FOAL 2006 Proceedings

Foundations of Aspect-Oriented Languages
Workshop at AOSD 2006

Curtis Clifton, Ralf Lämmel, and Gary T. Leavens (editors)

TR #06-01
March 2006

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Contents

Preface

FOAL logos courtesy of Luca Cardelli

Aspect-oriented programming is a paradigm in software engineering and programming languages that promises better support for separation of concerns. The fifth Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the Fifth International Conference on Aspect-Oriented Software Development in Bonn, Germany, on March 21, 2006. This workshop was designed to be a forum for research in formal foundations of aspect-oriented programming languages. The call for papers announced the areas of interest for FOAL as including: semantics of aspect-oriented languages, specification and verification for such languages, type systems, static analysis, theory of testing, theory of aspect composition, and theory of aspect translation (compilation) and rewriting. The call for papers welcomed all theoretical and foundational studies of foundations of aspect-oriented languages.

The goals of this FOAL workshop were to:

- Make progress on the foundations of aspect-oriented programming languages.
- Exchange ideas about semantics and formal methods for aspect-oriented programming languages.
- Foster interest within the programming language theory and types communities in aspect-oriented programming languages.
- Foster interest within the formal methods community in aspect-oriented programming and the problems of reasoning about aspect-oriented programs.

The workshop was organized by Curtis Clifton (Rose-Hulman Institute of Technology), Gary T. Leavens (Iowa State University), and Ralf Lämmel (Microsoft). The program committee was chaired by Mira Mezini (Darmstadt University of Technology) and included Mezini, Clifton, Jonathan Aldrich (Carnegie Mellon University), Don Batory (University of Texas, Austin), Paulo Borba (Universidade Federal de Pernambuco), Marc Eaddy (Columbia University), Robby Findler (University of Chicago), Matthew Flatt (University of Utah), Pascal Fradet (INRIA), Alan Jeffrey (Bell Labs), Shmuel Katz (Technion–Israel Institute of Technology), John Lefor (Microsoft Research), Karl Lieberherr (Northeastern University), Todd Millstein (University of California, Los Angeles), Oege de Moor (Oxford University), Hridesh Rajan (Iowa State University), David Walker (Princeton University), and Mitchell Wand (Northeastern University). We thank the organizers of AOSD 2006 for hosting the workshop.

Message from the Program Committee Chair

This volume contains the papers presented at FOAL '06, the 5th Workshop on Foundations of Aspect-Oriented Languages. The FOAL series of workshops organized in conjunction with the International Conference on Aspect-Oriented Software Development (AOSD) focusses on the theory and principles behind aspect-oriented programming language design and implementation. This year's FOAL workshop was held in conjunction with AOSD '06 in Bonn, Germany, on Tuesday, the 21st of March, 2006.

This year we received a total of 15 submissions. Each paper was reviewed by a minimum of four reviewers and some papers received five reviews. After the initial reviews were submitted, the program committee discussed each paper during a 3-day online program committee meeting held between February 7th and 9th. The final program includes 5 long and 2 short papers. Also, the authors of three submissions were invited for a short presentation.

I am very grateful to the program committee for their hard work in reading, reviewing and discussing all the submissions and for providing thorough feedback to authors. I am also very grateful to Shriram Krishnamurthi and Jay McCarthy from Brown University for giving us access to the Brown Continue Server which we used to administer the reviewing process. Last but not least, I am very grateful to the program organizers, Curtis Clifton, Ralf Lämmel and Gary T. Leavens, for their extraordinary and indispensable support during the whole process. I thank them for all their hard work organizing the details that made this workshop possible.

Sincerely,

Mira Mezini
FOAL 06 Program Chair
Darmstadt University of Technology

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 Pitλ. 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.

