String *.readLine());
9 }
```

Fig. 2. Generic logging aspect and its application to the console version

```
1 aspect HybridLogging extends UserInputLogging {
2   pointcut userInput(): call(String *.readLine());
3   pointcut userInput2(String s):
4     call(String *.onSubmit(String)) && args(s);
5   before(String s): userInput2(s) {
6     Log.add(s);
7   }
8 }
```

Fig. 3. Logging aspect for the hybrid version

2.2 Modifying the Aspect to the Hybrid Version

The generic logging 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 a hybrid version of the base program that receives user input from GUI components as well as from the console. The version uses the GUI framework which calls `onSubmit (String)` method on a listener object in the base program with the string *as an argument* when a user inputs a string via GUI interface.

Since `UserInputLogging` in Figure 2 can only log return values, we have to define a different pointcut and advice declaration as shown in Figure 3.

Making the logging aspect for hybrid version reusable is tricky and awkward. Since single pointcut and advice can not subsume differences between return values and arguments, we have to define a pair of pointcuts and advice declarations. In order to avoid duplication in advice bodies, we need to define an auxiliary method and let advice bodies call the method. The resulted aspect is shown in Figure 4.

Some might argue that it is possible to reuse `UserInputLogging` aspect in Figure 4 by finding join points that always run before calls to `onSubmit`. However, such join points can not always be found, especially when advice declarations take parameters from join points. Moreover, such a compromise usually

```
 1 abstract aspect UserInputLogging2 {
 2   abstract pointcut userInputAsReturnValue();
 3   abstract pointcut userInputAsArgument(String s);
 4   after() returning(String s): userInputAsReturnValue() {
 5     log(s);
 6   }
 7   before(String s): userInputAsArgument(s) {
 8     log(s);
 9   }
10   void log(String s) {
11     Log.add(s);
12   }
13 }
```

Fig. 4. Generic logging aspect that can log for both return values and arguments

makes aspects fragile because the pointcuts *indirectly* specify join points that the aspects are actually interested in.

2.3 Awkwardness in Advanced Pointcuts

Some advanced pointcuts require to distinguish beginnings and ends of actions as different events. However, since region-in-time model does not distinguish them as different join points, the resulted languages have to introduce mechanisms to not only identifying join points but also mechanisms to specify their beginnings and ends.

For example, the *trace maching* mechanism is one of the useful extensions to AOP languages that enables advice run based on the history of events[1]. The code below shows an example of a tracematch that logs `query` calls performed only after completion of a `login` call.

```
1 tracematch() {
2   sym login after returning: call(* login(User,..));
3   sym query before: call(* query(Query));
4   login query+        // any query after login
5   { Log.add(...); }   // shall be logged
6 }
```

The description of the tracematch consists of two parts, namely declarations of the symbols and a piece of code with a trace pattern. Line 2 and 3 declare symbols `login` and `query` as the end of a `login` call and the beginning of a `query` call, respectively. Then line 4 specify the trace pattern of those events in a regular expression of declared symbols.

One might first think that using named pointcuts instead of symbols could simplify the language without losing expressiveness. However, it is not possible as the named pointcuts can merely specify the join points and lack the information

whether the programmer is interested in either the beginnings or the ends of the join points.

2.4 Analysis of the Problem

By generalizing the above problem, we argue that pointcuts in the region-in-time join point model can not subsume differences between the beginnings of actions and the ends of actions.

Such a difference is not unique to the logging concern, but can also be seen in many cases. For example, following differences can not be subsumed by pointcuts in the region-in-time join point model:

- a polling style program that waits for events by calling a method and an event driven style program that receives events by being called by a system,
- a method that reports an error by returning a special value and a method that does by an exception, and
- a direct style program in which caller performs rest of the computation and continuation-passing style in which the rest of computation is specified by function parameters.

Our claim is that the problem roots from the design of join point model in which a join point represents a region-in-time, or a time interval during program execution. For example, in AspectJ, a call join point represents a region-in-time while invoking the method, executing the body of the method and returning from the method. This design in turn requires advice modifiers which indicate either the beginnings or the ends of the join points that are selected by pointcut.

