# Continuation Join Points

Yusuke Endoh
Department of Computer Science, University of Tokyo
mame@yl.is.s.u-tokyo.ac.jp

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

Akinori Yonezawa
Department of Computer Science, University of Tokyo
yonezawa@yl.is.s.u-tokyo.ac.jp

## ABSTRACT

In AspectJ-like languages, there are several different kinds of advice declarations, which are specified by using advice modifiers such as `before` and `after returning`. This makes semantics of advice complicated and also makes advice declarations less reusable since advice modifiers are not parameterized unlike pointcuts. We propose a simpler join point model and an experimental AOP language called PitJ. The proposed join point model has only one kind of advice, but has finer grained join points. Even though we unified different kinds of advice into one, the resulted language is sufficiently expressive to cover typical advice usages in AspectJ, and has improved advice reusability by allowing pointcuts, rather than advice modifiers, to specify when advice body runs. Along with the language design, this paper gives a formalization of the join point model in a continuation-passing style (CPS).

## 1. INTRODUCTION

One of the fundamental language mechanisms in aspect-oriented programming (AOP) is the *pointcut and advice* mechanism, which can be found in many AOP languages including AspectJ[12]. 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[3, 11, 18, 13, 10, 14].

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).
- 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[3, 11, 18, 13].

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[10], conditional branches[14], and local variable accesses[15] 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 Pit$\lambda$. Thanks to affinity with continuation passing style, the semantics gives a concise model with advanced features such as exception handling.

# 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 cross-cutting 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[12]. We assume that the base program receives user input as return values of `readLine` method in several classes.

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

**Figure 1: Logging aspect for the console version**

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())`.

```
 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():
 9     call(String *.readLine());
10 }
```

**Figure 2: Generic logging aspect and its application to the console version**

The generic logging aspect is reusable to log user's input from environment variables as shown in Figure 3. Note that we can achieve this without modifying the generic logging aspect.

```
1 aspect ConsoleAndEnvVarLogging
2   extends UserInputLogging {
3   pointcut userInput():
4     call(String *.readLine()) ||
5     call(String System.getenv(String));
6 }
```

**Figure 3: Logging aspect for console and environment variable**

## 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 4.

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 5.

```
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 }
```

**Figure 4: Logging aspect for the hybrid version**

```
 1 abstract aspect UserInputLogging2 {
 2   abstract pointcut userInputAsReturnValue();
 3   abstract pointcut userInputAsArgument(String s);
 4   after() returning(String s):
 5     userInputAsReturnValue() {
 6     log(s);
 7   }
 8   before(String s): userInputAsArgument(s) {
 9     log(s);
10   }
11   void log(String s) {
12     Log.add(s);
13   }
14 }
```

**Figure 5: Generic logging aspect that can log for both return values and arguments**

## 2.3 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*[1] *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.
- 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.

| 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**

Figures 6 and 7 illustrate the difference between the point-in-time join point model and region-in-time one.

Figure 8 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 8, 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.

[1]Older versions of AspectJ[12] have reception join points for representing different actions.

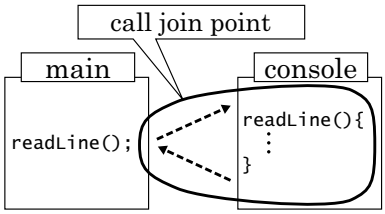


**Figure 6: Call join point in AspectJ-like languages**

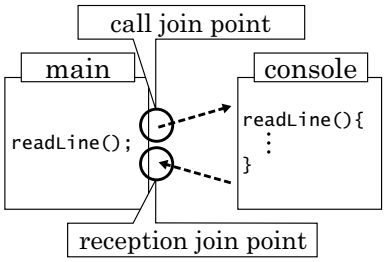


**Figure 7: Call and reception join points in PitJ**

```
 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):
 9     (reception(String *.readLine()) ||
10      call(* *.onSubmit(String))) && args(s);
11 }
```

**Figure 8: A logging abstract aspect and its application to the hybrid version in PitJ**

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[1] and AJHotDraw[16]. The details of the case study are presented in the other literature[9].

## 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 9 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 declaration[2].

In PitJ, 'termination by throwing an exception' is regarded as an independent failure join point. Figure 10 is an equivalent to the one in Figure 9. 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.

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

**Figure 9: An aspect to capture exceptions in AspectJ**

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

**Figure 10: An aspect to capture exceptions in PitJ**

## 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. The usages of around advice in AspectJ can be classified into the following four:

1. replacing the parameters to a join point with new ones,
2. replacing the return values to the caller of a join point,
3. going back to the caller without executing a join point, and
4. executing a join point more then once.