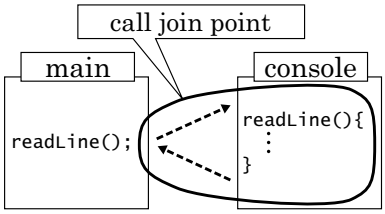


Figure 6: Call join point in AspectJ-like languages

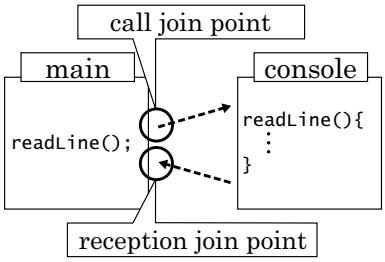


Figure 7: Call and reception join points in PitJ

```
 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 Pitλ, which is a simplified version of PitJ. Pitλ 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:

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

Figure 11: Syntax and semantics of the base language

4.2 Syntax of Pitλ_0

We begin with Pitλ_0, which is a core part of Pitλ that has only call and reception join points. Figure 12 displays the syntax.

4.3 Semantics of Pitλ_0

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

Figure 12: Pitλ_0 syntax

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

Figure 13: Semantics of pointcuts

We give a semantics of 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:

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

Additionally, we define an auxiliary function σ 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 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 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

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

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 Pitλ_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 κ. 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 η-expanding[3] κ 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 κ 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 κ that represents a join point,

[3]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.

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 Pitλ: exception handling, context sensitive pointcuts and around advice. We call the integrated version Pitλ_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 $\mathcal{E}$:

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

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

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

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

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

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:

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

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

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

(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 Pitλ_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 κ_h in $f\ \kappa_h\ \ldots$, show up the application form by η-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 κ_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 κ_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 Pitλ_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

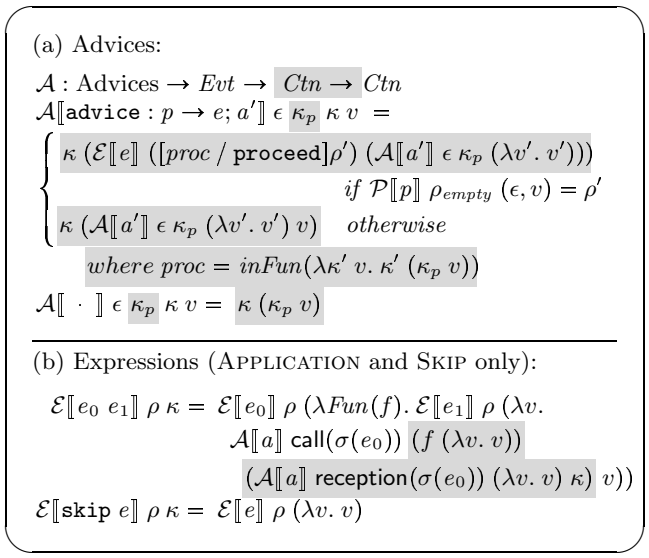


Figure 21: Semantics of Pitλ_1 with around advice

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 Pitλ_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 κ_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 κ_p and then the rest of continuation κ. 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 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[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.

Modular Generic Verification of LTL Properties for Aspects

Max Goldman Shmuel Katz
Computer Science Department
Technion — Israel Institute of Technology
{mgoldman, katz}@cs.technion.ac.il

ABSTRACT

Aspects are separate code modules that can be bound ("woven") to a base program at joinpoints to provide an augmented program. A novel approach is defined to verify that an aspect state machine will provide desired properties whenever it is woven over a base state machine that satisfies the assumptions of the aspect. A single state machine is constructed using the tableau of the linear temporal logic (LTL) description of the assumptions, a description of the joinpoints, and the state machine of the aspect code. A theorem is shown that if the constructed machine satisfies the desired properties, so will an augmented state machine using any base machine that satisfies the assumptions. The theorem is stated and shown for assumptions and properties given in LTL, for a somewhat restricted form of joinpoint description, and for aspect code that ends in states already reachable in the base state machine. A language-based description of aspects, as in AspectJ, can be converted to a state machine version using existing tools, thus providing generic modular verification of code-level aspects.

1. INTRODUCTION

1.1 Aspect-Oriented Programming

The aspect-oriented approach to software development is one in which concerns that cut across many parts of the system are encapsulated in separate modules called *aspects*. For example, when security or logging are encapsulated in an aspect, this aspect contains both the code associated with the concern, called *advice*, and a description of when this advice should run, called a *pointcut descriptor*. The pointcut descriptor identifies those points in the execution of a program at which the advice should be invoked. The combination of some *base program* with an aspect (or in general, a collection of aspects), is termed an *augmented program*.

1.2 Formal Verification

In this work we are concerned with generic formal verification of aspects relative to a specification. The specification of an aspect consists of *assumptions* about any base program to which the aspect can reasonably be woven, and *desired properties* intended to hold for the augmented program. We view both base programs and aspect code as nondeterministic finite state machines, in which particular computations are realized as infinite sequences of states within the machine. For both assumptions and desired properties to be verified we consider formulas written in linear temporal logic (LTL). An LTL formula consists of a *path formula* using *temporal quantifiers* and logical combinations of *atomic propositions*, prefixed by a single (usually implicit) *universal path quantifier*. The atomic propositions in a formula refer to the labels of states in a finite state machine; temporal quantifiers specify when these assertions about states must be true. The universal path quantifier requires that, in order for some initial state to satisfy an LTL assertion, all infinite paths from that state must satisfy the path formula. In general, a state machine also includes a fairness constraint, and only fair paths are considered.

1.3 Modular Aspectual Verification

It is clear that given a base program, a collection of aspects with their pointcut descriptors and advice, and a system for *weaving* together these components to produce a stand-alone augmented program, we can verify properties of this augmented system using the usual model checking techniques. Such weaving involves adding edges from joinpoint states of the base program to the initial states of the advice, and from the states after an advice segment to states where base program statements are executed.

It would be preferable, however, if we could employ a modular technique in which the aspect can be considered separately from the base program. This would allow us to:

- obtain verification results that hold for a particular aspect with any base program from some class of programs, rather than for only one base program in particular;
- use the results to reason about the application of aspects to base programs with multiple evolving state machines describing changing configurations during execution, or to other base systems not amenable to model checking; and
- avoid model checking augmented systems, which may be significantly larger than their base systems, and whose unknown behavior may resist abstraction.

The second point above relates to general object-oriented programs that create new instances of classes (objects) with associated state machine components. Often, the assumption of an aspect about the key properties of those base state machines to which it may be woven can indeed be shown to hold for every possible machine that corresponds to an object configuration of a program. For example, it may involve a so-called *class invariant*, provable by reasoning directly on class declarations, as in [1]. This point and more details on the connections between code-based aspects (as in AspectJ) and the state machine versions seen here are discussed in Section 5.

This problem of creating a single generic model that can represent any possible augmented program for an aspect woven over some class of base programs is especially difficult because of the aspect-oriented notion of *obliviousness*: base programs are generally unaware of aspects advising them, and have no control over when or how they are advised. There are no explicit markers for the transfer of control from base to advice code, nor are there guarantees about if or where advice will return control to the base program.

1.4 Results

In this paper we show how to verify once-and-for-all that for any base state machine satisfying the assumptions of the aspect, and for a weaving that adds the aspect advice as indicated in the joinpoint description, the resulting augmented state machine is guaranteed to satisfy the desired properties given in the specification. A single generic state machine is constructed from the tableau of the assumption, the pointcut descriptor, and the advice state machine, and verified for the desired property. Then, when a particular base program is to be woven with the aspect, it is sufficient to establish that the base state machine satisfies the assumption. Thus the entire augmented program never has to be model checked, achieving true modularity and genericity in the proof. This approach is especially appropriate for aspects intended to be reused over many base programs, e.g., those in libraries or middleware components.

LTL model checking is based on creating a tableau state machine automaton that accepts exactly those computations that satisfy the property to be verified. Usually, the negation of this machine is then composed as a cross-product with the model to be checked. A counter-example is produced when the composed system contains some infinite path, and the property is satisfied for the model when the cross-product has no such paths. Here we use the tableau of the assumption in a unique way, as the basis of the generic model to be checked for the desired property. It represents any base machine satisfying the assumption, because the execution sequences of the base program can be abstracted by sequences in the tableau.

For the soundness theorem presented in Section 4, the aspects treated are assumed to be *weakly invasive*, as defined in [7]. This means that when advice has completed executing, the system continues from a state that was already reachable in the original base program (perhaps for different inputs or actions of the environment). Many aspects fall into this category, including *spectative* aspects that never modify the state of the base system (logging is a good example), and *regulative* aspects that only restrict the reachable state space (for example, aspects implementing security checks). Also weakly invasive would be an aspect to enforce transactional requirements, which might roll back a series of changes so that the system returns to the state it was in before they were made. Even a 'discount policy' aspect that reduces the price on certain items in a retail system is weakly invasive, since the original price given as input could have been the discounted one.

Additionally, we assume that any executions of an augmented program that infinitely often include states resulting from aspect advice will be fair (and thus must be considered for correctness purposes). The version here does not treat multiple aspects or joinpoints influenced by the introduction of advice, although the approach can be expanded to treat such cases as well.

In the following section, needed terms and constructs are defined. Section 3 presents the algorithm, and Section 4 gives a proof of soundness in the weakly invasive aspect case. This section also gives an example. Section 5 details works related to the result here, and is followed by the conclusion.

2. DEFINITIONS

2.1 LTL Tableaux

Intuitively, the tableau of an LTL forumla f is a state machine whose fair infinite paths are exactly all those paths which satisfy the formula f. This intuition will be realized formally in Theorem 1 below.

In the context of performing model checking to verify satisfaction of an LTL property, a tableau is constructed for the *negation* of that property, in order to capture all possible computations that would cause a machine *not* to satisfy the formula in question. It is important to stress that here we use the tableau for the original non-negated formula. Nevertheless, because of the use of tableaux by LTL model checking tools, modules to perform the construction of a formula's tableau are available. For exploratory purposes, the authors have used the translator module of NuSMV [10], which produces a (tableau) finite state machine from a given LTL formula.

We define T_f, the tableau for LTL path formula f, as given in *Model Checking* [3] in the section on "Symbolic LTL Model Checking" (6.7). In this construction, the original formula is decomposed into the set of *elementary formulas* it contains, where all other temporal operators, such as *from now on* (G) and *eventually* (F), are expressed in terms of *next* (X) and *strong until* (U). Each state in the tableau is a subset of these elementary formulas, and the path relation between these states is defined by means of a function $sat(g)$, which captures the set of states in which subformula g of f is satisfied.

We denote $T_f = (S_T, S_0^T, R_T, L_T, F_T)$, where S_T is the set of states; S_0^T is the set of initial states, R_T is the transition relation, L_T is the labeling function, and F_T is the set of fair state sets. For ease of discussion, we clarify the definition as follows:

Define S_0^T, where for $\chi = Af$, we have $T_f \models \chi$:

$$S_0^T = sat(f)$$

Define F_T, where any fair path in T_f must visit each set in F_T infinitely many times:

$$F_T = \{sat((\neg(g \mathsf{U} h)) \vee h) \mid g \mathsf{U} h \text{ is a subformula of } f\}$$

This fairness constraint guarantees that obligations of the form $g \mathsf{U} h$ are fulfilled, either by visiting a state in $sat(h)$ infinitely often, or by infinitely often visiting a state outside of $sat(g \mathsf{U} h)$, which can only be reached by going via $sat(h)$ according to the construction of the path relation (not detailed here).

Two notable properties of T_f will be used below. First, if AP_f is the set of atomic propositions in f, then $L_T : S \to \mathcal{P}(AP_f)$ — that is, the labels of the states in the tableau will include sets of the atomic propositions appearing in f. A state in any machine is given a particular label if and only if that atomic proposition is true in that state.

The second interesting feature is a main theorem from the discussion in [3]:

Definition 1. For path π, let $label(\pi)$ be the sequence of labels (subsets of AP) of the states of π. For such a sequence $l = l_0, l_1, \ldots$ and set Q, let $l|_Q = m_0, m_1, \ldots$ where for each $i \geq 0$, $m_i = l_i \cap Q$.

THEOREM 1. *Given T_f, for any Kripke structure M, for all fair paths π' in M, if $M, \pi' \models f$ then there exists fair path π in T_f such that π starts in S_0^T and $label(\pi')|_{AP_f} = label(\pi)$.*

That is, for any possible computation of M satisfying formula f, there is a path in the tableau of f which matches the labels within AP_f along the states of that computation.

In the algorithm of Section 3, we restrict the tableau to its reachable component. Such restriction does not affect the result of this theorem, since all reachable paths are preserved, but is necessary in order to achieve useful results. This follows from the observation that the tableau for the negation of a formula has precisely the same states and transition relation, but the complementary set of initial states. Thus, any unreachable portion of the tableau is liable to contain exactly those behaviors which violate the formula of interest.

Finally, for $\chi = Af$, define $T_\chi = T_f$ as a convenient notation (a tableau can only be constructed for a path formula).

2.2 Aspects

An aspect machine $A = (S_A, S_0^A, S_{ret}^A, R_A, L_A)$ over atomic propositions AP is defined as usual for a machine with no fairness constraint, with the following addition:

Definition 2. S_{ret}^A is the set of *return states* of A, where $S_{ret}^A \subseteq S_A$ and for any state $s \in S_{ret}^A$, s has no outgoing edges.

2.3 Pointcuts

We do not give a prescriptive definition for pointcut descriptors here; in practice pointcut descriptions might take a number of forms. However, we require that descriptors operate in the following manner:

Definition 3. Given a pointcut descriptor ρ over atomic propositions AP and a finite sequence l of labels (subsets of AP), we can ask whether or not the end of l is *matched* by ρ, written $l \models\!\!\!| \rho$.

A reasonable choice for describing pointcuts might be LTL path formulas containing only past temporal operators. For example, the descriptor $\rho_1 = a \wedge \mathsf{Y}\, b \wedge \mathsf{Y}\, \mathsf{Y}\, b$ would match sequences ending with a state where a is true, preceded by b, preceded by another b (operator Y is the past analogue of X). Other languages could be imagined, for example regular expressions, where $\rho_2 = \mathsf{true}^* \cdot b \cdot b \cdot a$ might be equivalent to ρ_1. The use of regular expressions over automata is popular in industrial specification languages and has been examined in formal combination with LTL for example in [2].

2.4 Specifications

In addition to its advice, in the form of state machine A, and pointcut, described by ρ, an aspect is considered to have two pieces of formal specification:

- Formula ψ expresses the assumptions made by the aspect about any base machine to which it will be woven. This ψ is thus a requirement to be met by any such machine.
- Formula ϕ expresses the desired result to be satisfied by any augmented machine built by weaving this aspect with a conforming base machine. In other words, ϕ is the guarantee of the aspect.

2.5 Weaving

Weaving is the process of combining a base machine with some aspect according to a particular pointcut descriptor; the result is an augmented machine that includes the advice of the aspect.

The weaving algorithm has the following inputs:

- aspect machine $A = (S_A, S_0^A, S_{ret}^A, R_A, L_A)$ over AP,
- pointcut ρ over AP, and
- base machine $B = (S_B, S_0^B, R_B, L_B, F_B)$ over $AP_B \supseteq AP$.

And it produces as output:

- augmented machine $\widetilde{B} = (S_{\widetilde{B}}, S_0^{\widetilde{B}}, R_{\widetilde{B}}, L_{\widetilde{B}}, F_{\widetilde{B}})$.

The weaving is performed in two steps. First we construct from the base machine B a new state machine B^ρ which is *pointcut-ready* for ρ, wherein each state either definitely is or is not matched by ρ. Then we use B^ρ and A to build the final augmented machine $\widetilde{B}$.

This two-step division of the weaving process means that the algorithm cannot handle a number of problematic cases: when the pointcut descriptor matches advice states, and thus advice should be inserted on other advice; when the addition of advice states creates a new matching pointcut in the computation, and advice should be inserted; and when the addition of advice states causes a location that once matched a pointcut selector not to match it any longer. Proper handling of these scenarios is the subject of ongoing work.

2.5.1 Constructing a Pointcut-Ready Machine

Pointcut-ready machine $B^\rho = (S_{B^\rho}, S_0^{B^\rho}, R_{B^\rho}, L_{B^\rho}, F_{B^\rho})$ is a machine in which unwinding of certain paths has been performed, so that we can separate paths which match pointcut descriptor ρ from those that do not. The pointcut-ready machine contains states with a new label, *pointcut*, that indicates exactly those states where the descriptor has been matched.

This machine must meet the following requirements:

- $S_{B^\rho} \supseteq S_B$
- $S_0^{B^\rho} = S_0^B$
- L_{B^ρ} is a function from S_{B^ρ} to $\mathcal{P}\left(AP_B \cup \{pointcut\}\right)$
- For all finite-length paths $\pi = s_0, \ldots, s_k$ in B^ρ such that $s_0 \in S_0^{B^\rho}$, $label(\pi) \models \rho \Leftrightarrow s_k \models pointcut$.
- For all infinite sequences of labels $l = (\mathcal{P}(AP_B))^\omega$, there is a fair path π_{B^ρ} in B^ρ where $label(\pi_{B^\rho})|_{AP_B} = l$ if and only if there is a fair path π_B in B where $label(\pi_B) = l$.

Note that since B and B^ρ have the same paths (over AP, ignoring the added *pointcut* label), they must satisfy exactly the same LTL formulas over AP.

Figure 1 shows a simple example of this construction. Note that in state diagrams, the absence of an atomic proposition indicates that the proposition does not hold, not that the value is unknown or irrelevant. This is in contrast to a formula, where unmentioned propositions are not restricted.

2.5.2 Constructing an Augmented Machine

We construct the components of augmented machine $\widetilde{B} = (S_{\widetilde{B}}, S_0^{\widetilde{B}}, R_{\widetilde{B}}, L_{\widetilde{B}}, F_{\widetilde{B}})$ as follows:

- $S_{\widetilde{B}} = S_{B^\rho} \cup S_A$
- $S_0^{\widetilde{B}} = S_0^{B^\rho}$

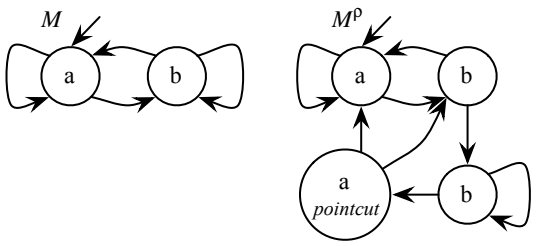


Figure 1: Constructing a pointcut-ready machine M^ρ for the given M and LTL past formula pointcut descriptor $\rho = a \wedge \mathbf{Y}\, b \wedge \mathbf{Y}\,\mathbf{Y}\, b$.

- $(s,t) \in R_{\widetilde{B}} \Leftrightarrow$

$$\begin{cases} (s,t) \in R_{B^\rho} \ \wedge \ s \not\models pointcut & \text{if } s,t \in S_{B^\rho} \\ (s,t) \in R_A & \text{if } s,t \in S_A \\ s \models pointcut \ \wedge \ t \in S_0^A & \\ \qquad \wedge\ L_{B^\rho}(s)|_{AP} = L_A(t) & \text{if } s \in S_{B^\rho},\ t \in S_A \\ s \in S_{ret}^A \wedge L_A(s) = L_{B^\rho}(t)|_{AP} & \text{if } s \in S_A,\ t \in S_{B^\rho} \end{cases}$$

Note that this relationship is 'if and only if.' In words, the path relation contains precisely all the edges from the pointcut-ready base machine B^ρ and from aspect machine A, except that *pointcut* states in B^ρ have edges only to matching start states in A, and aspect return states have edges to all matching base states.

- $L_{\widetilde{B}}(s) = \begin{cases} L_{B^\rho}(s) & \text{if } s \in S_{B^\rho} \\ L_A(s) & \text{if } s \in S_A \end{cases}$
- $F_{\widetilde{B}} = F_{B^\rho} \times S_A$

That is, $F_{\widetilde{B}} = \{F_i \cup S_A \mid F_i \in F_{B^\rho}\}$. A path is fair if it either satisfies the fairness constraint of the pointcut-ready machine, or if it visits some aspect state infinitely many times — a conservatively inclusive definition.

A weaving is considered *successful* if every reachable node in $S_{\widetilde{B}}$ has a successor according to $R_{\widetilde{B}}$.

2.6 Weakly Invasive Aspects

As mentioned above, we show our result for the broad class of aspects which, when they return from advice, do so to a reachable state in the base machine. Without this restriction, the aspect may return to unreachable parts of the base machine whose behavior is not bound by assumption formula ψ. In this case, the augmented system contains portions with unknown behavior, and is difficult to reason about in a modular way.

Definition 4. An aspect A and pointcut ρ are said to be *weakly invasive* for a base machine B if, for all states in S_{B^ρ} that are reachable by a fair path in $\widetilde{B}$, those states were reachable by a fair path in B^ρ.

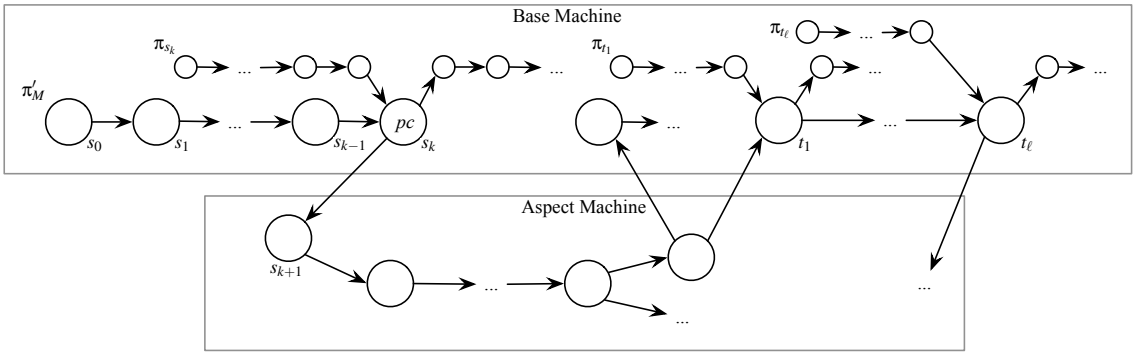