3 Point-in-Time Join Point Model

3.1 Overview

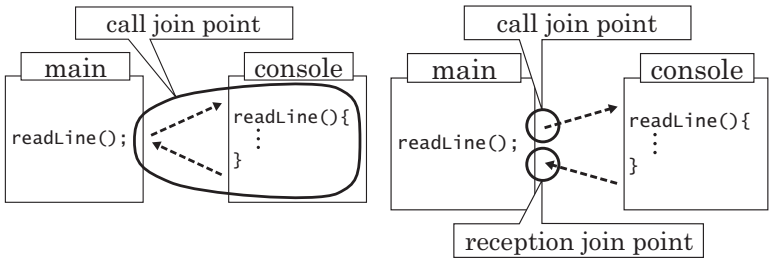
We propose a new join point model, called *point-in-time join point model*, and design an experimental AOP language, called *PitJ*. PitJ differs from AspectJ-like languages in the following ways:

- A join point represents a point-in-time (or an instant of program execution) rather than a region-in-time (or an interval). 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, a return from methods is an independent join point, which we call *a reception*[3] *join point*, from a call join point. Similarly, an exceptional return is *a failure join point*. Table 1 lists the join points in PitJ along with respective ones in AspectJ.

[3] Older versions of AspectJ[15] have reception join points for representing different actions.

PitJ	AspectJ
call / reception / failure	method call
execution / return / throw	method execution
get / success_get / failure_get	field reference
set / success_set / failure_set	field assignment

Table 1. Join points in PitJ and AspectJ



in AspectJ-like languages in PitJ

Fig. 5. Join points in languages based on region-in-time and point-in-time models.

- There are new pointcut constructs that match those new kinds of join points. For example, `reception(m)` is a pointcut that selects any reception join point that returns from the method `m`.
- Advice declarations no longer take modifiers like `before` and `after` to specify timing of execution.

Figure 5 illustrate the difference between the point-in-time join point model and region-in-time one.

Figure 6 shows example aspect definitions in PitJ. The generic aspect (lines 1–6) is not different from the one in AspectJ expect that the advice does not take a modifier (line 3). `HybridLogging` aspect concretizes the pointcut by using reception and call pointcut primitives (lines 9–10). When `readLine` returns to the base program, a reception join point is created and matches the `userInput`. The return value is bound to `s` by `args` pointcut. When `onSubmit` method is called, a call join point matches the pointcut with binding the argument to `s`.

As we see in Figure 6, 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 join point model. This ability allows us to define more reusable aspect libraries by using abstract pointcuts because users of the library can fully control the join points to apply aspect.

We verified the reusability problem which is effectively solved by the point-in-time join point model by case study with some realistic applications, aTrack[2] and AJHotDraw[20]. The details of the case study are presented in the other literature[13].

```
 1 abstract aspect UserInputLogging {
 2   abstract pointcut userInput(String s);
 3   advice(String s) : userInput(s) {
 4     Log.add(s);
 5   }
 6 }

 7 aspect HybridLogging extends UserInputLogging {
 8   pointcut userInput(String s): args(s) &&
 9     (reception(String *.readLine()) || call(* *.onSubmit(String)));
10 }
```

Fig. 6. A logging abstract aspect and its application to the hybrid vertion in PitJ

```
1 aspect ErrorReporting {
2   after() throwing: call(* *.readLine()) {
3     System.out.println("exception");
4   }
5 }
```

Fig. 7. An aspect to capture exceptions in AspectJ

3.2 Exception Handling

In AspectJ, advice declarations have to distinguish exceptions by using a special advice modifier `after() throwing`. It specifies to run the advice body when interested join points terminate by throwing exception. For example, a sample aspect in Figure 7 prints a message when an uncaught exception is thrown from `readLine`. Similar to the discussion on the `before` and `after` advice, termination by throwing an exception and normal termination can not be captured by single advice declartion[4].

In PitJ, 'termination by throwing an exception' is regarded as an independent failure join point. Figure 8 is an equivalent to the one in Figure 7. A pointcut `failure` matches a failure join point which represents a point-in-time at the termination of a specified method by throwing an exception.

3.3 Around-like Advice

One of the fundamental questions to PitJ is, by simplifying advice modifiers, whether it is expressive enough to implement around advice in AspectJ, which has powerful mechanisms. We analyzed that around advice in AspectJ has four abilities:

1. replace the parameters to the join point with new ones,

[4] It is possible to capture them by using `after` advice, which however can not access to return values or exception objects.

```
1 aspect ErrorReporting {
2   advice(): failure(* *.readLine()) {
3     System.out.println("exception");
4   }
5 }
```

Fig. 8. An aspect to capture exceptions in PitJ

2. replace the return values to the caller of the join point,
3. go back to the caller without executing the join point, and
4. execute the join point more then once.

In PitJ, the abilities 1 and 2 can be simulated by treating a return value of an advice body as a new value. For example, consider an advice declaration:

```
advice(String s): args(s) && (reception(* *.readLine())
                              || call(* *.onSubmit(String)) {
  return s.replaceAll("<", "&lt;").replaceAll(">", "&gt;");
}
```

This advice sanitizes user input by replacing unsafe characters with escape sequences. When an advice body ends without `return`, the value in the join points remains unchanged.