In PitJ, 1 and 2 are realized by using a `return` construct in an advice body. For example, the next advice declaration:

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

sanitizes user input by replacing meta-characters with escaped ones. When an advice body ends without `return`, the values in join points remain unchanged.

As for 3, we introduce a construct `skip`. When an advice declaration applied to a call join point evaluates `skip`, it jumps to the reception join point that corresponds to the current call join point *without executing subsequent advice declarations matching the call join point, and the call join*

*point itself.* When `skip` is evaluated at a reception or failure join point, it merely skips subsequent advice declarations matching the join points. For example, consider the next two advice declarations:

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

When `readLine()` is called, the first advice body immediately returns `"dummy"` to the caller without running the second advice and the body of `readLine`.

As for 4, we introduced a special function `proceed`. On a call join point, it executes the action until just before the subsequent reception join point that corresponds to the current call join point, and then returns the result of the call. On a reception or failure join point, `proceed` always returns the null. We show three examples of `proceed` below.

```
advice(): call(* *.readLine()) {
  String str = proceed();
}
```

The above advice performs the body of `readLine` by evaluating `proceed`, and performs `readLine` again after finishing the advice body. As a result, the method `readLine` skips every other line.

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

The second advice lets a call to `readLine` return a concatenation of two lines.

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

The above advice has no effect because the `proceed` executes the action until just before the reception join point that corresponds to the current call join point, and the `skip` jumps to the same reception join point.

Note that we introduced `skip` and `proceed` as a set of minimal constructs in order to realize the same functionalities to AspectJ's around advice. Further investigations would be needed in terms of conciseness and expressiveness in real-world applications.

## 3.4 More Advanced Features

Some existing AOP languages including AspectJ provide context sensitive pointcuts. 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 execution trace sensitive aspects[8, 7, 18], which use execution trace, or a history of generated join points, to judge whether to perform additional computation. We expect that our finer grained join points enhance its effectiveness and robustness.

# 4. FORMAL SEMANTICS

We present a formal semantics of Pit$\lambda$, which is a simplified version of PitJ. Pit$\lambda$ simplifies PitJ by using a lambda-calculus as a base language, and by supporting 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 11 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[6].

Syntax:

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

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:

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

**Figure 11: Syntax and semantics of the base language**

## 4.2 Syntax of Pit$\lambda_0$

We begin with Pit$\lambda_0$, which is a core part of Pit$\lambda$ that has only call and reception join points. Figure 12 displays the syntax.

## 4.3 Semantics of Pit$\lambda_0$

$$
\begin{array}{lll}
\text{(Expression)} & e ::= & x \qquad\qquad\qquad\qquad\qquad\qquad \text{(IDENTIFIER)} \\
& \mid & \texttt{fun}\ x \to e \qquad\qquad\qquad\qquad\quad \text{(FUNCTION)} \\
& \mid & e\ e \qquad\qquad\qquad\qquad\qquad\quad \text{(APPLICATION)} \\
\text{(Pointcut)} & p ::= & \texttt{call}(x) \mid \texttt{reception}(x) \\
& \mid & \texttt{args}(x) \mid p\ \texttt{\&\&}\ p \mid p\ \texttt{||}\ p \\
\text{(Advice)} & a ::= & \cdot \mid \texttt{advice} : p \to e;\ a
\end{array}
$$

**Figure 12: Pit$\lambda_0$ syntax**

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

**Figure 13: Semantics of pointcuts**

We give a semantics of Pit$\lambda_0$ by modifying the semantics of the base language in Section 4.1.

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

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

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

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

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 13. $\mathcal{P}[\![p]\!]\ \rho_{\mathit{empty}}\ \theta$ tests whether the pointcut $p$ and the current join point $\theta$ 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 14), 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.

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

**Figure 14: Semantics of advice**

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

**Figure 15: Semantics of expressions**

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 Pit$\lambda_0$ is defined by inserting application to $\mathcal{A}$ at appropriate positions. The original semantics of APPLICATION is as follows:

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

The shadowed part $f\ \kappa$ is a continuation that executes the function body and passes the result to the subsequent continuation $\kappa$. The application to the continuation $f\ \kappa\ v$, 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:

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

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 $\eta$-expanding[3] $\kappa$ as follows:

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

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

Figure 15 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 $\kappa$ that represents a join point,

[3]This $\eta$-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.