Figure 2: Using fair paths in M (small states along the top) to guarantee a matching path in $\widetilde{T_\psi}$.

In particular, this means that all states to which the aspect returns are states reachable in the pointcut-ready base machine. This could of course be checked directly, but would require construction of the augmented machine — precisely the operation we would like to avoid. In many cases, the aspect can be shown weakly invasive for any base machine satisfying its assumption ψ, by using static analysis, local model checking, or additional information (our reasoning in the discount price example from Section 1.4 uses such information). For further discussion, see [7].

3. ALGORITHM

The algorithm builds a tableau from ψ and weaves A with this tableau according to ρ, then performs model checking to verify the result with respect to ϕ. In the following section we prove that when this model check of the constructed augmented tableau succeeds, then for any base system satisfying requirement ψ, applying aspect A according to pointcut descriptor ρ will yield an augmented system satisfying result ϕ.

Given:

- set of atomic propositions AP;
- assumption ψ for base systems, an LTL formula over AP;
- desired result ϕ for augmented systems, an LTL formula over AP; and
- aspect machine A and pointcut descriptor ρ over AP.

Perform the following:

0. If it does not already, augment ψ with clauses of the form $\cdots \wedge (a \vee \neg a)$, such that ψ contains every atomic proposition $a \in AP$, without altering its meaning.
1. Construct T_ψ, the tableau for ψ. Since ψ contains every AP, the result of Theorem 1 will hold when all labels in AP are considered.
2. Restrict T_ψ to only those states reachable via a fair path.
3. Weave A into T_ψ according to ρ, obtaining $\widetilde{T_\psi}$.
4. Perform model checking in the usual way to determine if $\widetilde{T_\psi} \models \phi$.

4. CORRECTNESS

Given the components defined above, suppose that:

$$\widetilde{T_\psi} \models \phi \ .$$

What we have shown, then, is that the tableau for assumption ψ woven with aspect A according to ρ gives a resulting machine that satisfies desired augmented result ϕ. Our goal is to use the properties of $\widetilde{T_\psi}$ to show that A and ρ, when woven with *any* possible base machine M for which $M \models \psi$, will *always* yield an augmented $\widetilde{M}$ such that $\widetilde{M} \models \phi$. The proof below gives this result for a particular class of aspects.

THEOREM 2. *Given AP, ψ, ϕ, A, and ρ as defined, if $\widetilde{T_\psi} \models \phi$, then for any base program program M over a superset of AP such that A and ρ are weakly invasive for M, if $M \models \psi$ then $\widetilde{M} \models \phi$.*

PROOF. Since M and T_ψ contain exactly the same fair paths as M^ρ and T_ψ^ρ, and $M \models \psi$, by Theorem 1, for any fair path π_M in $M^\rho \models \psi$ starting from $S_0^{M^\rho}$, there is a fair path π_T in T_ψ^ρ with the same labels (restricted to AP). It suffices to show that after augmenting both of these pointcut-ready machines, this correspondence still holds.

Consider any fair path π'_M in $\widetilde{M}$ starting from an initial state.

Unmodified path Suppose no state on π'_M is labeled with *pointcut*. Then π'_M must be the same as some fair path π_M in M^ρ, which has matching fair path π_T in T_ψ^ρ. This path π_T contains no states labeled with *pointcut*, since for every finite subpath of π_M, ρ was not matched, and the labels on π_T are the same (restricted to AP).

Since π_T has no states labeled with *pointcut*, by the construction of $\widetilde{T_\psi}$, none of the edges along this path

have been removed during weaving. Therefore π_T is identical to a fair path π'_T in $\widetilde{T_\psi}$, and we have a matching path for π'_M.

Modified path Path π'_M must begin with a sequence of $k+1$ states $s_0, s_1, \ldots, s_k$ in M^ρ, where $k \geq 0$. Since s_k must be reachable from a fair path π_{s_k} in M^ρ, we can consider the path which begins $s_0, \ldots, s_k$ and continues along the remainder of π_{s_k} after s_k (see Figure 2). This path must also be fair, since it has the same infinite tail as π_{s_k} itself, and so must have a matching path in T^ρ_ψ; we begin π'_T by following this path.

Suppose that $s_k \models \mathit{pointcut}$, so s_{k+1} is in A. This state s_k can be labeled *pointcut* if and only if we have $\mathit{label}(s_0, \ldots, s_k) \equiv \rho$. In this case, the matching subpath in T^ρ_ψ must also match ρ, and will have *pointcut* on the state matching s_k. From both states, all edges go to machine A, so we can continue π'_T along an identical advice path; this includes the case when the advice never returns to the base machine.

If and when π'_M follows an edge from an aspect return state to a state t_1 in M^ρ, it does so to a state which is on a fair path π_{t_1} in that machine. There must be a matching path to π_{t_1} in the tableau. Furthermore, if we continue along a sequence of base machine states $t_1, \ldots, t_\ell$, since t_ℓ is also reachable from a fair path π_{t_ℓ}, the path which reaches t_1 via π_{t_1}, goes to t_ℓ, and then continues along π_{t_ℓ} from t_ℓ is also fair in M^ρ.

In $\widetilde{T_\psi}$, we have an edge from the aspect return state to every state whose labels match t_1; in particular, we must have an edge to the state corresponding to t_1 on the fair path matching our continuation from t_1 constructed above. We can continue the match π'_T for π'_M along this path.

If this continuation of base machine states is infinite, then the matching path in the tableau must be fair, since we are following the match of a fair path in M^ρ. If we never reach an infinite sequence of base states, but always reach another advice, then there must be some advice states which are visited infinitely many times, and again the path in the tableau is fair.

Therefore, for every fair path π'_M in $\widetilde{M}$ we have a corresponding fair path π'_T in $\widetilde{T_\psi}$. This correspondence completes the proof that $\widetilde{M} \models \phi$. □

4.1 Example

By way of example, suppose we have an aspect with base system assumption $\psi = \mathsf{A}\,\mathsf{G}\,((\neg a \wedge b) \rightarrow \mathsf{F}\,a)$ — that is, any state satisfying $\neg a \wedge b$ is eventually followed by a state satisfying a. We would like to prove that the application of our aspect to any base system satisfying ψ will give an augmented system satisfying result $\phi = \mathsf{A}\,\mathsf{G}\,((a \wedge b) \rightarrow \mathsf{X}\,\mathsf{F}\,a)$ — that is, any state satisfying $a \wedge b$ will eventually be followed by a later state satisfying a. While this example may not have clear correlation to a code-level problem, it serves to illuminate the capabilities of our technique.

Figure 3 shows the reachable portion of the tableau for the assumption ψ. In the diagram, shaded states are those contained in the only fairness set. The notation $\mathsf{X}g$, not actually

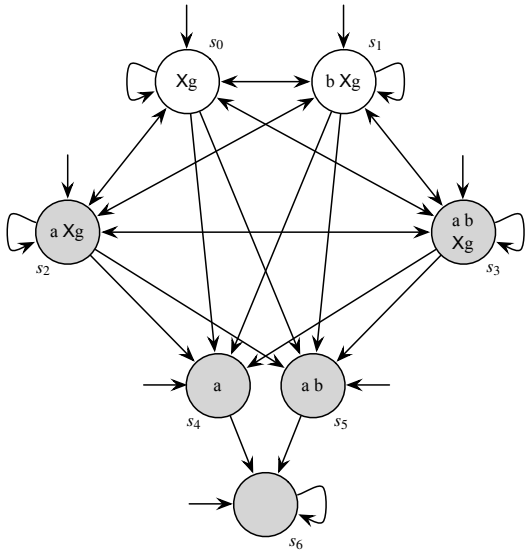


Figure 3: The reachable portion of tableau T_ψ for $\psi = \mathsf{A}\,\mathsf{G}\,((\neg a \wedge b) \rightarrow \mathsf{F}\,a)$.

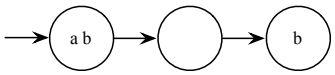


Figure 4: A simple aspect machine A.

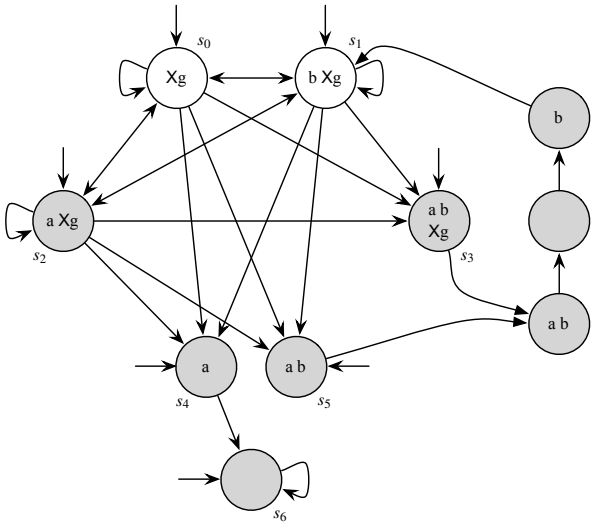


Figure 5: Augmented tableau $\widetilde{T_\psi}$, satisfying $\phi = \mathsf{A}\,\mathsf{G}\,((a \wedge b) \rightarrow \mathsf{X}\,\mathsf{F}\,a)$.

part of the state label, designates states in the tableau which satisfy $\mathsf{X}g$ for subformula $g = \mathsf{F}\,a$. For the example pointcut descriptor $\rho = (a \wedge b)$, this tableau machine is also pointcut-ready for ρ (since ρ references only the current state), simply by adding *pointcut* to the labels of s_3 and s_5.

Figure 4 shows the state machine A for the advice of our aspect. This advice will be applied at the states matched by ρ, and Figure 5 gives the weaving of A with T_ψ according to ρ. Model checking this augmented tableau will indeed establish that it satisfies the desired property ϕ. This result follows neither from the aspect nor base machine behavior directly, but from their combined behavior mediated by ρ. And since $\widetilde{T_\psi} \models \phi$, any $M \models \psi$ will yield $\widetilde{M} \models \phi$.

Reasoning intuitively about A and ρ without examining the tableau supports this conclusion: the advice is invoked at all states of such an M that match $(a \wedge b)$, the advice always leads to a state satisfying $(\neg a \wedge b)$, and ψ guarantees that from such a state we will always reach a state satisfying a, which is exactly the assertion of ϕ.

Figure 6 depicts a particular base machine M satisfying ψ, as could be easily verified by model checking. Again, the shaded states are those in the only fairness set. Although this M is small, it does contain atomic proposition c not 'visible' to the aspect, and it has a disconnected structure very much unlike the tableau. From Figure 7, one sees it is indeed the case that the augmented machine $\widetilde{M}$ satisfies ϕ — but there is no need to prove this directly by model checking. This holds true even though the addition of the aspect has made a number of invasive changes to M: state s_1 is no longer reachable, because its only incoming edge has been replaced by an advice edge; a new loop through s_0 has been added, when in M there was no path visiting s_0 more than once; there is a new path connecting the previously separated left-hand component to the right-hand; and so forth. In more realistic examples, the difference in size between the augmented tableau (involving only ψ, ρ, and A) and a concrete augmented system with advice over a full base machine would be substantial.

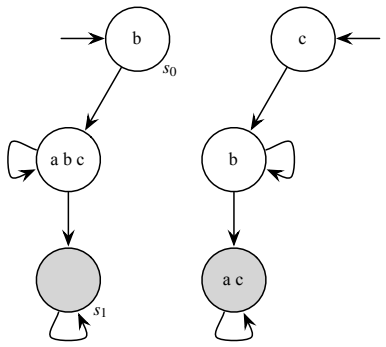


Figure 6: One particular base machine $M \models \psi$.

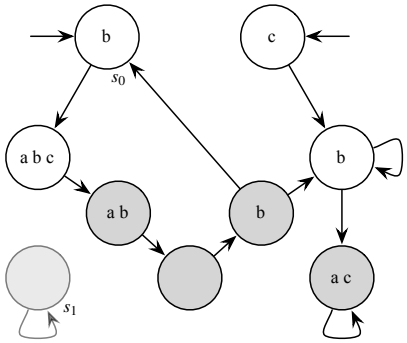


Figure 7: Augmenting M with A according to ρ gives result $\widetilde{M} \models \phi$.

5. RELATED WORK

The first work to separately model check the aspect state machine segments that correspond to advice is [9], where the verification is modular in the sense that base and aspect machines are considered separately. The verification method also allows for joinpoints within advice to be matched by a pointcut and themselves advised. However, the treatment there is for a particular aspect woven directly to a particular base program. Additionally, it shows only how to extend properties which hold for that base program, proving that the augmented program satisfies them as well (properties are specified in branching-time logic CTL). A key assumption of their method is that after the aspect machine completes, the continuation is always to the state following the joinpoint in the original base program. This requirement is much stronger than the assumption used here of a weakly invasive aspect.

In [8], model checking tasks are automatically generated for the augmented system that results from each weaving of an aspect. That approach has the disadvantage of having to treat the augmented system, but at least the needed annotations and set-up need only be prepared once. That work takes advantage of the Bandera [5] system that generates input to model checking tools directly from Java code, and can be extended to, for example, the aspect-oriented AspectJ language. Bandera and other systems like Java Pathfinder [6] that generate state machine representations from code can be used to connect common high-level aspect languages to the state machines used in the results here.

In [7] a semantic model based on state machines is given, and the treatment of code-level aspects and joinpoints defined in terms of transitions, as in AspectJ, is described. In particular, the variations needed to express in a state machine weaving the meaning of *before*, *after*, and *around* with *proceed* are outlined, although work remains to fully capture the intended semantics.

In [4] and [11], among others, an assume-guarantee structure for aspect specification is suggested, similar to the specifications here, but model checking is not used.

6. CONCLUSION

By reusing the notion of a tableau which contains all possible behaviors that satisfy a particular formula, we can achieve a modular verification for aspects by augmenting the tableau with the advice according to a pointcut descriptor and ex-

amining the result. In order to do so we must restrict our view to aspects which are weakly invasive and always return to states which were reachable in the original base system, and we take a liberal view of fairness in which any computation that infinitely often visits an aspect state is considered fair.

A number of directions for future work present themselves quite clearly. While the current technique only addresses a single aspect and pointcut descriptor, in principle it can be extended to work for multiple aspects, given proper definitions of the weaving mechanics. Further development of how weaving is formulated will also allow treatment of cases where advice introduction changes the set of joinpoints. Furthermore, the entire discussion here is given in terms of states and state machines, while, as noted earlier, the usual basic vocabulary of aspect-oriented programming talks about events. The language-level aspect terminology and problems of real object systems still must be fully expressed in the state-based model checking used here. Nevertheless, the generic method in this paper allows us for the first time to model check aspects independently of a concrete base program, and is a significant step toward the truly modular verification of aspects.

7. REFERENCES

[1] E. Abraham, F. de Boer, W.-P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded java. *Theoretical Computer Science*, 331(2-3):251–290, 2005.

[2] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Y. Vardi. Regular vacuity. In D. Borrione and W. Paul, editors, *Proc. of Correct Hardware Design and Verification Methods, CHARME'05*, volume 3725 of *LNCS*, pages 191–206. Springer, 2005.

[3] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[4] B. Devereux. Compositional reasoning about aspects using alternating-time logic. In *Proc. of Foundations of Aspect Languages Workshop (FOAL03)*, 2003.

[5] J. Hatcliff and M. Dwyer. Using the Bandera Tool Set to model-check properties of concurrent Java software. In K. G. Larsen and M. Nielsen, editors, *Proc. 12th Int. Conf. on Concurrency Theory, CONCUR'01*, volume 2154 of *LNCS*, pages 39–58. Springer-Verlag, 2001.

[6] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), Apr 2000.

[7] S. Katz. Aspect categories and classes of temporal properties. In *Transactions on Aspect Oriented Software Development, Volume 1, LNCS 3880*, pages 106–134, 2006.

[8] S. Katz and M. Sihman. Aspect validation using model checking. In *Proc. of International Symposium on Verification*, LNCS 2772, pages 389–411, 2003.

[9] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *Proc. SIGSOFT Conference on Foundations of Software Engineering, FSE'04*, pages 137–146. ACM, 2004.

[10] NuSMV. `http://nusmv.irst.itc.it/`.

[11] H. Sipma. A formal model for cross-cutting modular transition systems. In *Proc. of Foundations of Aspect Languages Workshop (FOAL03)*, 2003.

Temporal Aspects as Security Automata

Peter Hui
CTI, DePaul University
Chicago, IL, USA

James Riely*
CTI, DePaul University
Chicago, IL, USA

Abstract

Aspect-oriented programming (AOP) has been touted as a promising paradigm for managing complex software-security concerns. Roughly, AOP allows the security-sensitive events in a system to be specified separately from core functionality. The events of interest are specified in a pointcut. *When a pointcut triggers, control is redirected to* advice*, which intercepts the event, potentially redirecting it to an error handler.*

Many interesting security properties are history-dependent; however, currently deployed pointcut languages cannot express history-sensitivity (mechanisms like `cflow` *in AspectJ capture only the current call stack.) We present a language of pointcuts with past-time temporal operators and discuss their implementation using a variant of security automata. The main result is a proof that the implementation is correct.*

Refining our earlier work ([6]), we define a minimal language of events and aspects in which "everything is an aspect". The minimalist approach serves to clarify the issues and may be of independent interest.

1. Introduction

Aspect-oriented programming (AOP) ([12]) is a relatively new programming paradigm designed to address concerns that cut across encapsulation boundaries of traditional approaches. In this model, the programmer defines *aspects*, each consisting of an *advice body* – a block of code – and a *pointcut*, which states when the code is to be executed. Current implementations allow for the user to define pointcuts which trigger off of a specified atomic event, but facilities for triggering of a program's history is typically limited to the current call stack (as in AspectJ's `cflow`).

AOP has some potential for specifying and enforcing security policies. However, many such policies are both *history-sensitive* and *dynamic* (likely to change at runtime). In this paper, we define a syntax and operational semantics for *temporal aspects*, which allow for pointcuts to be defined temporally— that is, in terms of events which have happened in the past. For instance, we would like to be able to declare advice which triggers when some function f is called, but only if a function g has been called at some point in the program's history. AspectJ's `cflow` can only capture the case where g lies in the call stack at the time when f is invoked. An obvious solution is to record every single event during the course of the program's execution. Such an implementation is clearly impractical for long-lived programs. In this vein, we present an equivalent, automaton-based semantics, to be used as a model for implementation, which records only relevant events. The automaton state provides an abstraction of the history, and our main result demonstrates that this abstract view faithfully implements the original semantics.

We use a variant of Schneider's security automata [16]. A *security automaton* enforces a security policy by monitoring the execution of a target system, and intercepting instructions which would otherwise violate the specified policy. For instance, a user may specify that subsequent to a *FileRead* operation, the user is forbidden from executing a *Send* operation. The corresponding automaton would monitor the target system, watching for instances of *FileRead*. If one was seen, the automaton would then monitor the system for an attempted *Send*, and if such an attempt were made, it would intercept the call and presumably execute some error handling code instead.

Security automata have been widely investigated as a means of implementing security policies. In [21], Walker uses security automata to encode security policies to be enforced in automatically generated code. In [20], Erlingsson and Schneider use security automata to implement software fault isolation security policies, which prevent memory accesses outside of the allowable address space. In that work, they discuss techniques used to merge security automata directly into binary code at the x86 assembler and Java Virtual Machine Language (JVML) level. In [4], Barker and Stuckey investigate role based and temporal role based access control policies, implemented using constraint logic specifications. In [18], Thiemann incorporates security au-

* Research supported in part by NSF CAREER 0347542

tomata into an interpreter for a simply typed call-by-value lambda calculus, which he then translates to an equivalent two-level lambda calculus, upon which type specialization removes all run-time operations involving security state. The limitations of stack-based security policies are explored in [11]; history-based solutions are presented in [1]. Our work can be used as an alternative implementation technique for the ideas in the later paper.

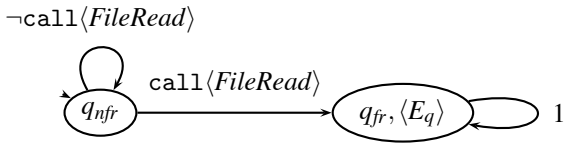
Several recent projects have studied history sensitivity in aspect languages. Douence, et. al. [10, 9] describe *event-based AOP*, in which advice is defined in-line with the event sequences that trigger it; the semantics is given in terms of weaving. Walker and Viggers [23] use a context-free grammar of *tracecuts*. Allan et. al. [2] extend this approach to *tracematches*, providing a novel technique to accommodate variable bindings, while restricting attention to regular properties. Stolz and Bodden [17] describe a technique for instrumenting Java bytecodes with LTL formula, using aspects to implement transitions in the underlying alternating automata. Bockish, et. al. [5] present a method for recording program history using a prolog database and using this to fire advice. De Fraine, et. al. [8] study dynamic weaving as a method for implementing stateful advice.

We contribute to this body of work by providing a foundational language for expressing dynamically loaded advice in a temporal framework, allowing us to define a full source-language semantics and to prove the correctness of its implementation. The situation is complicated by two facts: First, a pointcut may cause an event to be intercepted *before* it occurs; this is typical of security policies that specify sequences which must be aborted, rather than those which are allowed. Second, new advice may arrive at runtime, dynamically modifying existing policies. In both cases a key difficulty is getting the semantics of the source language "right". Refining our previous work [6], we adopt a minimalist approach which lays bare the essence of the problem without having to deal with the overhead of object-oriented details. Other work on the semantic foundations of AOP includes [22, 3, 14, 24, 19, 15, 7, 13].

We proceed as follows: in Section 2, we provide a motivating example. In Section 3, we define Polyadic μABC, a minimal aspect-based calculus defining roles, advice, and non-temporal advised messages. In Section 4, we augment Polyadic μABC to include temporal pointcuts, specified using a subset of the regular expressions, namely those of the form $\phi\alpha$, where ϕ is a regular expression abstracting the program's history, and α is the atomic event (i.e., `call`) which triggers the advice. In Section 5, we define an equivalent, automaton-based implementation semantics. In Section 6, we prove equivalence of the two semantics by providing a translation of a configuration in the history-based semantics to an equivalent configuration in the automaton-based semantics, and showing that the translation is preserved across evaluation. Future work is discussed in Section 7.

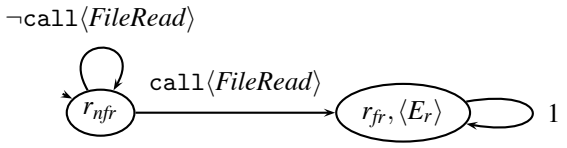
2. Motivation

The following automaton implements a security policy which prohibits *Send* operations after a *FileRead* has been executed [16].



Our presentation differs slightly from that of [16] in that we attach an error handling aspect E_q to state q_{fr}. Its task is to watch for and intercept an attempted `call`$\langle$*Send*$\rangle$. We attach an aspect to the state instead of transitioning into a new "error" state because transitions represent *committed* function calls— our intent is to block the `call`$\langle$*Send*$\rangle$, whereas transitioning into a new state would indicate that we have indeed committed it.

Now, say that at some point during the program's execution, the user executes a `call`$\langle$*FileRead*$\rangle$, and as a result, the automaton is in state q_{fr}. Furthermore, suppose that at this point, a new quarantine policy is added to the system, which prohibits a user from logging into some system A after a *FileRead* is called. One possible automaton for this policy is shown below:



Here, E_r is the error handling advice which monitors for a `call`$\langle Login, A\rangle$ after seeing a `call`$\langle$*FileRead*$\rangle$. It may seem that the automaton resulting from adding the quarantine policy to the "read-send" policy is simply the product of the above two automata. However, in general this is unimplementable without storing the entire history. New policies may reference arbitrary events in the system history, whereas a given security automaton is committed to a particular abstract view of that history. Our solution is simple: we interpret policies as holding only from the point at which they are implemented.

Consider, in our example, what happens if the next operation is a `call`$\langle Login,\ A\rangle$. If we "play back" the program history (`call`$\langle$*FileRead*$\rangle$, `call`$\langle Login,\ A\rangle$) on the product automaton, advice $\langle E_r \rangle$ will fire, which is incorrect according to our interpretation — the quarantine policy was implemented *after* the `call`$\langle$*FileRead*$\rangle$. Thus, when constructing the combined automaton, we must be careful to take into account the history of the execution.

As this example shows, while the implementation of such security policies using finite automata is straightforward, a subtlety arises when new policies may be added at runtime; one must be careful in defining which program traces are in fact captured by a new policy added to a running system. To clarify the issues, we define the semantics of dynamic temporal aspects over complete execution histories. We subsequently provide an equivalent, automaton-based implementation semantics which records only an abstraction of the execution history. Finally, we define a translation between states in the two semantics and prove that this translation commutes with evaluation.

3. Polyadic μABC

NOTATION. For any metavariable X, we write $\bar{X}$ for an ordered sequence of X's.

We define a polyadic variant of μABC, introduced in [6]. The earlier paper followed the style of object-oriented languages; each message "$p \to q : \ell$" had a source p, a destination q, and a name ℓ. Such a message is *triadic* in that its meaning depends on a triple of names, or *roles*. Here we generalize triadic messages to polyadic *events*, $\langle p_1, \ldots, p_n \rangle$ (equivalently $\langle \bar{p} \rangle$), with triadic messages as a special case "$\langle p, q, \ell \rangle$".

For simplicity, in this paper, we look at a single-threaded variant. At each moment in runtime, there is a single event $\langle \bar{p} \rangle$ under consideration. Execution is determined by *advice* that triggers on the event. At any given moment, the current event is decorated with a vector of advice $\bar{a}$, which is waiting to process the event. Thus $a_1, \ldots, a_n \langle \bar{p} \rangle$ indicates that advices named a_i are waiting to process event $\bar{p}$. We say that a_i *advises* $\bar{p}$, and that $\bar{a}\langle \bar{p} \rangle$ is an *advised* event. For consistency with the precedence of declarations, we read the advice list from right to left; thus a_n is the first advice to process the event.

The special advice call initiates advice lookup. When $\mathsf{call}\langle \bar{p} \rangle$ executes, all the advice triggering on $\langle \bar{p} \rangle$ is listed, resulting in a new execution state: $\bar{a}\langle \bar{p} \rangle$. To determine whether an advice is triggered, we use the pointcut α. Pointcuts may be defined to trigger on an exact role, or a set of roles. We facilitate the specification of such sets using a role preorder, with maximal element top.

An advice body $\mathsf{adv}\, a[\alpha] = u(\bar{x})\, N$ is parameterized both on the event $\bar{x}$ and the remaining advice u. Following the terminology of *around* advice in AspectJ, we refer to u as the *proceed* variable.

3.1. Syntax and Evaluation

We give the syntax and evaluation semantics of the language parametrically with respect to pointcuts α and pointcut satisfaction $\bar{D} \vdash \langle \bar{p} \rangle\ \mathsf{sat}\ \alpha$, described in the next subsection. Note that terms have the form $\bar{D}\,;\, \bar{a}\langle \bar{p} \rangle$; ie, a term is a list of declarations followed by a single advised event. We refer to $\bar{p}$ as the *current event*, $\bar{a}$ as the *current advice list*, and a_n as the *current advice* ($\bar{a} = a_1, \ldots, a_n$).

TERM SYNTAX

a–e, u–w	Advice Names; call, commit reserved
f–t, x–z	Role Names; top reserved
$D, E ::=$	Declarations
$\quad \mathsf{role}\ p < q$	Role; $dn(\mathsf{role}\ p) = p$
$\quad \mathsf{adv}\ a[\alpha] = u(\bar{x})\ N$	Advice; $dn(\mathsf{adv}\ a) = a$; u and $\bar{x}$ bound in N
$L, M, N ::=$	Terms
$\quad D\,;\, M$	Declaration; $dn(D)$ bound in M
$\quad \bar{a}\langle \bar{p} \rangle$	Message

NOTATION. We write $dn(D)$ for the declared name of D. Reserved names may not be declared. We identify syntax up to renaming of bound names. For any syntactic category with typical element $\mathcal{E}$, we write $fn(\mathcal{E})$ for the set of free names occurring in $\mathcal{E}$. We write $\mathcal{E}\{^{a}/_{x}\}$ for the capture avoiding substitution of a for x in $\mathcal{E}$. We write $\mathcal{E}\{^{\bar{a}}/_{\bar{x}}\}$ for $\mathcal{E}\{^{a_1}/_{x_1}, \ldots, ^{a_n}/_{x_n}\}$; note that $\mathcal{E}\{^{\bar{a}}/_{\bar{x}}\}$ is defined only if $\bar{x}$ and $\bar{a}$ have the same length.

CONVENTION. To improve readability, we use the following discipline for names:

- a–e are advice names (including the reserved names call and commit);
- u–w are advice names that are bound in the body of an advice declaration;
- f–t are role names (including the reserved name top);
- x–z are role names that are bound in the body of an advice declaration;
- _ is a reserved name used to bind a name that is not of interest — that is, does not occur free in any subterm.

We drop syntactic elements that are not of interest. Consider the declaration "$\mathsf{adv}\ a[\alpha] = u(\bar{x})\ N$"; we may elide the name "$\mathsf{adv}[\alpha] = u(\bar{x})\ N$", or the pointcut "$\mathsf{adv}\ a = u(\bar{x})\ N$", or the body "$\mathsf{adv}\ a[\alpha]$", or both the pointcut and the body "$\mathsf{adv}\ a$".

Evaluation is defined using configurations which consist of a vector of declarations and a term. By EVAL-DEC, declarations are recorded in the configuration whenever they are encountered in a term. By EVAL-CALL, if an event $\langle \bar{p} \rangle$ is being processed with first advice call, then the advice list $\bar{a}$ is calculated, consisting of the advice names a_i such that the pointcut declared with a_i is satisfied by $\langle \bar{p} \rangle$. By EVAL-ADV, if an event $\langle \bar{p} \rangle$ is being processed with first advice a, then the body of a is executed; the advice body is parameterized by both the event $\langle \bar{p} \rangle$ and remaining advice $\bar{b}$. (Note that the

syntax requires that "$\bar{b}, a\langle\bar{p}\rangle$" be parsed as "$(\bar{b}, a)\langle\bar{p}\rangle$". Further note that the substitution $\bar{b}/u$ results in a well-formed term because free advice names can only appear in the context of a sequence.)

EVALUATION $(\bar{D} \rhd M \to \bar{E} \rhd N)$

$$\frac{[\bar{a}] = \left[a \;\middle|\; \begin{array}{l}\bar{D} \ni \mathsf{adv}\ a[\alpha] \\ \bar{D} \vdash \langle\bar{p}\rangle\ \mathsf{sat}\ \alpha\end{array}\right]}{\bar{D} \rhd \bar{b}, \mathsf{call}\langle\bar{p}\rangle \to \bar{D} \rhd \bar{b}, \bar{a}\langle\bar{p}\rangle}\ (\text{EVAL-CALL}) \qquad \frac{}{\bar{D} \rhd E; M \to \bar{D}, E \rhd M}\ (\text{EVAL-DEC})$$

$$\frac{\bar{D} \ni \mathsf{adv}\ a = u(\bar{x})\ N}{\bar{D} \rhd \bar{b}, a\langle\bar{p}\rangle \to \bar{D} \rhd N\{\bar{b}/u, \bar{p}/x\}}\ (\text{EVAL-ADV})$$

3.2. Atomic Event Pointcuts

We now consider a simple boolean logic over events. We allow event sets to be specified using role patterns which include subroles and "varargs", ie, optional roles.[1]

POINTCUT SYNTAX

$P, Q ::=$	Role Pattern
$\quad p$	Exact Role
$\quad +p$	Sub Role
$\alpha, \beta ::=$	Atomic Event Pointcut
$\quad \langle\bar{P}\rangle$	Call Event
$\quad \langle\bar{P}, *\rangle$	Call Event, varargs
$\quad \alpha \vee \beta$	Disjunction
$\quad \neg\alpha$	Negation
$\sigma, \rho ::= \langle\bar{p}\rangle$	Atomic Event

Define 1 as $\langle *\rangle$; define 0 as $\neg 1$; and define $\alpha \wedge \beta$ as $\neg(\neg\alpha \vee \neg\beta)$. We write $\bar{D} \vdash r \leqslant p$ for the obvious preorder generated from the role declaration order. From this, we derive the following definition of pointcut satisfaction; the obvious rules for conjunction and disjunction are elided.

ATOMIC POINTCUT SATISFACTION $(\bar{D} \vdash \sigma\ \mathsf{sat}\ \alpha)$

$$\frac{}{\bar{D} \vdash \langle\bar{p}\rangle\ \mathsf{sat}\ \langle *\rangle}\ (\text{SAT-CALL-ANY}) \qquad \frac{}{\bar{D} \vdash \langle\rangle\ \mathsf{sat}\ \langle\rangle}\ (\text{SAT-CALL-EMPTY})$$

$$\frac{\bar{D} \vdash \langle\bar{q}\rangle\ \mathsf{sat}\ \langle\bar{Q}\rangle}{\bar{D} \vdash \langle r, \bar{q}\rangle\ \mathsf{sat}\ \langle r, \bar{Q}\rangle}\ (\text{SAT-CALL-EXACT}) \qquad \frac{\bar{D} \vdash \langle\bar{q}\rangle\ \mathsf{sat}\ \langle\bar{Q}\rangle \quad \bar{D} \vdash r \leqslant p}{\bar{D} \vdash \langle r, \bar{q}\rangle\ \mathsf{sat}\ \langle +p, \bar{Q}\rangle}\ (\text{SAT-CALL-SUB})$$

4. Temporal Pointcuts

We extend μABC with temporal pointcuts. To do this, we modify the language of advice to include a temporal formula ϕ in addition to the atomic formula α. Intuitively, the pointcut fires when ϕ matches the past and α matches the current event.

In an aspect language, the ontology of events is complicated by the fact that events can be diverted; that is, an event can trigger advice that intercepts the event *before* it occurs, potentially causing the event to abort. This is particularly common in applications to security, where pointcuts often specify dangerous event sequences that interrupt normal processing. To indicate that an event is to be recorded in the history, we include the special advice commit.

Thus when the past is considered in firing a pointcut, we require that advice specify both the past ϕ and the potential future α. The past is specified as a regular expression over atomic event pointcuts; the potential future is specified as an atomic event pointcut.

SYNTAX

$D, E ::= \cdots$	Declarations
$\quad \mathsf{adv}\ a[\phi\alpha] = u(\bar{x})\ N$	Declare Advice
$\phi, \psi, \chi ::=$	Temporal Pointcuts
$\quad \alpha$	Atomic Event Pointcut
$\quad \varepsilon$	Empty Sequence
$\quad \phi\psi$	Sequence
$\quad \phi^*$	Kleene Star
$\quad \phi + \psi$	Disjunction
$\sigma, \rho ::= \langle\bar{p}\rangle$	Atomic Events

The semantics of temporal formulas $\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi$ is defined in the standard way (recalled in Appendix A) over strings of events, building on the semantics of atomic events ($\bar{D} \vdash \sigma\ \mathsf{sat}\ \alpha$). Note that the regular expression $\emptyset$ is represented here as the atomic event pointcut 0. We define the language of the formula as follows: $\mathcal{L}_H(\bar{D}, \phi) = \{\bar{\sigma} \mid \bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi\}$.

We now give the evaluation semantics for the language with temporal advice. We augment the semantics to record an execution history. We write $|\bar{\sigma}|$ for the length of string $\bar{\sigma}$. We define $\alpha^n \triangleq \alpha\alpha^{n-1}$, where $\alpha^0 \triangleq \varepsilon$. We write "$\mathsf{adv}\ a[\alpha] = u(\bar{x})\ N$" as shorthand for "$\mathsf{adv}\ a[1^*\alpha] = u(\bar{x})\ N$".

EVALUATION $(\bar{\sigma}; \bar{D} \rhd M \to \bar{\rho}; \bar{E} \rhd N)$

$$\frac{}{\bar{\sigma}; \bar{D} \rhd \mathsf{role}\ p < q; M \to \bar{\sigma}; \bar{D}, \mathsf{role}\ p < q \rhd M}\ (\text{EVAL-DEC-ROLE}) \qquad \frac{}{\bar{\sigma}; \bar{D} \rhd \mathsf{adv}\ a[\phi\alpha] = u(\bar{x})\ N; M \to \bar{\sigma}; \bar{D}, \mathsf{adv}\ a[1^{|\bar{\sigma}|}\phi\alpha] = u(\bar{x})\ N \rhd M}\ (\text{EVAL-DEC-ADV})$$

$$\frac{}{\bar{\sigma}; \bar{D} \rhd \bar{b}, \mathtt{commit}\langle\bar{p}\rangle \to \bar{\sigma}, \langle\bar{p}\rangle; \bar{D} \rhd \bar{b}}\ (\text{EVAL-COMMIT})$$

$$\frac{[\bar{a}] = \left[a \;\middle|\; \bar{D} \ni \mathsf{adv}\ a[\phi\alpha] \text{ and } \bar{D} \vdash \bar{\sigma}, \langle\bar{p}\rangle\ \mathsf{sat}\ \phi\alpha\right]}{\bar{\sigma}; \bar{D} \rhd \bar{b}, \mathsf{call}\langle\bar{p}\rangle \to \bar{\sigma}; \bar{D} \rhd \bar{b}, \bar{a}\langle\bar{p}\rangle}\ (\text{EVAL-CALL})$$

$$\frac{\bar{D} \ni \mathsf{adv}\ a = u(\bar{x})\ N}{\bar{\sigma}; \bar{D} \rhd \bar{b}, a\langle\bar{p}\rangle \to \bar{\sigma}; \bar{D} \rhd N\{\bar{b}/u, \bar{p}/x\}}\ (\text{EVAL-ADV})$$

1 In the full version we also allow vararg parameters in advice declarations, ie $\mathsf{adv}\ a[\alpha] = u(\bar{x}, *)\ N$.

EVAL-COMMIT causes an event to be recorded in the history. The original EVAL-DEC is split into different cases for roles and advice. EVAL-DEC-ROLE, EVAL-CALL, and EVAL-ADV are largely unchanged from the non-temporal semantics. Note only that in EVAL-CALL the history is used, along with the current event, to determine whether an advice fires.

Of particular note is the rule EVAL-DEC-ADV, which takes a newly declared advice, and prepends a string of 1s to the temporal pointcut prior to adding it to the list of declarations. The purpose of doing so is to ensure that the advice only triggers on the event α *from the point of declaration onwards*, as opposed to some event that has already occurred in the past.

5. Automaton

In this section, we define an equivalent automaton-based semantics.

Our automata are constructed from regular expressions of the form $\phi\alpha$, corresponding to an advice declaration $\mathsf{adv}\ a[\phi\alpha]$, where ϕ is a regular expression abstracting the relevant events in the program history, and α is the triggering atomic event. For each advice $\mathsf{adv}\ a[\phi\alpha]$, we construct the automaton for ϕ. From the point of declaration onwards, the automaton monitors program execution. If the automaton ever enters its final state, this indicates that an attempt to execute α should be intercepted, and the advice body executed instead. To implement this, we attach the advice name to each final state for the automaton. For this reason, we refer to final states as advice states:

ADVICE STATES $(\phi\checkmark)$

$$\frac{}{\varepsilon\checkmark} \qquad \frac{\phi\checkmark \quad \psi\checkmark}{\phi\psi\checkmark} \qquad \frac{\phi\checkmark}{\phi+\psi\checkmark} \qquad \frac{\psi\checkmark}{\phi+\psi\checkmark} \qquad \frac{}{\phi^*\checkmark}$$

The states are sets of temporal pointcut formulas ϕ, the transition alphabet ranges over the atomic event pointcuts α, and the transitions of the automaton are defined by the standard transition relation:

TRANSITION RELATION $(\phi \xrightarrow{\alpha} \psi)$

$$\frac{}{\alpha \xrightarrow{\alpha} \varepsilon} \qquad \frac{\phi \xrightarrow{\alpha} \phi'}{\phi\psi \xrightarrow{\alpha} \phi'\psi} \qquad \frac{\phi\checkmark \quad \psi \xrightarrow{\alpha} \psi'}{\phi\psi \xrightarrow{\alpha} \psi'} \qquad \frac{\phi \xrightarrow{\alpha} \phi'}{\phi^* \xrightarrow{\alpha} \phi'\phi^*}$$

$$\frac{\phi \xrightarrow{\alpha} \phi'}{\phi+\psi \xrightarrow{\alpha} \phi'} \qquad \frac{\psi \xrightarrow{\alpha} \psi'}{\phi+\psi \xrightarrow{\alpha} \psi'}$$

Transitions between states are taken on `commit`s.

We write $\xRightarrow{\phi}$ for the reflexive transitive closure of $\xrightarrow{\alpha}$. Next, we formally state how to derive an automaton from an advice $\mathsf{adv}\ a[\phi\alpha]$:

NOTATION. For any advice $\mathsf{adv}\ a[\phi\alpha]$, let the automaton $\iota(\phi,a)$ *induced by* a be the security automaton with states and transitions as defined by the transition relation given above, with start state ϕ, and advice a associated with each advice state.

We represent our automata as (state, advice set) pairs:

AUTOMATON SYNTAX

$\Phi,\Psi ::= \phi \mid \phi,\Phi$	State
$\mathcal{A} ::= \langle\Phi,\bar{a}\rangle \mid \langle\Phi,\bar{a}\rangle,\mathcal{A}$	Automaton

For instance, $\mathcal{A}_R$ and $\mathcal{A}_Q$ from Section 2 are represented as $\langle\phi_0,\emptyset\rangle,\langle\phi_1,\{E_1\}\rangle$ and $\langle\psi_0,\emptyset\rangle,\langle\psi_1,\{E_2\}\rangle$, respectively, with $\phi_0 = \psi_0 \triangleq [\neg\texttt{call}\langle \mathit{FileRead}\rangle]^*\texttt{call}\langle \mathit{FileRead}\rangle 1^*$, and $\phi_1 = \psi_1 \triangleq 1^*$. The product automaton $\mathcal{A}_R \times \mathcal{A}_Q$ would be represented as

$$(\langle(\phi_0,\psi_0),\emptyset\rangle,(\langle\phi_0,\psi_1\rangle,\{E_2\}),$$
$$(\langle\phi_1,\psi_0\rangle,\{E_1\}),(\langle\phi_1,\psi_1\rangle,\{E_1,E_2\})$$

There is no need to explicitly encode the transition relation. For instance, in the product automaton just presented, we know from the definition of the transition relation that $\langle\phi_1,\psi_0\rangle \xrightarrow{\texttt{call}\langle \mathit{FileRead}\rangle} \langle\phi_1,\psi_1\rangle$. To make the presentation more readable, we elide advice when a state has none associated with it. That is, we write the state "$\langle\phi\rangle,\emptyset$" simply as ϕ.

We can modulate the transition relation from atomic event pointcuts to atomic events: define $\bar{D} \vdash \phi \xrightarrow{\sigma} \phi'$ if $\phi \xrightarrow{\alpha} \phi$' and $\bar{D} \vdash \sigma\ \mathsf{sat}\ \alpha$. Further we can lift the definition to automaton states: $\bar{D} \vdash \phi_1,\ldots,\phi_n \xRightarrow{\sigma} \psi_1,\ldots,\psi_n$ if $\bar{D} \vdash \phi_i \xrightarrow{\sigma} \psi_i$ for all i between 1 and n. Finally we lift the resulting relation ($\bar{D} \vdash \Phi \xRightarrow{\sigma} \Psi$) to event sequences: $D \vdash \Phi_0 \xrightarrow{\sigma_1,\ldots,\sigma_n} \Phi_n$ if $\bar{D} \vdash \Phi_{i-1} \xRightarrow{\sigma_i} \Phi_i$ for all i between 1 and n.

We define the product of two automata using the standard product construction, taking the set union of each component state's associated advice names:

DEFINITION 1. For any two automata A,B,

$$A \times B = \{\langle\Phi_A,\Phi_B;\bar{a},\bar{b}\rangle \mid \langle\Phi_A,\bar{a}\rangle \in A, \langle\Phi_B,\bar{b}\rangle \in B\}$$

Next, we show how to merge an advice $\mathsf{adv}\ a[\phi\alpha]$ with an existing automaton $\mathcal{A}$. Namely, we construct the automaton for the advice, and create the product automaton:

$$\nu(\mathcal{A},\phi,a) \triangleq \mathcal{A} \times \iota(\phi,a)$$

We now give the equivalent, automaton-based evaluation semantics to our language. Whereas previously we recorded the entire program history, we now instead maintain an automaton and state, which records only events of interest.

EVALUATION $(\mathcal{A};\Phi;\bar{D}\rhd M\to\mathcal{A}';\Psi;\bar{D}'\rhd M')$

(EVAL-DEC-ROLE and EVAL-ADV as before)

(EVAL-COMMIT)

$$\frac{\bar{D}\vdash\Phi\stackrel{\bar{p}}{\Longrightarrow}\Psi}{\mathcal{A};\Phi;\bar{D}\rhd\bar{b},\texttt{commit}\langle\bar{p}\rangle\to\mathcal{A};\Psi;\bar{D}\rhd\bar{b}}$$

(EVAL-DEC-ADV)

$$\frac{}{\begin{array}{l}\mathcal{A};\Phi;\bar{D}\rhd(\mathsf{adv}\ a[\phi\alpha]=u(\bar{x})\,N\,;M)\\ \to\nu(\mathcal{A},\phi,a);\langle\Phi,\phi\rangle;\bar{D},(\mathsf{adv}\ a[\alpha]=u(\bar{x})\,N)\rhd M\end{array}}$$

(EVAL-CALL)

$$\frac{[\bar{a}]=\left[a\;\middle|\;\begin{array}{l}\langle\Phi,(\bar{b},a,\bar{b}')\rangle\in\mathcal{A}\\ \bar{D}\ni\mathsf{adv}\ a[\alpha]\\ \bar{D}\vdash\langle\bar{p}\rangle\ \mathsf{sat}\ \alpha\end{array}\right]}{\mathcal{A};\Phi;\bar{D}\rhd\bar{b},\mathsf{call}\langle\bar{p}\rangle\to\mathcal{A};\Phi;\bar{D}\rhd\bar{b},\bar{a}\langle\bar{p}\rangle}$$

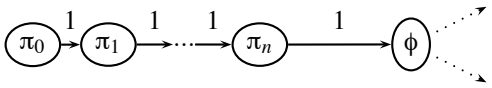
Operationally, EVAL-DEC-ROLE and EVAL-ADV act the same as in the history-based semantics. EVAL-DEC-ADV takes a new advice, merges it into the automaton, updates the current state, and adds the advice to the list of declarations. EVAL-CALL looks through the list of advices attached to the current state for one whose atomic pointcut matches the role vector $\bar{p}$ being called. If a matching advice is found, then the $\texttt{call}\langle\bar{p}\rangle$ is replaced with the advice body. EVAL-COMMIT simply updates the state of the automaton.

6. Equivalence

In this section, we demonstrate equivalence of the history-based semantics provided in Section 4 with the automaton-based semantics provided in Section 5 by providing a translation from a configuration in the former to an equivalent one in the latter. We conclude by showing that evaluation preserves the translation.

Intuitively, we translate a history-based configuration $\langle\bar{\sigma},\bar{D}\rangle$ to an automaton-based configuration $\langle\mathcal{A},\Phi,\bar{E}\rangle$ as follows: given a history $\bar{\sigma}$ and a set of declarations $\bar{D}$, we first construct an intermediate automaton $\mathcal{A}'$ using the aspect declarations in $\bar{D}$. We compute the state Φ by simulating the history $\bar{\sigma}$ on $\mathcal{A}'$. Finally, we convert the intermediate automaton $\mathcal{A}'$ to the final automaton $\mathcal{A}$ by removing intermediate states.

Recall the manner in which EVAL-DEC-ADV is defined in the history-based semantics: whenever an advice is declared, the current "timestamp" is explicitly noted in the form of a string of '1's prepended to the temporal pointcut. Thus if an advice $\mathsf{adv}\ a[\phi\alpha]$ is declared at time n, then in the history-based semantics, the pointcut is noted as $\mathsf{adv}\ a[1^n\phi\alpha]$, and the corresponding automaton in the automaton-based semantics will have a string of n "placeholder" states $\pi_1,\ldots,\pi_n$, where $\pi_i\xrightarrow{1}\pi_{i+1}$ for i between 1 and $n-1$, and $\pi_n\xrightarrow{1}\phi$, as shown below:



CONVENTION. In constructing an automaton for an advice $\mathsf{adv}\ a[\phi\alpha]$ declared at time n, we label the states used as placeholders for time 1 through n as $\pi_0,\ldots,\pi_n$, and we refer to these as π-states.

Strictly speaking, we must account for the fact that for an advice $\mathsf{adv}\ a[\phi\alpha]$, ϕ may in fact begin with a string of leading 1s. We can easily get around this by syntactically differentiating between those 1s implicitly inserted by EVAL-DEC-ADV as a timestamp, and those explicitly specified by the user. In the interest of simplifying the presentation, we choose not to do so here.

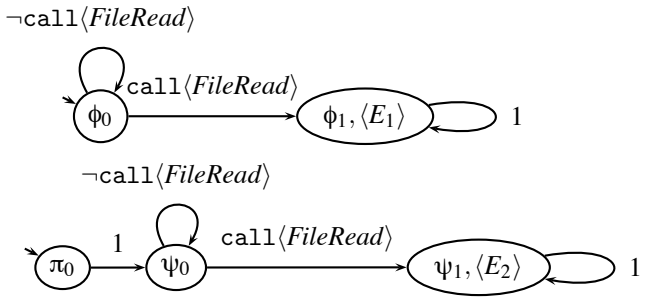
If a state $\Phi=\langle\phi_i,\psi_i,\ldots\chi_i\rangle$ is such that none of $\phi_i,\psi_i,\ldots\chi_i$ are π-states, we say that Φ is π-free. We will need to project the π-free states of an automaton, so we formally define this operation:

$$\mathcal{P}_{\not\pi}(A)=\{\langle\Phi,\bar{a}\rangle\in\mathcal{A}|\Phi\text{ contains no }\pi\text{ states}\}$$

LEMMA 2. *For two automata $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{P}_{\not\pi}(\mathcal{A}\times\mathcal{B})=\mathcal{P}_{\not\pi}(\mathcal{A})\times\mathcal{P}_{\not\pi}(\mathcal{B})$*

Proof. Immediate. □

To construct $\mathcal{A}'$, we take the product of the automata induced by each advice in $\bar{D}$. To construct Φ, we simulate the program history $\bar{\sigma}$ on $\mathcal{A}'$. For instance, in the example in Section 2, $\mathcal{A}'$ is the product of the following automata, with ϕ_0,ϕ_1,ψ_0, and ψ_1 defined as in Section 5:



Simulating the program history $(\texttt{call}\langle\mathit{FileRead}\rangle,\texttt{call}\langle\mathit{Login},A\rangle)$ on the product automaton places us in state $\langle\phi_1,\psi_0\rangle$, as expected.