For the ability 3, we introduce a new construct `skip`. When it is evaluated in a call join point, jump occurs to the subsequent reception join point with no execution between the two join points. Nothing happens when evaluated in a reception and failure join points. For example, consider an advice declaration:

```
advice(): call(* *.readLine()) { skip "dummy"; }
```

With the advice, even if `readLine()` is evaluated, it immediately returns `"dummy"` without reading any string from a console.

For the ability 4, a special function `proceed` is added. It executes the action until the subsequent reception one, and then returns the result. For example, consider an advice declaration:

```
advice(): call(* *.readLine()) {
  skip(proceed() + proceed());
}
```

With this advice, the method `readLine` receives two lines at once, concatenates them, and returns it.

We introduced the construct `skip` so that advice declarations can dynamically control how to `proceed`. An alternative design would be to introduce a different kind of advice that does not proceed to original join points even if it does not evaluate `skip`. We need further programming experience to compare those alternatives in terms of program readability.

3.4 More Advanced Features

Some existing AOP systems including AspectJ provides some context sensitive pointcuts. They don't always match specific kinds of join points. Instead, they judge whether a join point is in a specific context. PitJ has `cflow` pointcut, which is a kind of context sensitive pointcuts. It identifies join points based on whether they occur in the dynamic context during a region-in-time between a specified call join point and the subsequent reception one. For example, `cflow(call(* *.onSubmit(String)))` specifies any join point that occurs between when a `onSubmit` method is called and when it returns.

In addition, we are considering the integration of trace sensitive aspects[9, 10, 21] which use execution trace, or history of occured join points, to judge whether to perform additional computation, We expect that our finer grained join points enhance its effectiveness and robustness.

3.5 Design Considerations of Pointcut Primitives

The design of the pointcuts in PitJ is chosen among several alternatives. In fact, we examined the following three designs, which have different advantages and disadvantages:

1. Provide a primitive for each kind of join point, similar to the pointcuts in AspectJ. While it makes each pointcut description simple, it requires many pointcut primitives. This is our current design.
2. Provide a set of primitives that discriminates kinds of events (e.g., call and execution) and a set of primitives taht discriminates timing relative to an event (e.g., entry and exit). For example, `call(* *.readLine())` matches both beginnings and ends of `readLine` calls, and `call(* *.readLine()) && exit()` matches only ends of `readLine` calls. It requires a smaller set of pointcut primitives, but often makes each pointcut description longer.
3. Provide a set of primitive that identifies join points that represent beginnings of events, in addition to `cflow`-like pointcuts that create pointcuts that identify ends and failures of events from a given pointcut. For example, `call(* *.readLine())` matches begginings of `readLine` calls and `cont(call(* *.readLine()))` matches ends of `readLine` calls. Though this design might be more powerful than the above two designs, it is not certain whether we can define a clear semantics.

We chose the first design because its simplicity and affinity with AspectJ. No design is, however, clearly better than others. More programming experiences will give us better insight to discuss about the right design.

4 Formal Semantics

We present a formal semantics of Pitλ, which is a simplified version of PitJ. Pitλ simplifies PitJ by using a lambda-calculus as a base language, and by supporting

Syntax:

$$
\begin{array}{lrll}
\text{(Expression)} & e ::= & x & \text{(Identifier)} \\
& \mid & \texttt{fun}\ x \to e & \text{(Function)} \\
& \mid & e\ e & \text{(Application)}
\end{array}
$$

Semantic algebras:
numbers Int, booleans $Bool$, identifiers Ide

$$
\begin{array}{lll}
v \in Val = Int + Bool + Fun & & \text{(Values)} \\
\rho \in Env = Ide \to Val & & \text{(Environments)} \\
\kappa \in Ctn = Val \to Ans & & \text{(Continuations)} \\
f \in Fun = Ctn \to Ctn & & \text{(Functions)} \\
Ans = Val_\perp & & \text{(Answers)}
\end{array}
$$

Valuation function for the expressions:

$$
\begin{array}{l}
\mathcal{E} : \text{Expression} \to Env \to Ctn \to Ans \\
\quad \mathcal{E}[\![x]\!]\ \rho\ \kappa = \kappa\ (\rho\ x) \\
\mathcal{E}[\![\texttt{fun}\ x \to e]\!]\ \rho\ \kappa = \kappa\ (inFun(\lambda\kappa' v.\ \mathcal{E}[\![e]\!]\ ([v\ /\ x]\rho)\ \kappa')) \\
\quad \mathcal{E}[\![e_0\ e_1]\!]\ \rho\ \kappa = \mathcal{E}[\![e_0]\!]\ \rho\ (\lambda Fun(f).\ \mathcal{E}[\![e_1]\!]\ \rho\ (\lambda v.f\ \kappa\ v))
\end{array}
$$

Fig. 9. Syntax and semantics of the base language

only call, reception and failure join points. The semantics contributes to clarify the detailed behavior of the program especially when integrated with other advanced features such as exception handling and context sensitive pointcuts. It also helps to compare expressiveness of the point-in-time join point model against the region-in-time one.