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. SEMANTICS OF ADVANCED FEATURES

In the section, with the aid of the clarified semantics, we investigated integration of advanced language features with the point-in-time join point model. Thus far, the following features are integrated into Pit$\lambda$: exception handling, context sensitive pointcuts and around advice. We call the integrated version Pit$\lambda_1$. For the sake of simplicity, we explain about each integration step orthogonally.

## 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[19, 17] 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 16 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

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

**Figure 16: Additional constructs for exception handling**

the special values which represent an exception; an arbitrary value can be raised. For example, `(fun x → raise x) 1` raises the value `1` as an exception. `try ((fun x → raise x) 1) + 2 with x → x+ 3` is evaluated normally to the value `4`. But, with `advice : failure(*) && args(x) → 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 $\mathcal{E}$:

$$f \in Fun = Ctn \to Ctn \to Ctn$$

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

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

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

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

The new definition of $\mathcal{A}$ is in Figure 17-(b). This modification, adding the shadowed parts, is mechanical since additional continuations are dealt with only by the additional

(a) Pointcuts (`failure` only):

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

(b) Advices:

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

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

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

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

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

$$\mathcal{E}[\![\texttt{try}\ e_0\ \texttt{with}\ x \to e_1]\!]\ \rho\ \kappa_h\ \kappa = \mathcal{E}[\![e_0]\!]\ \rho\ (\lambda v.\ \mathcal{E}[\![e_1]\!]\ ([v\ /\ x]\rho)\ \kappa_h\ \kappa)\ \kappa$$

$$\mathcal{E}[\![\texttt{raise}\ e]\!]\ \rho\ \kappa_h\ \kappa = \mathcal{E}[\![e]\!]\ \rho\ \kappa_h\ \kappa_h$$

**Figure 17: Semantics of Pit$\lambda_1$ with exception handling**

constructs. After that, we can define a semantics of the Try and the Raise as Figure 17-(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 17-(a).

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

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

This continuation $\kappa_h$ corresponds to a failure join point. We therefore define the semantics of Application as Figure 17-(c), in a similar way to call and reception.

The above semantics clarifies the detailed behavior of the case where aspect mechanism gets tangled up with exception handling. For example, consider the case where an exception is thrown in an advice declaration which corresponds to a call join point. Then, a question: "After that, will any advice declaration be executed?" See the semantic function $\mathcal{A}$ which represents advice execution in a call join point: $\mathcal{A}[\![a]\!]\ \mathsf{call}(name)\ \kappa_h\ \ldots$. It receives $\kappa_h$ as an exception handler directly. So, we can easily answer, "No advice declaration will be executed."

(Pointcut) $p ::= \ldots \mid$ `cflow(`$p$`)`

**Figure 18: `cflow` pointcut syntax**

## 5.2 Context Sensitive Pointcuts

The subsection describes how we integrate `cflow` pointcut, which is a kind of context sensitive pointcuts. The pointcut identifies join points based on whether they occur in the dynamic context during a region-in-time of other join points. For example, `cflow(call(* func(..)))` specifies each join point that occurs in the dynamic context during a region-in-time of the join points specified by `call(* func(..))`. In other words, this specifies each join point that occurs between when a `func` method is called and when it returns.

The context required by `cflow` is *call stack*. When a method is called, the call join point is pushed onto the stack. And the stack is popped at a reception join point.

First, we add `cflow(`$p$`)` to the pointcut (Figure 18). Its informal semantics is explained by example as follows. Consider an advice declaration:

```
advice : cflow(call(saveFile) && args(x)) && call(write)
       → log ("real save : " + x)
```

When the `write` method is called in the dynamic context during `saveFile`, or when `saveFile ("save.dat")` is executing, a string "`real save : save.dat`" is logged. Out of the dynamic context during `saveFile ("save.dat")`, a call to `write` makes no logging. Note that the pointcut `args(x)` binds the actual parameters of `saveFile`, not `write`. A `args` pointcut in a `cflow` binds the value of join point that is matched by the `cflow`.

We now define a formal semantics of a `cflow` pointcut. First, we modify the semantic algebras of join point and function:

$$\theta \in Jp = (Evt * Val * Jp) \mid Nil$$
$$f \in Fun = Jp \to Ctn \to Ctn$$

The semantic algebra $Jp$ comes to take the form of stack (or list) of join points; it represents the context required by `cflow`. And the semantic algebra $Fun$ receives a join point as well as a continuation. This additional argument is a call join point at which this function is called.