We compute $\mathcal{A}$ by removing from $\mathcal{A}'$ any states containing a π state. In our example, this amounts to removing from the product automaton states $\langle(\phi_0,\pi_0)\rangle$ and $\langle(\phi_1,\pi_0),\{E_1\}\rangle$. The result is equivalent to the product automaton $\mathcal{A}_Q\times\mathcal{A}_R$, where $\mathcal{A}_Q$ and $\mathcal{A}_R$ are as in Section 2.

We now formalize the translation just discussed. That is, given a history, declaration pair $\langle \bar{\sigma}, \bar{D} \rangle$, we formally show how to construct the corresponding automaton, state, declaration triple $\langle \mathcal{A}, \Phi, \bar{E} \rangle$.

Our translation makes use of the following functions:

$$\mathcal{T}_{dec}(\bar{D}) = \{\mathsf{adv}\ a[\alpha] | \mathsf{adv}\ a[\phi\,\alpha] \in \bar{D}\}$$
$$\mathcal{T}_{state}(\bar{\sigma}, \bar{D}) = \Psi, \text{ where}$$
$$\bar{D} = \langle \mathsf{adv}\ \_[1^{i_1}\phi_1\,\alpha_1], \ldots, \mathsf{adv}\ \_[1^{i_n}\phi_n\,\alpha_n] \rangle \text{ and}$$
$$\langle 1^{i_1}\phi_1, ..., 1^{i_n}\phi_n \rangle \xRightarrow{\bar{\sigma}} \Psi$$

We are now in a position to define the function $\mathcal{T}$ which translates a history-based configuration $\langle \bar{\sigma}; \bar{D} \rangle$ to an automaton-based configuration $\langle \mathcal{A}; \Phi; \bar{E} \rangle$:

DEFINITION 3. $\mathcal{T}(\bar{\sigma}; \bar{D}) = \mathcal{A}; \Phi; \bar{E}$, where

$$\mathcal{A} = \mathcal{P}_{\not\pi}\left[\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\right] \qquad \Phi = \mathcal{T}_{state}(\bar{\sigma}, \bar{D})$$

$$\bar{E} = \mathcal{T}_{dec}(\bar{D})$$

We define the language of the formula as follows:

$$\mathcal{L}_A(\bar{D}, \phi) = \{\bar{\sigma} | \bar{D} \vdash \phi \xRightarrow{\bar{\sigma}} \phi',\ \phi'\checkmark,\ \text{and } \phi' \in \mathcal{T}_{state}(\bar{\sigma}, \bar{D})\}$$

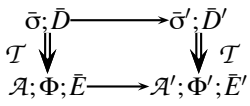
LEMMA 4. *For all $\bar{D}$ and ϕ, $\mathcal{L}_H(\bar{D}, \phi) = \mathcal{L}_A(\bar{D}, \phi)$.*

Proof. By induction on the structure of $\bar{\sigma}$. □

We conclude by showing that the translation is preserved by evaluation. That is, if

- $\bar{\sigma}; \bar{D} \rhd M \to \bar{\sigma}'; \bar{D}' \rhd M'$
- $\mathcal{T}(\bar{\sigma}, \bar{D}) = \mathcal{A}; \Phi; \bar{E}$,
- $\mathcal{A}; \Phi; \bar{E} \rhd M \to \mathcal{A}'; \Phi'; \bar{E}' \rhd M'$, and
- $\mathcal{T}(\bar{\sigma}', \bar{D}') = \mathcal{A}''; \Phi''; \bar{E}''$

then $\mathcal{A}' = \mathcal{A}'', \Phi' = \Phi''$, and $\bar{E}' = \bar{E}''$, as shown below:



PROPOSITION 5. *If $\bar{\sigma}; \bar{D} \rhd M \to \bar{\sigma}'; \bar{D}' \rhd M'$ and $\mathcal{T}(\bar{\sigma}, \bar{D}) = \mathcal{A}, \Phi, \bar{E}$, then $\mathcal{A}; \Phi; \bar{E} \rhd M \to \mathcal{A}'; \Phi'; \bar{E}' \rhd M'$, where $\mathcal{T}(\bar{\sigma}', \bar{D}') = \mathcal{A}', \Phi', \bar{E}'$.*

Proof. In each case, we first translate the left hand side into the automaton-based semantics. We then apply the evaluation rule (e.g., EVAL-DEC-ADV) to the automaton to obtain the next configuration $\langle \mathcal{A}', \Phi', \bar{E}' \rangle$. We then translate the right hand side into the automaton based semantics and show that the result equals $\langle \mathcal{A}', \Phi', \bar{E}' \rangle$.

In the cases of EVAL-DEC-ROLE and EVAL-ADV, this is trivial. In the case of EVAL-DEC-ADV, recall its evaluation rule in the history-based semantics:

$$\bar{\sigma}; \bar{D} \rhd \mathsf{adv}\ b[\psi\,\beta], M \to \bar{\sigma}; \bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\,\beta] \rhd M$$

The declarations $\bar{D}$ (equivalently $\bar{E}$) are trivially preserved by EVAL-DEC-ADV, which leaves us to show that the automaton $\mathcal{A}$ and the state Φ are preserved. Translating the left hand side yields $\mathcal{T}(\bar{\sigma}; \bar{D}) = \mathcal{A}; \Phi; \bar{E}$, where

$$\mathcal{A} \triangleq \mathcal{P}_{\not\pi}\left[\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\right] \quad \Phi \triangleq \mathcal{T}_{state}(\bar{D}) \quad \bar{E} \triangleq \mathcal{T}_{dec}(\bar{D})$$

By EVAL-DEC-ADV in the automaton semantics, $\mathcal{A}; \Phi; \bar{E} \rhd \mathsf{adv}\ b[\psi\,\beta], M \to \mathcal{A}'; \Phi'; \bar{E}' \rhd M$, where

$$\mathcal{A}' \triangleq \mathcal{P}_{\not\pi}\left[\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\right] \times \iota(\psi, b) \quad \Phi' \triangleq \langle \Phi, \psi \rangle$$
$$\bar{E}' \triangleq \mathcal{T}_{dec}(\bar{D}), b$$

Finally, we must show that $\mathcal{T}(\bar{\sigma}'; \bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\,\beta]) = \mathcal{A}'; \Phi'; \bar{E}'$. By definition, $\mathcal{T}(\bar{\sigma}'; \bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\,\beta]) = \mathcal{A}''; \Phi''; \bar{E}''$, where

$$\mathcal{A}'' = \mathcal{P}_{\not\pi}\left[(\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)) \times \iota(1^{|\bar{\sigma}|}\psi, b)\right]$$
$$= \mathcal{P}_{\not\pi}\left[(\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)) \times \iota(\psi, b)\right]$$

Finally, Lemma 2 gives us that $\mathcal{A}' = \mathcal{A}''$:

$$\mathcal{P}_{\not\pi}\left[(\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)) \times \iota(\psi, b)\right]$$
$$= \mathcal{P}_{\not\pi}\left[\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\right] \times \iota(\psi, b)$$

and hence that the automaton is preserved by EVAL-DEC-ADV.

To show that the state Φ is preserved by EVAL-DEC-ADV, we simulate $\bar{\sigma}$ on the intermediate automaton $\left[\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\right] \times \iota(1^{|\bar{\sigma}|}\psi, b)$. It immediately follows that the resulting state $\Phi'' = \langle \Phi, \psi \rangle = \Phi'$. The declarations $\bar{E}$ are trivially preserved by EVAL-DEC-ADV.

We now consider the case of EVAL-CALL. We must show that in a history-based configuration $\langle \bar{\sigma}; \bar{D} \rangle$, for any declared advice $\mathsf{adv}\ a[\phi\alpha]$, if $\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi$ and $\bar{D} \vdash \bar{p}\ \mathsf{sat}\ \alpha$ where $\bar{p}$ is the role vector being called, then $\mathsf{adv}\ a[\alpha]$ is associated with the state Φ in $\mathcal{T}(\bar{\sigma}, \bar{D})$. This follows directly from Lemma 4: if $\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi$ in the history-based semantics, then in the automaton based semantics, $\phi \xRightarrow{\bar{\sigma}} \phi'$, where $\phi'\checkmark$, so $\langle \phi', a \rangle \in \Phi$.

Finally, the case of EVAL-COMMIT is trivial. Recall the evaluation rule in the history based semantics: $\bar{\sigma}; \bar{D} \rhd M, \texttt{commit}\langle\bar{p}\rangle \rightarrow \bar{\sigma},\bar{p}; \bar{D} \rhd M$, and in the automaton-based semantics:

$$\frac{\bar{D} \vdash \Phi \stackrel{\bar{p}}{\Longrightarrow} \Psi}{\mathcal{A}; \Phi; \bar{E} \rhd \bar{b}, \texttt{commit}\langle\bar{p}\rangle \rightarrow \mathcal{A}; \Psi; \bar{E} \rhd \bar{b}} \text{(EVAL-COMMIT)}$$

In this case, $\mathcal{T}(\bar{\sigma};\bar{D}) = \mathcal{A};\Phi;\bar{E}$, and $\mathcal{T}(\bar{\sigma},\bar{p};\bar{D}) = \mathcal{A};\Phi';\bar{E}$ where

$$\mathcal{A} = \mathcal{P}_{\mathcal{H}}\left[\prod_{\mathsf{adv}\ a[\phi\,\alpha]\in\bar{D}} \iota(\phi,a)\right] \qquad \bar{E} = \mathcal{T}_{dec}(\bar{D})$$

What remains is to show that $\Psi = \Phi'$. In doing so, we will have succeeded in showing that $\mathcal{A}, \Phi$, and $\bar{E}$ are all preserved by EVAL-COMMIT. By definition of $\mathcal{T}$, simulating $\bar{\sigma}$ on the intermediate automaton $\prod_{\mathsf{adv}\ a[\phi\,\alpha]\in\bar{D}} \iota(\phi,a)$ places $\mathcal{A}$ in state Φ. To derive Φ' from $\bar{\sigma},\bar{p};\bar{D}$, we simply carry the simulation one step further, taking transition $\bar{p}$. By EVAL-COMMIT in the automaton-based semantics, we know that $\Phi \stackrel{\bar{p}}{\Longrightarrow} \Phi'$, and hence that $\Psi = \Phi'$, which is what we needed to show. □

PROPOSITION 6. *If $\mathcal{A}; \Phi; \bar{E} \rhd M \rightarrow \mathcal{A}'; \Phi'; \bar{E}' \rhd M'$, and $\mathcal{T}(\bar{\sigma},\bar{D}) = \mathcal{A},\Phi,\bar{E}$, then $\bar{\sigma}; \bar{D} \rhd M \rightarrow \bar{\sigma}'; \bar{D}' \rhd M'$, where $\mathcal{T}(\bar{\sigma}',\bar{D}') = \mathcal{A}',\Phi',\bar{E}'$.*

Proof. The proof closely parallels that of Proposition 5, and as such, we omit the details here. Details can be found in Appendix B.

This brings us to the main result: that the two semantics are equivalent:

THEOREM 7. *$\bar{\sigma}; \bar{D} \rhd M \rightarrow^* \bar{\rho}; \bar{E} \rhd N$ if and only if $\mathcal{T}(\bar{\sigma}; \bar{D} \rhd M) \rightarrow^* \mathcal{T}(\bar{\rho}; \bar{E} \rhd N)$.*

Proof. By Propositions 5 and 6, and induction on the length of $\rightarrow^*$. □

7. Conclusions

We have described a novel minimal language for aspect-oriented programming with temporal pointcuts. We described an implementation of the language using security automata and proved the correctness of the implementation. We have presented examples of applications to software security.

Future work will address type-preserving translations of class-based languages into μABC. We have already developed untyped translations; finding type-preserving translations presupposes a suitable typing systems for μABC.

Acknowledgments

The comments of the anonymous referees were helpful in sharpening the focus of the paper. Alan Jeffrey and Radha Jagadeesan contributed to early discussions of this work.

References

[1] Martín Abadi and Cedric Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium Conference*, 2003.

[2] Chris Allan, Pavel Avgustinov, Sascha Kuzins, Oege de Moor, Damien Sereni, Ganesh Sittampalam, Julian Tibble Aske Simon Christensen, Laurie Hendren, and Ondřej Lhoták. Adding trace matching with free variables to aspectj. In *OOPSLA 2005*, 2005.

[3] J. Andrews. Process-algebraic foundations of aspectoriented programming. In *In Reflection, LNCS 2192*, 2001.

[4] Steve Barker and Peter Stuckey. Flexible access control policy specifi cation with constraint logic programming. *ACM Transations on Information and System Security*, 6(4):501–546, 2003.

[5] Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over dynamic properties of program execution. In *Dynamic Aspects Workshop (DAW05)*, 2005. Available at http://www.aosd.net/2005/workshops/daw/.

[6] Glen Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: A minimal aspect calculus. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004: Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, London, August 2004. Springer.

[7] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Submitted for publication, at ftp://ftp.ccs.neu.edu/pub/people/wand/papers/clw-03.pdf, oct 2003.

[8] Bruno De Fraine, Wim Vanderperren, Davy Suvée, and Johan Brichau. Jumping aspects revisited. In *Dynamic Aspects Workshop (DAW05)*, 2005. Available at http://www.aosd.net/2005/workshops/daw/.

[9] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *3rd Int. Conf. on Aspect-Oriented Software Development (AOSD'04)*, mar 2004.

[10] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns*, LNCS. Springer Verlag, September 2001. long version is http://www.emn.fr/info/recherche/publications/RR01/01-3-INFO.ps.gz.

[11] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*,

volume 1241 of *Lecture Notes in Computer Science*. Springer, June 1997.

[13] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, April 2002. ACM Press.

[14] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs.

[15] W. De Meuter. Monads as a theoretical foundation for aop. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1997.

[16] Fred Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[17] V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *RV'05 - Fifth Workshop on Runtime Verifi cation*, 2005. To Appear.

[18] Peter Thiemann. Enforcing safety properties using type specialization. In *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001*, volume 2028. Springer, 2001.

[19] David Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *Conference Record of AOSD 03: The 2nd International Conference on Aspect Oriented Software Development*, 2003.

[20] Úlfar Erlingsson and Fred Schneider. SASI enforcement of security policies: a retrospective. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA, 2000. ACM Press.

[21] David Walker. A type system for expressive security policies. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267, New York, NY, USA, 2000. ACM Press.

[22] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Conference Record of ICFP 03: The ACM SIGPLAN International Conference on Functional Programming*, 2003.

[23] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Twelfth International Symposium on the Foundations of Software Engineering*, 2004.

[24] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. appeared in Informal Workshop Record of FOOL 9, pages 67-88; also presented at FOAL (Workshop on Foundations of Aspect-Oriented Languages), a satellite event of AOSD 2002, 2002.

A. Semantics of Temporal Pointcuts

TEMPORAL POINTCUT SATISFACTION $(\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi)$

$$\frac{\bar{D} \vdash \sigma\ \mathsf{sat}\ \alpha}{\bar{D} \Vdash \sigma\ \mathsf{sat}\ \alpha}\ (\text{SAT-ATOM})$$

$$\frac{\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi}{\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi + \psi}\ (\text{SAT-OR-LEFT})$$

$$\frac{\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \psi}{\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi + \psi}\ (\text{SAT-OR-RIGHT})$$

$$\frac{\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi \qquad \bar{D} \Vdash \bar{\rho}\ \mathsf{sat}\ \psi}{\bar{D} \Vdash \bar{\sigma}, \bar{\rho}\ \mathsf{sat}\ \phi\psi}\ (\text{SAT-SEQ})$$

$$\frac{}{\bar{D} \Vdash \varepsilon\ \mathsf{sat}\ \varepsilon}\ (\text{SAT-SEQ-EMPTY})$$

$$\frac{\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi \qquad \bar{D} \Vdash \bar{\rho}\ \mathsf{sat}\ \phi^*}{\bar{D} \Vdash \bar{\sigma}, \bar{\rho}\ \mathsf{sat}\ \phi^*}\ (\text{SAT-STAR})$$

$$\frac{}{\bar{D} \Vdash \varepsilon\ \mathsf{sat}\ \phi^*}\ (\text{SAT-STAR-EMPTY})$$

B. Proof of Proposition 6

Again, in the cases of EVAL-DEC-ROLE and EVAL-ADV, this is trivial. In the case of EVAL-DEC-ADV, recall its evaluation rule:

$$\mathcal{A}; \Phi; \bar{E} \rhd \mathsf{adv}\ b[\psi\,\beta], M \to \mathcal{A}'; \Phi'; \bar{E}' \rhd M$$

where

$$\mathcal{A}' = \mathcal{A} \times \iota(\psi, b) \qquad \Phi' = \Phi, \psi \qquad \bar{E}' = \bar{E}, b$$

In the history-based semantics, we have

$$\bar{\sigma}; \bar{D} \rhd \mathsf{adv}\ b[\psi\,\beta], M \to \bar{\sigma}; \bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\,\beta] \rhd M$$

where $\mathcal{T}(\bar{\sigma}; \bar{D}) = \mathcal{A}; \Phi; \bar{E}$. What remains is to show that $\mathcal{T}(\bar{\sigma}; \bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\,\beta]) = \mathcal{A}'; \Phi'; \bar{E}'$, which we already proved in Proposition 5.

In the case of EVAL-CALL, if a $\mathtt{call}\langle\bar{p}\rangle$ is replaced by the body of some advice $\mathsf{adv}\ a[\phi\,\alpha]$, this must mean that advice a is associated with the current state of the automaton, and that $\bar{D} \vdash \bar{p}\ \mathsf{sat}\ \alpha$. We must show that in the history-based semantics, (i) the advice $\mathsf{adv}\ a[\phi\,\alpha]$ is declared (trivial), (ii) that $\bar{D} \vdash \bar{p}\ \mathsf{sat}\ \alpha$ (given), and that (iii) $\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi$. Point (iii) follows directly from Lemma 4: since $\mathsf{adv}\ a[\alpha]$ is associated with the current state, it must mean that $\phi \stackrel{\bar{\sigma}}{\Longrightarrow} \phi'$, and $\phi' \checkmark$. By Lemma 4, it immediately follows that $\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi$.

Finally, in the case of EVAL-COMMIT, recall its evaluation rule in the automaton-based semantics:

$$\frac{\bar{D} \vdash \Phi \stackrel{\bar{p}}{\Longrightarrow} \Psi}{\mathcal{A}; \Phi; \bar{E} \rhd \bar{b}, \mathtt{commit}\langle\bar{p}\rangle \to \mathcal{A}; \Psi; \bar{E} \rhd \bar{b}}\ (\text{EVAL-COMMIT})$$

If $\mathcal{A}; \Phi; \bar{E} \rhd \bar{b}, \mathtt{commit}\langle\bar{p}\rangle \to \mathcal{A}; \Psi; \bar{E} \rhd \bar{b}$, then it must be the case that $\Phi \stackrel{\bar{p}}{\Longrightarrow} \Psi$. Now, let $\bar{\sigma}; \bar{D}$ be the history-based configuration such that $\mathcal{T}(\bar{\sigma}; \bar{D}) = \mathcal{A}; \Phi; \bar{E}$. Then by definition of $\mathcal{T}$,

$$\mathcal{A} = \mathcal{P}_{\not\pi}\left[\prod_{\mathsf{adv}\ a[\phi\,\alpha] \in \bar{D}} \iota(\phi, a)\right] \qquad \bar{E} = \mathcal{T}_{dec}(\bar{D})$$

Recall the rule in the history-based semantics:

$$\bar{\sigma}; \bar{D} \rhd M, \mathtt{commit}\langle\bar{p}\rangle \to \bar{\sigma}, \bar{p}; \bar{D} \rhd M$$

We must show that $\mathcal{T}(\bar{\sigma},\bar{p};\bar{D}) = \mathcal{A};\Phi';\bar{E}$ where $\Phi' = \Psi$. $\mathcal{A}$ and $\bar{E}$ follow immediately from the definition of $\mathcal{T}$.

Furthermore, by definition of $\mathcal{T}$, simulating $\bar{\sigma}$ on the intermediate automaton $\prod_{\mathsf{adv}\, a[\phi\,\alpha]\in\bar{D}} \iota(\phi,a)$ puts the automaton in state Φ. To derive Φ' from $\bar{\sigma},\bar{p};\bar{D}$, we simply carry the simulation one step further, taking transition $\bar{p}$. By EVAL-COMMIT in the automaton-based semantics, we know that $\Phi \xRightarrow{\bar{p}} \Phi'$, and hence that $\Phi' = \Psi$, which is what we needed to show. □

C. Derived Forms

To give a feel for the language, we define a few derived forms and discuss their execution.

This is an encoding of let that uses roles for continuations. In this encoding we require an additional reserved role continue.

DERIVED FORMS (LET) (c fresh)

$\mathsf{role}\, p \triangleq \mathsf{role}\, p < \mathsf{top}$	Trivial Role
$\mathsf{let}\, x = N;\, M \triangleq \mathsf{role}\, c;$ $\quad \mathsf{adv}[\langle c, +\mathsf{top}\rangle] = (\_, x)\, M;$ $\quad N\{^{c}/_{\mathsf{continue}}\}$	Let
$\mathsf{ret}\, p \triangleq \mathsf{call}\langle \mathsf{continue}, p\rangle$	Return
$M;N \triangleq \mathsf{let}\, \_ = N;\, M$	Sequencing

For example, we have the following.

$$
\begin{aligned}
&\mathsf{let}\, x = N;\, \mathsf{let}\, y = L;\, M \\
&\quad = \mathsf{role}\, c; \\
&\qquad \mathsf{adv}[\langle c, +\mathsf{top}\rangle] = (\_, x)\ \mathsf{let}\, y = L;\, M; \\
&\qquad N\{^{c}/_{\mathsf{continue}}\} \\
&\quad = \mathsf{role}\, c; \\
&\qquad \mathsf{adv}[\langle c, +\mathsf{top}\rangle] = (\_, x)\ \ \mathsf{role}\, d; \\
&\qquad\qquad\qquad\qquad\ \ \mathsf{adv}[\langle d, +\mathsf{top}\rangle] = (\_, y)\, M; \\
&\qquad\qquad\qquad\qquad\ \ L\{^{d}/_{\mathsf{continue}}\}; \\
&\qquad N\{^{c}/_{\mathsf{continue}}\}
\end{aligned}
$$

For example, we have the following.

$$
\begin{aligned}
&\mathsf{let}\, x = (\mathsf{let}\, y = L;\, N);\, M \\
&\quad = \mathsf{role}\, c; \\
&\qquad \mathsf{adv}[\langle c, +\mathsf{top}\rangle] = (\_, x)\, M; \\
&\qquad \mathsf{let}\, y = L;\, N\{^{c}/_{\mathsf{continue}}\} \\
&\quad = \mathsf{role}\, c; \\
&\qquad \mathsf{adv}[\langle c, +\mathsf{top}\rangle] = (\_, x)\, M; \\
&\qquad \mathsf{role}\, d; \\
&\qquad \mathsf{adv}[\langle d, +\mathsf{top}\rangle] = (\_, y)\, L\{^{d}/_{\mathsf{continue}}\}; \\
&\qquad N\{^{c}/_{\mathsf{continue}}\}
\end{aligned}
$$

DERIVED FORMS (FUNCTIONS) (f and x fresh)

$\lambda x.N \triangleq \mathsf{role}\, f;$ $\quad \mathsf{adv}[\langle f, +\mathsf{top}, +\mathsf{top}\rangle] = (\_, x, c)\, N\{^{c}/_{\mathsf{continue}}\};$ $\quad \mathsf{ret}\, f$	Abstraction
$L\, M \triangleq \mathsf{let}\, f = L;$ $\quad \mathsf{let}\, x = M;$ $\quad \mathsf{call}\langle f, x, \mathsf{continue}\rangle$	Application

For example, we have the following.

$$
\begin{aligned}
(N\, L)\, M &= \mathsf{let}\, f = N\, L; \\
&\qquad \mathsf{let}\, x = M; \\
&\qquad \mathsf{call}\langle f, x, \mathsf{continue}\rangle \\
&= \mathsf{let}\, g = N; \\
&\qquad \mathsf{let}\, y = L; \\
&\qquad \mathsf{let}\, f = \mathsf{callcc}\langle g, y\rangle; \\
&\qquad \mathsf{let}\, x = M; \\
&\qquad \mathsf{call}\langle f, x, \mathsf{continue}\rangle
\end{aligned}
$$

Fine-Grained Generic Aspects

Tobias Rho, Günter Kniesel, Malte Appeltauer
Dept. of Computer Science III
University of Bonn
Römerstr. 164, D-53117 Bonn
Germany
{rho,gk,appeltau}@cs.uni-bonn.de

ABSTRACT
In theory, join points can be arbitrary places in the structure or execution of a program. However, most existing aspect languages do not support the full expressive power of this concept, limiting their pointcut languages to a subset of the theoretically possible join points. In this paper we explore a minimal language design based on only three built-in *fine-grained pointcuts*, which enable expressing the entire spectrum of structures of an underlying base language, from types to statements and expressions. The combination of fine-grained pointcuts with *uniform genericity* in our *LogicAJ 2* language yields the concept of *fine-grained generic aspects*. We demonstrate their power by showing how they allow programmers to express and extend the static primitive pointcuts of AspectJ and how they can model applications that previously required run-time reflection or special purpose language extensions.

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

General Terms
Languages

Keywords
Fine-Grained Genericity, Homogeneous Generic Aspect Language, Program Transformation, Multi Join Points, Fine-Grained Pointcuts

1. INTRODUCTION
The notion of join points is central to aspect-oriented programming languages. Join points are well-defined places in the structure or execution flow of a program [4], [5], [3], [21], [16]. In theory, they could be arbitrary program elements or run-time events. In practice, however, the classes of join points supported by most existing aspect languages are limited. Method call, method execution, field access and field modification are the typical join points that are widely supported. Different researchers [7], [6], [17], [9] have noted independently that finer grained join points are necessary in various application areas. For instance, Kniesel and Austermann [9] show that thorough code coverage analysis requires access to every individual statement in a program. They present a professional code coverage tool for Java based on load-time byte code adaptation [11]. Sullivan and H. Rajan [17] address the same problem domain but provide a solution at a higher level of abstraction. They show how code coverage analysis can be implemented in a language that provides statement-level join points. Their approach uses reflection and generates new aspects based on reflective information at join points. Unfortunately, this makes static type checking impossible.

Another impressive application area for fine-grained pointcuts is the automatic detection and optimization of highly parallel loops. Figure 1 shows a simple example. B. Harbulot and R. Gurd [7] demonstrate that with *AspectJ* [8] parallelization of such loops is only possible after refactoring the loop into a new method with a defined signature pattern that makes its lower and upper bounds explicit. Because typical code in parallel scientific applications almost never makes loop bounds explicit as methods the authors conclude that statement-level join points are needed to enable aspect-based optimizations of highly parallel programs. In *LoopsAJ* [6] they provide a solution tailored specifically to the interception of loops. However, their solution also imposes some constraints on the structure of the code in and before loops.