4.1 Base Language

Figure 9 shows the syntax of the base language and its denotational semantics in a continuation passing style (CPS). We use untyped lambda-calculus as the base language. The semantics follows the style of Danvy and Filinski[8].

4.2 Syntax and semantics of Pitλ_0

We begin with Pitλ_0, which is a core part of Pitλ that has only call and reception join points. Syntactically, it uses the same expressions to the base language, and has pointcuts and a list of advice as shown in Figure 10.

We give a semantics of Pitλ_0 by modifying the semantics of the base language in Section 4.1.

First, we define additional semantic algebras. An event ϵ is either call or reception with a function name and a join point θ is a pair of an event and an argument:

$$
\begin{array}{lll}
\epsilon ::= \mathsf{call}(x) \mid \mathsf{reception}(x) & & (Evt) \\
\theta ::= (\epsilon, v) & & (Jp)
\end{array}
$$

$$
\begin{array}{llll}
\text{(Expression)} & e ::= & x & (\textsc{Identifier}) \\
 & \mid & \texttt{fun}\ x \to e & (\textsc{Function}) \\
 & \mid & e\ e & (\textsc{Application}) \\
\text{(Pointcut)} & p ::= & \texttt{call}(x) \mid \texttt{reception}(x) \mid \texttt{args}(x) \mid p\ \texttt{\&\&}\ p \mid p\ \texttt{||}\ p & \\
\text{(Advice)} & a ::= & \cdot \mid \texttt{advice} : p \to e;\ a &
\end{array}
$$

Fig. 10. Pitλ_0 syntax

$$
\begin{array}{l}
\mathcal{P} : \text{Pointcut} \to \mathit{Env} \to \mathit{Jp} \to (\mathit{Env} \cup \{\mathit{False}\}) \\
\mathcal{P}[\![\texttt{call}(x)]\!]\ \rho\ (\mathsf{call}(x'), v) = \begin{cases} \rho & \mathit{if}\ x = x'\ \mathit{or}\ x = * \\ \mathit{False} & \mathit{otherwise} \end{cases} \\
\mathcal{P}[\![\texttt{reception}(x)]\!]\ \rho\ (\mathsf{reception}(x'), v) = \begin{cases} \rho & \mathit{if}\ x = x'\ \mathit{or}\ x = * \\ \mathit{False} & \mathit{otherwise} \end{cases} \\
\mathcal{P}[\![\texttt{args}(x)]\!]\ \rho\ (\epsilon, v) = [v\ /\ x]\rho \\
\mathcal{P}[\![p_0\ \texttt{\&\&}\ p_1]\!]\ \rho\ \theta\ = \begin{cases} \mathcal{P}[\![p_1]\!]\ \rho'\ \theta & \mathit{if}\ \mathcal{P}[\![p_0]\!]\ \rho\ \theta = \rho' \\ \mathit{False} & \mathit{otherwise} \end{cases} \\
\mathcal{P}[\![p_0\ \texttt{||}\ p_1]\!]\ \rho\ \theta\ = \begin{cases} \rho' & \mathit{if}\ \mathcal{P}[\![p_0]\!]\ \rho\ \theta = \rho' \\ \mathcal{P}[\![p_1]\!]\ \rho\ \theta & \mathit{otherwise} \end{cases}
\end{array}
$$

Fig. 11. Semantics of pointcuts

Additionally, we define an auxiliary function σ that extracts a signature (or a name) from an expression.

$$
\begin{array}{l}
\sigma : \text{Expression} \to \textsc{Identifier} \\
\sigma(e) = \begin{cases} e & \mathit{if}\ e\ \mathit{is}\ \textsc{Identifier} \\ \$ & \mathit{otherwise} \end{cases}
\end{array}
$$

If it receives an IDENTIFIER, the argument itself is returned. Otherwise, it returns the dummy signature \$. For example, $\sigma(x)$ is x, and $\sigma(\texttt{fun}\ x \to x)$ is \$.

The semantics of the pointcuts is a function $\mathcal{P}$ shown in Figure 11. $\mathcal{P}[\![p]\!]\ \rho_{empty}\ \theta$ tests whether the pointcut p and the current join point θ match. If they do, it returns an environment that binds a variable to a value by `args` pointcut. Otherwise, it returns *False*.