Along with the change, the semantic function of the pointcuts needs to be slightly modified:

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

Other pointcuts are similar. In addition, we add the semantic equation for the `cflow` pointcut (Figure 19-(a)). If the pointcut $p$ of `cflow(`$p$`)` matches the current join point (or the top of stack), $\mathcal{P}[\![\texttt{cflow(}p\texttt{)}]\!]$ returns the result environment. Otherwise, it tries to match the outer join point (or the next element of stack). This is repeated until $Nil$ (or stack is empty).

(a) Pointcuts (`cflow` and $Nil$ only):

$$\mathcal{P}[\![\texttt{cflow(}p\texttt{)}]\!]\ \rho\ ((\epsilon, v, \theta')\ as\ \theta) = \begin{cases} \rho' & \texttt{if } \mathcal{P}[\![p]\!]\ \rho\ \theta = \rho' \\ \mathcal{P}[\![\texttt{cflow(}p\texttt{)}]\!]\ \rho\ \theta' & otherwise \end{cases}$$
$$\mathcal{P}[\![p]\!]\ \rho\ Nil\ =\ False$$

(b) Advices:

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

(c) Expressions:

$$\mathcal{E} : \text{Expression} \to Env \to Jp \to Ctn \to Ans$$
$$\mathcal{E}[\![x]\!]\ \rho\ \theta\ \kappa = \kappa\ (\rho\ x)$$
$$\mathcal{E}[\![\texttt{fun}\ x \to e; a']\!]\ \rho\ \theta\ \kappa = \kappa\ (inFun(\lambda\theta'\kappa' v.\ \mathcal{E}[\![e]\!]\ ([v\ /\ x]\rho)\ \theta'\ \kappa'))$$
$$\mathcal{E}[\![e_0\ e_1]\!]\ \rho\ \theta\ \kappa = \mathcal{E}[\![e_0]\!]\ \rho\ \theta\ (\lambda Fun(f).\ \mathcal{E}[\![e_1]\!]\ \rho\ \theta\ (\lambda v.\ \mathcal{A}[\![a]\!]\ \mathsf{call}(\sigma(e_0))\ \theta\ (f\ (\mathsf{call}(\sigma(e_0)), v, \theta)\ (\mathcal{A}[\![a]\!]\ \mathsf{reception}(\sigma(e_0))\ \theta\ \kappa))\ v))$$

**Figure 19: Semantics of Pit$\lambda_1$ with `cflow` pointcut**

(Expression) $e ::= \ldots \mid$ `skip` $e$ (Skip)

**Figure 20: `skip` syntax**

The semantics of the advice has to be similarly modified too (Figure 19-(b)).

Finally, we modify the semantic function of the expression (Figure 19-(c)). In the semantics of Application, the function's argument $(\mathsf{call}(\sigma(e_0)), v, \theta)$ is a dynamic context. And, in the semantics of Function, the semantic lambda function receives a dynamic context.

## 5.3 Around Advice Modifier

As described in Subsection 3.3, we introduce a construct `skip` (Figure 20). A special function `proceed` is also added.

We here have two options: when integrating only `skip`, and when integrating both `skip` and `proceed`. If only `skip` is required, we integrate it by only adding a continuation which represents current skip handler. This way is very similar to exception handling (Subsection 5.1), so we omit explanation. Although we feel that it may be convenient enough without `proceed`, it's not to say that we can not integrate both. But we need a technique like *partial continuation*[6]. It is *a part of* the rest of computation, rather than the whole

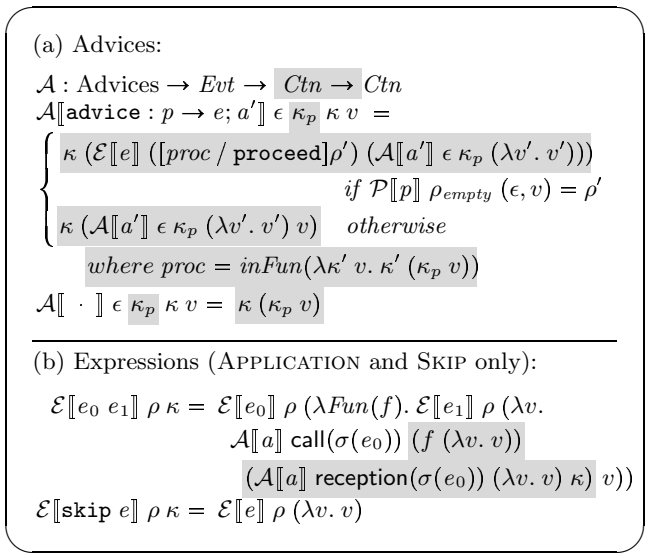


**Figure 21: Semantics of Pit$\lambda_1$ with around advice**

rest of computation as in the full continuation. We use a partial continuation to represent a region-in-time which may be skipped or be run more than once.