```
1 public void m(){
2   int[] a = new int[42];
3   int[] b = new int[42];
4
5   for(int i = 0;i< a.length;i++) {
6     a[i] = 2*i;
7     b[i] = i*i;
8   }
9 }
```

Figure 1. Simple highly parallel *for* loop

The work reported in this paper goes beyond previous approaches in that it provides a comprehensive design for fine-grained pointcuts in an extensible, statically checkable, high-level language. Although our aspect language, *LogicAJ 2*[1], provides only 3 built-in pointcuts, it is able to express all possible join points whose shadows are elements of the base language (in our case Java). We demonstrate the expressiveness of our concept by showing how it allows programmers to express and extend the static primitive pointcuts of AspectJ and how to model applications that previously required special purpose language extensions. In particular, we show that LogicAJ 2‘s combination of fine-grained genericity and extensibility can express the functionality of LoopsAJ, without imposing any constraints on the structure of base programs.

[1] LogicAJ 2 is based on the generic aspect language LogicAJ [10].

Section 2 introduces our language design. Section 3 demonstrates how the language can be easily extended by expressing 'basic' poincuts of AspectJ using fine-grained genericity. In Section 4 we give two examples of modelling applications that previously required specific language extensions: the Law Of Demeter and the for-loop pointcut. Section 5 discusses related work and Section 6 ongoing work. Finally, section 7 concludes.

2. FINE-GRAINED GENERICITY

In this section we gradually introduce the basic concepts behind LogicAJ 2 and illustrate them on small examples.

2.1 Basic Pointcuts

The aim of our design was the identification of a minimal, orthogonal set of pointcuts that are able to express more complex ones offered in other languages. Our analysis resulted in just *three basic pointcuts*, representing the distinct classes of basic elements of any programming language: declarations, statements and expressions. Their syntax is:

- `decl(join_point, declaration_code_pattern)`
- `stmt(join_point, statement_code_pattern)`
- `expr(join_point, expression_code_pattern)`

The first argument of each basic pointcut is an explicit representation of the matched join point (see Section 2.4). The second argument is a pattern describing the join point (see Section 2.2).

The expr pointcut selects any expression matching the given source code pattern and binds the join_point argument to an explicit representation of the matched join point. The stmt pointcut does the same for statements. These two pointcuts can match any element within a method body. The decl pointcut additionally matches declarations of classes, interfaces, methods and fields.

Taken together, these three pointcuts can match any structure of a base (Java) program, from the coarsest to the finest granularity. Therefore we also call them *fine-grained pointcuts*.

The basic pointcuts can be used to bind any syntax element of their domain, by omitting the code pattern.

2.2 Logic Meta-Variables

Unlike most aspect languages, we do not provide a special syntax for the patterns used to specify join points. Instead, join point descriptions are simply base language code or patterns resulting from the ability to use placeholders for all base language elements that are not syntactic delimiters or keywords.

Instead of unnamed wildcards our placeholders are named *logic meta-variables* (LMV). Meta-variables are variables that can range over syntactic elements of the base language (e.g. Java). They are denoted syntactically by names starting with a question mark, e.g. "?methodBody".

Named meta-variables give us the ability (1) to express that different occurrences of the same placeholder must agree on the matched value and (2) to use the matched values as a building block of advice code. Rho and Kniesel [10], [12] show that uniform use of logic meta-variables in pointcuts and advice (*uniform genericity*) increases the expressiveness, reusability and modularity of aspects – even without the added power of fine-grained pointcuts introduced here. The combination of uniform genericity and fine-grained basic pointcuts is called *fine-grained genericity*. We demonstrate the increased expressiveness of fine-grained genericity in Section 3 and 4.

In addition to logic meta-variables that have a one-to-one correspondence to individual base language elements, *logic list meta-variables* (LLMV) can match an arbitrary number of elements, e.g. arbitrary many call arguments or method parameters. These variables are indicated syntactically by two leading question marks, e.g. "??parameterList". Their introduction is motivated by the fact that in truly generic application scenarios one often needs to say things like "match every constructor invocation" or "add a forwarding method for every method from type T" or "select all update expressions in a for-loop". In such cases it is neither possible to know the exact number of parameters of an invocation or a method, nor is it practical to specify a finite set of method argument lists. The language provides a set of built-in operations on LLMVs, e.g. concatenation and member check.

Unnamed logic meta-variables are indicated by an underscore (?_ and ??_). If a pointcut contain several unnamed meta-variables, they are all treated as distinct variables.

2.3 Named Pointcuts

Using the basic pointcuts, programmers can define arbitrary custom pointcuts. Custom pointcut definitions can be named and can have meta-variables as arguments. Unlike in AspectJ, for instance, custom pointcuts can be defined recursively. This is useful for expressing transitive relationships, for instance the subtype relation. The recursive definition of a generalized version of AspectJ's withincode pointcut is discussed in Section 2.5.

2.4 Explicit Join Points

Our language design leverages on the power of meta-variables by making the join point selected by a basic pointcut explicit as a meta-variable argument. This is an extremely powerful concept, since it makes join points first class entities of the aspect language.

Figure 2 shows a pattern that selects if statements. Upon every match the ?if meta-variable is bound to the complete matched statement (the join point), whereas the meta-variables contained in the pattern are bound to the respective sub-elements of the statement. In this case ?cond is bound to the condition expression and ??someStatements is bound to the list of statements in the body of the if statements' block.