We then define the semantic function $\mathcal{A}$ for lists of advice declarations (Figure 12), which receives an advice list, an event and a continuation. When the pointcut of the first advice matches a join point, it returns a continuation that evaluates the advice body and then evaluates the rest of the advice list. Otherwise, it returns a continuation that evaluates the rest of the advice list. At the end of the list, it continues to the original computation.

We finally define the semantic function of the expression. In the section, the semantics of IDENTIFIER and FUNCTION remain unchanged. The semantics of APPLICATION in Pitλ_0 is defined by inserting application to $\mathcal{A}$ at appropriate

$$
\begin{array}{l}
\mathcal{A} : \text{Advices} \to \mathit{Evt} \to \mathit{Ctn} \to \mathit{Ctn} \\
\mathcal{A}[\![\texttt{advice} : p \to e; a']\!]\ \epsilon\ \kappa\ v = \begin{cases} \mathcal{E}[\![e]\!]\ \rho'\ (\mathcal{A}[\![a']\!]\ \epsilon\ \kappa) & \mathit{if}\ \mathcal{P}[\![p]\!]\ \rho_{\mathit{empty}}\ (\epsilon, v) = \rho' \\ \mathcal{A}[\![a']\!]\ \epsilon\ \kappa\ v & \mathit{otherwise} \end{cases} \\
\mathcal{A}[\![\ \cdot\ ]\!]\ \epsilon\ \kappa\ v = \kappa\ v
\end{array}
$$

Fig. 12. Semantics of advice

$$
\begin{array}{l}
\mathcal{E} : \text{Expression} \to \mathit{Env} \to \mathit{Ctn} \to \mathit{Ans} \\
\mathcal{E}[\![x]\!]\ \rho\ \kappa = \kappa\ (\rho\ x) \\
\mathcal{E}[\![\texttt{fun}\ x \to e]\!]\ \rho\ \kappa = \kappa\ (\mathit{inFun}(\lambda\kappa' v.\ \mathcal{E}[\![e]\!]\ ([v\ /\ x]\rho)\ \kappa')) \\
\mathcal{E}[\![e_0\ e_1]\!]\ \rho\ \kappa = \mathcal{E}[\![e_0]\!]\ \rho\ (\lambda \mathit{Fun}(f).\ \mathcal{E}[\![e_1]\!]\ \rho\ (\lambda v. \\
\qquad\qquad \mathcal{A}[\![a_0]\!]\ \mathsf{call}(\sigma(e_0))(f\ (\mathcal{A}[\![a_0]\!]\ \mathsf{reception}(\sigma(e_0))\ \kappa))\ v))
\end{array}
$$

Fig. 13. Semantics of expressions

positions. The original semantics of Application is as follows:

$$\mathcal{E}[\![e_0\ e_1]\!]\ \rho\ \kappa\ =\ \mathcal{E}[\![e_0]\!]\ \rho\ (\lambda \mathit{Fun}(f).\ \mathcal{E}[\![e_1]\!]\ \rho\ (\lambda v.\ \boxed{f\ \kappa}\ v))$$

The shadowed part $\boxed{f\ \kappa}$ is a continuation that executes the function body and passes the result to the subsequent continuation κ. The application to the continuation $\boxed{f\ \kappa}\ v$, therefore, corresponds to a call join point. By replacing the continuation with $\mathcal{A}[\![a]\!]\ \mathsf{call}(x)\ (f\ \kappa)$, we can run applicable advice at function calls:

$$\mathcal{E}[\![e_0\ e_1]\!]\ \rho\ \kappa\ =\ \mathcal{E}[\![e_0]\!]\ \rho\ (\lambda \mathit{Fun}(f).\ \mathcal{E}[\![e_1]\!]\ \rho\ (\lambda v.\mathcal{A}[\![a_0]\!]\ \mathsf{call}(\sigma(e_0))\ (f\ \kappa)\ v))$$

where a_0 is the globally defined list of all advice declarations.

Similarly a reception of a return value from a function application can be found by η-expanding[5] κ as follows:

$$\mathcal{E}[\![e_0\ e_1]\!]\ \rho\ \kappa = \mathcal{E}[\![e_0]\!]\ \rho\ (\lambda \mathit{Fun}(f).\mathcal{E}[\![e_1]\!]\ \rho\ (\lambda v.f\ (\lambda v'.\boxed{\kappa}\ v')\ v))$$

Therefore, advice application at reception join point can be achieved by replacing κ with $\mathcal{A}[\![a]\!]\ \mathsf{reception}(x)\ \kappa$.

Figure 13 shows the final semantics for the expression with call and reception join points. As we have seen, advice application is taken into the semantic function in a systematic way: given a continuation κ that represents a join point, substitute with $\mathcal{A}[\![a]\!]\ \epsilon\ \kappa$. In the next section, we will see advanced features can also be incorporated in the same ways.