In what follows, we give a denotational semantics of Pit$\lambda_1$ in a *continuation composing style* (CCS). It allows some kinds of nested function application unlike CPS. Although it loses the CPS's important property, enforcing strict call-by-value evaluation, we know that it can be restored by converting the definition once more into CPS.

Now, we give the semantics of `skip` and `proceed` by using a partial continuation. We first add a partial continuation which represents the current `proceed` function.

$$f \in Fun = Ctn \to Ctn \to Ctn$$

And next we modify the semantics of advice (Figure 21-(a)). Additional continuation $\kappa_p$ is a partial continuation that represents the action until an appropriate join point, not until program termination. So, $\kappa\ (\kappa_p\ v)$ executes first a partial continuation $\kappa_p$ and then the rest of continuation $\kappa$. Such applications are not permitted in CPS, but CCS allows.

Finally, we define the semantics of the expressions (Figure 21-(b)). In the Application, $(f\ (\lambda v.\ v))$ corresponds to `proceed` of a call join point, and $(\lambda v.\ v)$ corresponds to the one of a reception join point. The Skip evaluates the argument, and *does not apply* the result to the continuation. This allows jumping from a call join point to the counterpart, or the following reception join point, without execution between the two join points.

## 6. RELATED WORK

As far as we know, practical AOP languages with pointcut and advice, including AspectJ[12], AspectWerkz[2] and JBoss AOP[4], 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, such as MinAML[17], treat beginning and end of an event as different join points. However, motivations behind those studies are different from ours. MinAML is a low-level language that serves as a target of translation from a high-level AOP language. Douence and Teboul's work[8] focuses on identifying calling contexts from execution history.

Including the region-in-time and point-in-time models, previous formal studies focus on different properties of aspect-oriented languages. Aspect SandBox (ASB)[19] 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. Tucker and Krishnamurthi[**?**] presented a pointcut and advice mechanism for higher-order languages and implemented a prototype on top of PLT Scheme. The pointcuts in their mechanism are first-class entities, and can be parameterized. Although the design could improve reusability of advice declarations, their mechanism is based on the region-in-time model; hence it can not uniformly treat beginnings and ends of actions. MiniMAO$_1$[5] 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 Pit$\lambda$, 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[13], first-class continuation and tail-call elimination. We will also plan to implement compiler for PitJ languages.

## 8. 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. We would also like to thank the anonymous reviewers.

## 9. REFERENCES

[1] R. Bodkin. aTrack. https://atrack.dev.java.net/.

[2] J. Bonér and A. Vasseur. AspectWerkz. http://aspectwerkz.codehaus.org/.

[3] J. Brichau, W. D. Meuter, and K. De Volder. Jumping aspects. In C. Lopes, L. Bergmans, M. D'Hondt, and P. Tarr, editors, *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

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

[5] C. Clifton and G. T. Leavens. MiniMAO: Investigating the semantics of proceed. In G. T. Leavens, C. Clifton, and R. Lämmel, editors, *Foundations of Aspect-Oriented Languages*, Mar. 2005.

[6] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 151–160, New York, NY, 1990. ACM.

[7] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. S udholt. An expressive aspect language for system applications with arachne. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[8] R. Douence and L. Teboul. A pointcut language for control-flow. In G. Karsai and E. Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 95–114. Springer, 2004.

[9] Y. Endoh. Continuation join points. Master's thesis, Department of Computer Science, University of Tokyo, 2006. to appear.

[10] B. Harbulot and J. R. Gurd. A join point for loops in aspectj. In *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, Mar. 2005.

[11] G. Kiczales. Making the code look like the design. In *AOSD 2003*, 2003. keynote.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[13] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In A. Ohori, editor, *APLAS*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

[14] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 181–191, New York, NY, USA, 2005. ACM Press.

[15] Y. Usui and S. Chiba. Bugdel: An aspect-oriented debugging system. In *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pages 790–795. IEEE Press, 2005.

[16] A. van Deursen, M. Marin, and L. Moonen. AJHotDraw: A showcase for refactoring to aspects. In T. Tourwé, A. Kellens, M. Ceccato, and D. Shepherd, editors, *Linking Aspect Technology and Evolution*, Mar. 2005.

[17] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003. ACM Press.

[18] R. J. Walker and G. C. Murphy. Implicit context: easing software evolution and reuse. *SIGSOFT Softw. Eng. Notes*, 25(6):69–78, 2000.

[19] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.