```
stmt(?if, if(?cond){??someStatements})
```

Figure 2. Selection of an if statement, its condition and its body

Figure 3 shows the use of two expr pointcuts that select all calls of the methods *foo* and *bar*. The matched join points are explicitly represented by the meta-variable ?jp, which is passed as a parameter to the pointcut definition. Thus, it can be used as the join point of an advice based on fooBarCalls(?jp).

```
1  pointcut fooBarCalls(?jp):
2      expr(?jp , foo() )
3   || expr(?jp , bar() )
```

Figure 3. Join points made explicit via meta-variables

Alternatively, we can reuse this pointcut as shown in Figure 4. Note that the meta-variable ?call is used within the if statement pattern in the stmt pointcut *and* within fooBarCalls. This way we express that the calls to foo and bar must be the condition of an if statement. Note further that the defined pointcut provides ?if and ?call as parameters to its users. Thus it does not predetermine whether the matched if statement or the matched call is the join point that it selects.

```
1 pointcut fooBarCallsWithinIf(?if, ?call):
2     stmt(?if, if(?call){??someStatements} )
3  && fooBarCalls(?call) ;
```

Figure 4. Refining the pointcut from Figure 3 to select calls of foo or bar that occur within an if-condition

Since different meta-variables can represent different join points at the same time it is possible to express relations between join points, as illustrated in Figure 4. This is an extremely powerful concept. In Section 4.1 we demonstrate how it enables a concise implementation of the Law of Demeter.

In addition it gives us the option to let a generic advice choose at which of the multiple join points the advice code should be woven. Therefore the syntax of LogicAJ 2 advice[2] was slightly extended. The target join point of the advice must be specified as the *first* argument of the advice. The advice shown in Figure 5 counts all invocations of foo or bar that occur as the condition of an if statement. If we change the advice parameter to ?if the advice will count the number of if statements whose conditions are calls to foo or bar.

```
1 around(?call):  fooBarCallsWithinIf(?if, ?call) {
2   Counter.count++;
3 }
```

Figure 5. Explicit choice of the effective join point for an advice

2.5 Meta-Variable Attributes

For many uses, it is not sufficient to consider only a syntactic element itself but also the static context of the element. For example, the declaring type is important information about a method or a field declaration. Similarly, the statically resolved binding between a method call and its called method or between a variable access and the declared variable is necessary for several pointcuts.

We make this information available via LMV *attributes*. An attribute *a* of a meta-variable ?*mv* is accessible via:

?mv::a

Figure 6 describes the attributes used in the remainder of the paper. They are a subset of the attributes supported by LogicAJ 2.

Attrib.	Represented context information of a LMV
parent	The enclosing element of the syntax element represented by the LMV.
ref	The statically resolved declaration referenced by an expression: a call references a method, and an identifier a field, variable or parameter declaration.
type	The statically resolved Java type of an element bound to a LMV. This attribute is syntactic sugar. It is inferable via the ref attribute.

Figure 6. LMV attributes provide additional information about the syntactic elements' context and the resolved Java bindings.

Figure 7 demonstrates the use of the parent attribute for the withincode pointcut, known from AspectJ. It checks if a join point is defined in the body of a given method. We present a generalized version that checks the withincode relationship of statements and expressions to any enclosing element.

```
1 pointcut withincode(?jp,?enclosing):
2     ( expr(?jp)  ||  stmt(?jp) )
3   && ( equals(?jp::parent, ?enclosing)
4        || withincode(?jp::parent, ?enclosing)
5      );
```

Figure 7. Definition of the `withincode` pointcut in LogicAJ 2

First, the ?jp variable is bound to an expression or statement. The *equals* predicate has a double role. If ?enclosing was bound to a value before withincode was called, *equals* just checks if the value of ?enclosing is ?jp's parent element Otherwise it binds ?enclosing to ?jp::parent. In order to get all the directly and indirectly enclosing elements, of ?jp the pointcut is evaluated recursively for the parent of the ?jp.

3. EXTENSIBILITY OF THE POINTCUT LANGUAGE

This section shows how basic pointcuts can be used to build pointcuts known from common aspect languages. Designing semantic meta-levels with basic pointcuts drastically enriches the usability of pointcuts and is an important criterion for the expressiveness of an AO language.

3.1 Static AspectJ Pointcuts

The pointcuts offered by AspectJ are very useful. Due to the expressiveness of our minimalist pointcut language we can define custom pointcuts that implement AspectJ pointcut semantics with little effort. We have already shown the implementation of the withincode pointcut in Figure 7. In this section we show the implementation of the call and get pointcut in LogicAJ 2. The other static pointcuts of AspectJ can be implemented following the same scheme.

3.1.1 Call Pointcut

Our implementation of the AspectJ call pointcut (Figure 8) starts with an expr pointcut selecting the call expression, (line 4) and a decl pointcut binding the called method (lines 5-6). The equals predicate in line 7 denotes that the call join points are statically bound to the method ?method. Line 8 binds the declaring type of ?method to ?declType by using the ref and type attributes (see Line 8). Line 9 binds the ??parTypes list to the types of the method parameters. We omit the definition of the parameterTypes pointcut. It can easily be implemented as a recursive pointcut using the type argument.

```
1 pointcut call(?jp, ?declType, ??modifiers,
2               ?returnType, ?name, ??parTypes):
3
4     expr(?jp, ?name(??args) )
5  && decl(?method,
6       ??modifiers ?returnType ?name(??par){??stmts} )
7  && equals(?method, ?jp::ref)
8  && equals(?declType, ?method::parent::type)
9  && parameterTypes(??parTypes,??par);
```

Figure 8. Implementation of the call pointcut

Figure 9 shows two usage examples of the call pointcut. For comparison, the respective AspectJ syntax is shown as a comment. Line 1-3 illustrates the selection of calls to the method with signature m(int) in class Foo. Line 5-7

```
1 //AJ:  pointcut m(): call( void Foo.m(int) )
```

[2] and declare error/warning constructs

```
2 pointcut m(?jp):
3      call(?jp, Foo, ??_, void, m, [int]);
4
5 //AJ:  pointcut anycall(): call(* *.*(..));
6 pointcut anycall(?jp): call(?jp,?_,??_,?_,?_,??_);
```

Figure 9. Comparison between AspectJ and LogicAJ 2 call syntax. Square brackets (line 3) denote a list of values.

3.1.2 *Get Pointcut*

Our next example implements AspectJ's `get` pointcut. It selects field read accesses. Field declarations can have different syntactic forms. For instance, they can be declared with or without a value assignment. Each syntactic occurrence selects a different set of join points. Figure 10 presents the unification of the different syntactic join point variants by a more general field access pointcut. The union is expressed by the disjunction in lines 4-5, which states that all declarations with *or* without value assignment are bound to ?field.

```
1 pointcut get(?jp,??modifier,?declType,?name,?retType):
2      expr(?jp,?name )
3   && (
4           decl(?field,??modifier ?retType ?name; )
5        || decl(?field,??modifier ?retType ?name = ?v;)
6      )
7   && equals(?field, ?jp::ref)
8   && equals(?declType, ?field::parent::type);
```

Figure 10. Implementation of AspectJ get pointcut semantics

In this example, join points are selected on the syntactic, not on the semantic level. However, we do not see this as a limitation of our approach. The defined custom pointcut implements a semantic selection criterion. It can be reused, hiding the syntactic details.

3.2 New Pointcuts

In the following, we give examples illustrating how easy it is to define additional semantic pointcuts that are neither built-in nor expressible in common AO languages.

3.2.1 *Local Variable Access Pointcuts*

As a complement to the common get and set pointcuts that select fields, we introduce getL and setL pointcuts that select read and write accesses to *local* variables. We describe the implementation of the getL pointcut in detail. The setL implementation is analoguous.