[5] This η-expansion prevents *tail-call elimination*. It fits the facts that defining an advice whose pointcut specifies a reception join point makes tail-call elimination impossible.

(Expression)	$e ::= \ldots$	
	$\mid$ `try` e `with` $x \to e$	(Try)
	$\mid$ `raise` e	(Raise)
(Pointcut)	$p ::= \ldots \mid$ `failure(`x`)`	

Fig. 14. Additional constructs for exception handling

5 Advanced Features with Pitλ

With the aid of the clarified semantics, we are now able to discuss advanced language features with the point-in-time model. Thus far, we investigated several advanced features by defining an extended language called Pitλ_1. The investigated features include exception handling, context sensitive pointcuts (i.e., `cflow`-like pointcut) and around advice. Due to the space limitation, we only present the exception handling mechanism below. The other features are explained in the other literatures[12, 13].

5.1 Exception Handling

In AspectJ, advice declarations have to distinguish exceptions by using a special advice modifier (as described in Subsection 3.2). It not only complicates the problem in reusability, but also makes the semantics awkward. This is because we have to pay attention to all combinations of advice modifiers and pointcuts. In fact, some existing formalizations[22, 23] gave a slightly different semantic equation to each kind of advice declarations. Meanwhile, the point-in-time join point model has no advice modifiers, which makes the semantics simpler.

Figure 14 shows additional constructs for exception handling: Try and Raise as the expression, and `failure` as the pointcut. For the sake of simplicity, we don't introduce the special values which represent an exception; an arbitrary value can be raised. For example, `(fun x` $\to$ `raise x) 1` raises the value `1` as an exception. `try ((fun x` $\to$ `raise x) 1) + 2 with x` $\to$ `x+ 3` is evaluated normally to the value `4`. But, with `advice : failure(*) && args(x)` $\to$ `x * 2`, it is evaluated to the value `5`.

We first give a standard denotational semantics to these constructs. In preparation for it, we introduce a continuation which represents current exception handler to the semantics algebra *Fun* and the semantic functions $\mathcal{A}$ and $\mathcal{E}$:

$$f \in \mathit{Fun} = \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Ctn}$$

$$\mathcal{E} : \text{Expression} \to \mathit{Env} \to \mathit{Ctn} \to \mathit{Ctn} \to \mathit{Ans}$$

$$\mathcal{E}[\![x]\!]\,\rho\,\kappa_h\,\kappa = \kappa\,(\rho\,x)$$

$$\mathcal{E}[\![\texttt{fun}\ x \to e]\!]\,\rho\,\kappa_h\,\kappa = \kappa\,(\mathit{inFun}(\lambda \kappa_h'\,\kappa' v.\ \mathcal{E}[\![e]\!]\,([v\ /\ x]\rho)\,\kappa_h'\,\kappa'))$$

$$\mathcal{E}[\![e_0\ e_1]\!]\,\rho\,\kappa_h\,\kappa = \mathcal{E}[\![e_0]\!]\,\rho\,\kappa_h\,(\lambda \mathit{Fun}(f).\ \mathcal{E}[\![e_1]\!]\,\rho\,\kappa_h\,(\lambda v.\ \mathcal{A}[\![a]\!]\,\mathsf{call}(\sigma(e_0))\,\kappa_h\,(f\ \kappa_h\ (\mathcal{A}[\![a]\!]\,\mathsf{reception}(\sigma(e_0))\,\kappa_h\,\kappa))\,v))$$

(a) Pointcuts (`failure` only):

$$\mathcal{P}[\![\texttt{failure}(x)]\!]\ \rho\ (\mathsf{failure}(x'), v) = \begin{cases} \rho & \textit{if } x = x' \textit{ or } x = * \\ \textit{False} & \textit{otherwise} \end{cases}$$

(b) Advices:

$$\mathcal{A} : \text{Advices} \to \textit{Evt} \to \boxed{\textit{Ctn} \to \textit{Ctn}} \to \textit{Ctn} \to \textit{Ctn}$$

$$\mathcal{A}[\![\texttt{advice} : p \to e; a']\!]\ \epsilon\ \boxed{\kappa_h}\ \kappa\ v\ = \begin{cases} \mathcal{E}[\![e]\!]\ \rho'\ (\mathcal{A}[\![a']\!]\ \epsilon\ \boxed{\kappa_h}\ \kappa) & \textit{if } \mathcal{P}[\![p]\!]\ \rho_{\textit{empty}}\ (\epsilon, v) = \rho' \\ \mathcal{A}[\![a']\!]\ \epsilon\ \boxed{\kappa_h}\ \kappa\ v & \textit{otherwise} \end{cases}$$

$$\mathcal{A}[\![\ \cdot\ ]\!]\ \epsilon\ \boxed{\kappa_h}\ \kappa\ v = \kappa\ v$$

(c) Expressions (Application, Try and Raise only):

$$\begin{aligned} \mathcal{E}[\![e_0\ e_1]\!]\rho\kappa_h\kappa = \mathcal{E}[\![e_0]\!]\ \rho\ \kappa_h\ (\lambda \textit{Fun}(f).\ \mathcal{E}[\![e_1]\!]\ \rho\ \kappa_h\ (\lambda v.\ \\ \mathcal{A}[\![a]\!]\ \mathsf{call}(\sigma(e_0))\ \kappa_h \\ (f\ (\mathcal{A}[\![a]\!]\ \mathsf{failure}(\sigma(e_0))\ \kappa_h\ \kappa_h)(\mathcal{A}[\![a]\!]\ \mathsf{reception}(\sigma(e_0))\ \kappa_h\ \kappa))\ v)) \\ \mathcal{E}[\![\texttt{try}\ e_0\ \texttt{with}\ x \to e_1]\!]\ \rho\ \kappa_h\ \kappa = \mathcal{E}[\![e_0]\!]\ \rho\ (\lambda v.\ \mathcal{E}[\![e_1]\!]\ ([v\ /\ x]\rho)\ \kappa_h\ \kappa)\ \kappa \\ \mathcal{E}[\![\texttt{raise}\ e]\!]\ \rho\ \kappa_h\ \kappa = \mathcal{E}[\![e]\!]\ \rho\ \kappa_h\ \kappa_h \end{aligned}$$

Fig. 15. Semantics of Pitλ_1 with exception handling

The new definition of $\mathcal{A}$ is in Figure 15-(b). This modification, adding the shadowed parts, is mechanical since additional continuations are dealt with only by the additional constructs. After that, we can define a semantics of the Try and the Raise as Figure 15-(c).

Now, we define the semantics of a failure join point by modifying the original semantics. The failure is added to the events *Evt*:

$$\epsilon\ ::=\ \ldots \mid \mathsf{failure}(x)$$

and the semantics of the `failure` pointcuts is defined as Figure 15-(a).

Then, look the semantics of Application. From the first argument κ_h in $f\ \kappa_h\ \ldots$, show up the application form by η-expansion.

$$\begin{aligned} \mathcal{E}[\![e_0\ e_1]\!]\ \rho\ \kappa_h\ \kappa = \mathcal{E}[\![e_0]\!]\ \rho\ \kappa_h\ (\lambda \textit{Fun}(f).\ \mathcal{E}[\![e_1]\!]\ \rho\ \kappa_h\ (\lambda v.\ \\ \mathcal{A}[\![a]\!]\ \mathsf{call}(\sigma(e_0))\ \kappa_h \\ (f\ (\lambda v.\ \boxed{\kappa_h}\ v)(\mathcal{A}[\![a]\!]\ \mathsf{reception}(\sigma(e_0))\ \kappa_h\ \kappa))\ v)) \end{aligned}$$

This continuation $\boxed{\kappa_h}$ corresponds to a failure join point. We therefore define the semantics of Application as Figure 15-(c), in a similar way to call and reception.

The above semantics clarifies the detailed behavior of the aspect mechanism with exception handling. For example, consider that an exception is to be thrown in an advice body, which runs at a call join point. It is not obvious whether other advice declarations matching the same join point shall be executed in this case. With the above semantics, we can easily tell that no declaration will be executed.

This is because the semantics of APPLICATION passes κ_h to the semantic $\mathcal{A}$ in order to execute advice at a call join point, like $\mathcal{A}[\![a]\!]\ \mathsf{call}(name)\ \kappa_h\ \ldots$, which means that the exception handler of the advice execution is the same one to the one of the function application.

6 Related Work

As far as we know, practical AOP languages with pointcut and advice, including AspectJ[15], AspectWerkz[3] and JBoss AOP[6], are all based on the region-in-time model. Therefore, the reusability problem in Section 2 is common to those languages even though they have mechanisms for aspect reuse.

A few formal studies[5, 9, 22] treat beginning and end of an event as different join points. However, motivations behind those studies are different from ours. MinAML[22] is a low-level language that serves as a target of translation from a high-level AOP language. Douence and Teboul's work[9] focuses on identifying calling contexts from execution history. Brichau et al.[5] attempt to provide a language model that generalizes many AOP languages.

Including the region-in-time and point-in-time models, previous formal studies focus on different properties of aspect-oriented languages. Aspect SandBox (ASB)[23] focuses on formalizing behavior of pointcut matching and advice execution by using denotational semantics. Since ASB is based on the region-in-time model, the semantics of advice execution has to have a rule for each advice modifier. MiniMAO$_1$[7] focuses on type soundness of `around` advice, based on ClassicJava style semantics. It is also based on the region-in-time model.

7 Conclusion

We proposed an experimental new join point model. The model treats ends of actions, such as returns from methods, as different join points from beginnings of actions. In PitJ, ends of actions can be captured solely by pointcuts, rather than advice modifiers. This makes advice declaration more reusable. Even with simplified advice mechanism, PitJ is as expressive as AspectJ in typical use cases.

We also gave a formal semantics of Pitλ, which simplified from PitJ. It is a denotational semantics in a continuation passing style, and symmetrically represents beginnings and ends of actions as join points. With the aid of the semantics, we investigated integration of advanced language features with the point-in-time join point model.

Our future work includes the following topics. We will integrate more advanced features, such as `dflow` pointcut[17], first-class continuation and tail-call elimination. We will also plan to implement compiler for PitJ languages.

Acknowledgments. We would like to thank Kenichi Asai, the members of the Principles of Programming Languages Group at University of Tokyo, the members of the Kumiki Project and the anonymous reviewers for their careful proofreading and valuable comments. An earlier version of the paper was presented

at the FOAL'06 workshop. We appreciate the comments from the workshop attendees, especially from Gregor Kiczales, Gary Leavens and Mira Mezini.

References

1. Allan, C., et al.: Adding trace matching with free variables to AspectJ. In: OOPSLA'05. (2005) 345–364
2. Bodkin, R., Almaer, D., Laddad, R.: aTrack: an enterprise bug tracking system using AOP. Demonstration at AOSD'04. (2004)
3. Bonér, J.: What are the Key Issues for Commercial AOP use: How Does AspectWerkz Address Them? In: AOSD'04. (2004) 5–6 Invited Industry Paper.
4. Brichau, J., Meuter, W.D., De Volder, K.: Jumping aspects. In: Workshop on Aspects and Dimensions of Concerns at ECOOP'00. (2000)
5. Brichau, J., et al.: An initial metamodel for aspect-oriented programming languages. AOSD-Europe-VUB-12, Vrije Universiteit Brussel. (2006)
6. Burke, B., Brok, A.: Aspect-oriented programming and JBoss. Published on The O'Reilly Network (2003) `http://www.oreillynet.com/pub/a/onjava/2003/05/28/aop_jboss.html`.
7. Clifton, C., Leavens, G.T.: MiniMAO: Investigating the semantics of proceed. In: FOAL'05. (2005)
8. Danvy, O., Filinski, A.: Abstracting control. In: LFP '90. (1990) 151–160
9. Douence, R., Teboul, L.: A pointcut language for control-flow. In: GPCE'04. (2004)
10. Douence, R., Fritz, T., Loriant, N., Menaud, J.M., Ségura-Devillechaise, M., Südholt, M.: An expressive aspect language for system applications with Arachne. In: AOSD'05. (2005) 27–38
11. Elrad, T., Filman, R.E., Bader, A.: Aspect-oriented programming. Communications of the ACM **44**(10) (2001) 29–32
12. Endoh, Y., Masuhara, H., Yonezawa, A.: Continuation join points. In: FOAL'06. (2006) 1–10
13. Endoh, Y.: Continuation join points. Master's thesis, Department of Computer Science, University of Tokyo (2006) Revised version is available at `http://www.graco.c.u-tokyo.ac.jp/ppp/projects/pit/`.
14. Harbulot, B., Gurd, J.R.: A join point for loops in AspectJ. In: AOSD'06. (2006) 63–74
15. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP'01, LNCS 2072. (2001) 327–353
16. Kiczales, G.: The fun has just begun. Keynote Speech at AOSD'03. (2003)
17. Masuhara, H., Kawauchi, K.: Dataflow pointcut in aspect-oriented programming. In: APLAS'03, LNCS 2895. (2003) 105–121
18. Rajan, H., Sullivan, K.: Aspect language features for concern coverage profiling. In: AOSD'05. 181–191
19. Usui, Y., Chiba, S.: Bugdel: An aspect-oriented debugging system. In: Proceedings of The First Asian Workshop on AOSD. (2005) 790–795
20. van Deursen, A., Marin, M., Moonen, L.: AJHotDraw: A showcase for refactoring to aspects. In: LATE'05. (2005)
21. Walker, R.J., Murphy, G.C.: Implicit context: Easing software evolution and reuse. In: FSE-8. ACM SIGSOFT Software Engineering Notes 25(6). (2000) 69–78
22. Walker, D., Zdancewic, S., Ligatti, J.: A theory of aspects. In: ICFP'03. (2003)
23. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. In: FOAL'02. (2002) 1–8