```
1 pointcut getL(?jp,?type,?name):
2     //Select all identifier expressions:
3     expr(?jp, ?name)
4     // Select all local variable declarations:
5     &&(   stmt(?localdecl, ?type ?name;)
6        || stmt(?localdecl, ?type ?name = ?val;)
7        )
8     // Check that the local variable declaration
9     // is referenced by the identifier:
10  && equals(?localdecl, ?jp::ref) );
```

Figure 11. Implemantation of new pointcuts: the getL pointcut only selects local variable accesses

The intended semantics of getL is to select all identifiers whose declaration is a local variable. We start by selecting all expressions that have the form of an identifier, that is, consist of a single name. This is done by the expr(?jp, ?name) pointcut (Figure 11, line 3). The set of identifiers matched this way can also contain fields and parameters. In order to understand how we limit it to local variables only, it is helpful to recall that, in Java, the declaration of a local variable is a *statement*. Accordingly, we enclose each of the code patterns corresponding to a variable declaration into a stmt pointcut (see Figure 11, lines 5-6). The final equals predicate checks whether the selected local declaration is indeed the declaration of the identifier matched in line 3.

3.2.2 *Field Pointcut*

Within the declarations of the get pointcut we had to consider syntactic differences of field declarations. We can encapsulate those within a field pointcut definition in order to achieve a more modular and readable implementation of get and other pointcuts that deal with field declarations. We will see another use of the field pointcut in Section 4.1.

The definition illustrated in **Figure 12** selects all field declarations, with and without initializers. Then it determines the declaring type by accessing the static join point context captured by the meta-variable attributes ?jp::parent::type. Similarly, we can implement a method or a class pointcut that abstracts from the syntactic variants of the base language.

```
1 pointcut field(?jp,?declaringType,?returnType,?name):
2   (
3        decl(?jp, ?returnType ?name; )
4     || decl(?jp, ?returnType ?name = ?anyVal;)
5   )
6   && equals(?declaringType, ?jp::parent::type);
```

Figure 12. `field` semantics

4. EXAMPLES

We now consider two different use cases that rely on fine-grained pointcuts. Section 4.1 presents a simple check for the static variant of the Law of Demeter, expressing a contract previously claimed to require extension of *AspectJ* by *statically executable advice* [14]. Section 4.2 comes back to the high-performance computing example.

4.1 Example: Law Of Demeter

Binding several join points at the same time enables very expressive pointcuts. This section gives a thorough example. It shows how the declare warning construct can be used to check the Law of Demeter with the help of a fine-grained pointcuts.

The *Law of Demeter* (*LoD*) [13] restricts the method calls used in a class *C* to methods from 'known' types. These include

1. *C*,
2. the types of the calling method's parameters,
3. the types of *C*'s fields,
4. the return types of *C*'s methods
5. the classes instantiated in *C and*
6. all the supertypes of any of the known types from 1-5.

Figure 13 shows the *LawOfDemeter* aspect, concisely implemented in *LogicAJ 2*. The aspect uses the custom pointcuts method, constructorcall and the subtype, which can be defined easily, like the pointcuts in Section 3.

The pointcut knownTo defined in line 3-11 encapsulates rules 1-5 of the LoD. It defines the basic set of types known to a method. Its semantics is that ?Type is known to a method with parameter types ?ParameterTypes contained in ?CallingType.

- Line 4 checks the first rule: The ?CallingType is known to itself.

- Line 5 checks the second rule: Every member of ?ParamTypes is known. The member predicate successively binds ?Type to an element of ?ParamTypes.
- Line 6 checks the third rule: The type of every field of ?CallingType is known. The field predicate successively binds ?Type to the type of a field of ?CallingType.
- Line 7 checks the fourth rule: The return type of every method of ?CallingType is known. The method predicate successively binds ?Type to the return type of a method of ?CallingType.
- Lines 10-11 check the fourth rule: The type instantiated by a constructor call contained in the ?CallingType is known. The constructorcall predicate in line 9 determines all constructor calls and the withincode predicate in line 11 ensures that only calls within ?CallingType are regarded.

The knownTo pointcut is the core of the real LoD checking in line 13-25. Lines 14-17 set the stage by binding the meta-variable ?CalledType to the static receiver type of a method call, ?CallingType and ?CallingMethod to the type and method containing the call, and ??ParamTypes to the list of parameter types of ?CallingMethod. Line 21 uses the knownTo pointcut to determine a known ?Type and line 22 verifies whether ?CalledType is a supertype of the known ?Type[3]. If not, a violation is reported in line 24 along with the violating call.

This example uses explicit join point LMVs (see Section 2.4). The call and the constructorcall pointcuts are used within a single scope: the one in line 14 and the nested one in the definition of knownTo (line 9). This example could not be expressed if the join points could not be explicit parameters of the pointcuts in line 9 and 10. If there were only implicit join points (as in *AspectJ*), it would not be clear that constructor calls should not be reported as LoD violations and that failure of matching constructor calls in line 9 should not inhibit reporting a violation of other calls.

```
1 aspect LawOfDemeter {
2
3  pointcut knownTo(?CallingType,??ParamTypes,?Type):
4      equals(?Type, ?CallingType)                 //rule 1
5   || member(?Type, ??ParamTypes)                 //rule 2
6   || field(?jp,?CallingType, ?Type, ?Fname)      //rule 3
7   || method(?jp,??_,?Type,?CallingType,?Mname,??_)//r.4
8   ||
9   (
10        constructorcall(?ConstrCall,?Type,??_)  //rule 5
11     && withincode(?ConstrCall, ?CallingType )  //rule 5
12   );
13
14  declare warning:
15      call(?called,?CalledType,??_,?_,?CalledMeth,??_)
16   && method(?call,??_,?_,?CallingType,?CallingMeth,
17              ??ParamTypes )
18    && withincode(?called, ?call)
19    &&
20    !(
21           knownTo(?CallingType,??ParTypes,?Type)//r.1-5
22        && subtype(?Type, ?CalledType)          //rule 6
23      )
24     : "The call violates the Law of Demeter.";
25 }
```

Figure 13. Aspect checking the Law of Demeter (LoD) at weave-time. It reports a warning for every method invoked on a type that is not among those "known" to the calling type or the parameters of the calling method.

[3] Note that every type is a super type of itself. Subtype can be declared recursively, similar to withincode (see Figure 6)

4.2 Example High-Performance Computing

The following section shows how the execution of highly parallel loops can be distributed by a generic aspect onto a set of threads, following the approach described in [6]. Unlike the approach in [6] our solution does not rely on code conventions.

The target of a high-performance aspect could be the following for-loop, whose body uses the local array variables a and b.

```
1 public void m(){
1     int[] a = new int[42];
2     int[] b = new int[42];
3
4   for(int i = 0;i< a.length;i++) {
5     a[i] = i*i;
6     b[i] = i*i*i;
7   }
8 }
```

Figure 14. Simple `for`-loop

The detection of the *highly parallel* loop can be performed with the simple detector presented in Figure 15. The pointcut highlyParallelLoop checks that no method call is present in the block (avoiding side-effects) and tests that values are exclusively read from (or written to) a variable in the block.

This ensures that the order of the computation does not affect the result. Below we see the pointcut describing these checks. We assume to have implemented withincode, set, setL, get and getL pointcuts as shown in Section 3.

```
1 pointcut highlyParallelLoop(?jp,?range,?lb,
2                             ?ub,?body,?incr):
3     //selects for-loops
4  stmt(?jp,
5     for(?type ?range=?lb; ?range<?ub; ?incr){?body})
6     // selects all join points within the loop body
7  && withincode(?stmts, ?body)
8     // excludes calls within loops
9  && !call(?stmts,?_,??_,?retType,?name,??args)
10 && // no read AND write access to a variable allowed
11 !(
12       setL(?bodyJPs, ?body, ?type, ?name, ?val)
13    && withincode(?another, ?body)
14    && getL(?another, ?body, ?type, ?name, ?val)
15 )
16 &&  // the same for fields
17 !(
18       set(?bodyJPs,?p,?cl,?meth,??mod,?type,?name,?val)
19    && withincode(?another, ?body)
20    && get(?another,?p,?cl,?meth,??mod,?type,?name,?val)
21 );
```

Figure 15. The pointcut selecting highly parallel loops.

The around advice in Figure 16 wraps the for-loop into a run() method of a thread and modifies the bounds of the loop. The for loop now runs in parallel in several threads, each thread calculating only a uniform range of the loop.

```
1 around(?jp) :
2   highlyParallelLoop(?loop,?range,?lb,?ub,?body,?incr){
3
4    int THREADS = 5;
5    //resolve the value of the bounds
6    final ?ub::type ub = ?ub;
7    final ?lb::type lb = ?lb;
8    final ?range::type range = (ub - lb)/THREADS;
9    List list = new ArrayList();
10
```

```
11    for(int threads=0; threads < THREADS; threads++){
12      final int finalThreads = threads;
13      Thread thread = new Thread() {
14        public void run(){
15          ?range::type newLb = lb+range*finalThreads;
16          ?range::type newUb = newLb + range;
17
18          if(newUb >= ub)
19            newUb = ub;
20
21          for(?range::type ?range = newLb;?range < newUb;
22             ?incr){
23                ?body
24          }
25        }
26        thread.run();
27        list.add(thread);
28      }
29
30     for(int threads = 0;threads<list.size();threads++)
31        try{
32            ((Thread)list.get(threads)).join();
33        }
34         catch(InterruptedException e){
35            e.printStackTrace();
36         }
37      }
38    }
```

Figure 16. The for-loop parallelization aspect.

5. RELATED WORK

A comparison with existing generic aspect languages is given in [12]. It includes a thorough comparison of LogicAJ 2 to related work from logic meta-programming and program transformation systems. Here we confine ourselves to related work specific to fine-grained genericity.

The *TyRuBa* language [19] introduces logic meta-programming for Java programs by defining Prolog-like predicates on Java programs. All code blocks of the base program are represented as quoted Java code within the *TyRuBa* rules. The quoted code may contain meta-variables for types and identifiers. In *TyRuBa* the quoted code can only be used for the generation of Java code, but not in the query language.

Several approaches exist that describe finer-grained extensions to the *AspectJ* join point model.

The extensible AspectJ compiler *abc* [1] provides a Java API to extend the set of known *AspectJ* pointcuts. For every additional pointcut, the lexer, parser and weaver must be extended to support the new pointcut. Implementation of custom pointcuts in LogicAJ 2 does not require any such changes. Es we have shown, the extensions are expressible within the language.

EOS-T [17] extends the *AspectJ* primitive pointcuts with pointcuts for conditionals and loops. It does not provide no ability to refer to join points statement-arguments or -blocks. Harbulot presents *LoopAJ* [6], an AspectJ extension for loop pointcuts. His approach is bult on abc and uses byte code analysis to identify loops. Kniesel and Austermann [9] present a professional code coverage tool for Java, *CC4J*, implemented based on the *JMangler* load-time adaptation framework [11]. Working at byte code level, however, is not the preferred level of abstraction for most programmers.

Borba et al. introduce *JaTS* [2], a language for pattern based transformations of Java programs. Similar to our basic pointcuts, code patterns are used to describe program parts on which transformations should take place. The transformation specification is described with another pattern. Like in *LogicAJ* 2, both parts can be linked by the use of meta-variables, which substitute syntactic elements at the interface level of a base-program. According to personal communication with the authors, meta-variables can also match finer grained elements. That lets JaTS appear to be the closest match for our concept of fine-grained genericity. Comparison of *JaTS* and *LogicAJ* 2 will therefore be a rewarding topic of future work.

6. ONGOING WORK

The design of LogicAJ 2 described in this paper is currently implemented as an extension of our existing LogicAJ compiler, which is available at [15].

The added expressive power of a generic aspect language does not come for free. In particular, static analysis of aspect code is difficult in the presence of meta-variables.

In order to prevent substitution of statements where expressions are expected and vice-versa, meta-variables need to be *syntactically typed*, that is every meta-variable needs to have a type that determines the kind of syntactic entity from the base language that may be substituted. Syntactic types can either be declared or inferred from the definition of the predicates that are used to bind meta-variable values. For lack of space, we did not address this issue in this paper. This is a topic of ongoing work.

Currently we do not support dynamic join points like cflow, this or target with our basic pointcut model. In contrast to static join points, dynamic ones have no counterpart in the base program that could be described by a unique code pattern. Overcoming this limitation is also subject of ongoing work.

We will evaluate LogicAJ 2 by applying fine-grained genericity to different application areas. General software transformation approaches, like [18] have addressed optimizations techniques like partial evaluation and data-flow optimization with generic transformations. We will analyze how they can be translated to fine-grained generic aspects.

Contract4J [20] uses *AspectJ* to check contracts on Java. Currently the contracts are limited to *AspectJ* join points. For instance loop invariants can not be checked. Fine-grained genericity could be used to remove this restriction.

7. CONCLUSION

In this paper, we have introduced the concept of fine-grained genericity for aspect languages. Our approach is based on a minimal set of fine-grained pointcuts and base-language code patterns containing logic-meta variables. This enables us to express context-dependent aspect effects and dependencies between multiple join points. In addition, we have shown that fine-grained genericity is able to express the static pointcuts known from *AspectJ* and to define arbitrary other kinds of pointcuts that previously required specific language extensions.

Thus, we have shown that there is no need for extending an aspect language in order to implement new ‘basic’ pointcuts if the language itself is powerful enough to select *all* base-language join points.

8. ACKNOWLEDGEMENTS

We want to thank the FOAL’06 reviewers for their constructive and knowledgeable comments on the submitted draft of our paper.

9. REFERENCES

[1] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. abc: An extensible AspectJ compiler, Transactions on Aspect-Oriented Software Development, 2005

[2] Castor, F., and Borba, P. A language for specifying Java transformations. In V Brazilian Symposium on Programming Languages, pages 236-251, Curitiba, Brazil, 23rd-25th May 2001.

[3] Conejero, J. M., van den Berg, K., and Chitchyan, R. Aosd ontology. http://www.aosdeurope.net.

[4] Filman, R. E., and Friedman, P. D. Aspect-Oriented Programming is Quantification and Obliviousness, Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis.

[5] Filman, R. E., Elrad, T., Clarke, S., and Aksit, M. Aspect-Oriented Software Development. Addison-Wesley, Boston, 2005.

[6] Harbulot, B., and Gurd, J. R. A join point for loops in aspectj. FOAL 2005, 2005.

[7] Harbulot, B., and Gurd, J. R. Using AspectJ to Separate Concerns in Parallel Scientific Java Code, Proceedings of 3rd International Conference on Aspect-Oriented Software Development (AOSD), March 2004

[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. An Overview of AspectJ, Proceedings of ECOOP 2001

[9] Kniesel, G., and Austermann, M. CC4J - Code Coverage for Java, Lecture Notes in Computer Science, Volume 2370, Jan 2002, Page 155

[10] Kniesel, G., and Rho, T. Beyond Type Genericity - Homogeneous Genericity for Aspect Languages, Technical Report IAI-TR-2004-4, University of Bonn, Dec 2004.

[11] Kniesel, G., Costanza, P., and Austermann, M. JMangler - a framework for load-time transformation of Java class files Source Code Analysis and Manipulation. Proceedings. First IEEE International Workshop on 10 Nov. 2001 Page(s):98 – 108

[12] Kniesel, G., Rho, T., Generic Aspect Languages - Needs, Options and Challenges, Special issue of L'Objet, Hermes Science Publishing, London, 2006

[13] Lieberherr, K. J., Holland, I., Riel, A.: Object-Oriented Programming: An Objective Sense of Style, *In Proceedings of the Conference on Object-oriented Systems, Languages and Applications (OOPSLA)*, San Diego, California, USA, pages 38-48, September 1988.

[14] Lieberherr, K., Lorenz, D., and Wu, P. A case for statically executable advice – checking the law of Demeter with AspectJ, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, MA, March 17-21, ACM Press, 2003

[15] LogicAJ Homepage, http://roots.iai.uni-bonn.de/research/logicaj

[16] Masuhara, H., Kiczales, G., and Dutchyn, C. A Compilation and Optimization Model for Aspect-oriented Programs. Compiler Construction (CC) also Lecture Notes in Computer Science (LNCS) vol. 2622.

[17] Rajan, H., and Sullivan, K. "Generalizing AOP for Aspect-Oriented Testing", In the proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD 2005), 14-18 March, 2005, Chicago, IL, USA

[18] Visser, E., Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer et al., editors, Domain-Specific Program Generation, *volume 3016 of Lecture Notes in Computer Science*, pages 216--238. Spinger-Verlag, June 2004.

[19] Volder, K. D. Aspect-oriented logic meta programming. In Proceedings of the Second International Conference on Metalevel Architectures and Reflection, volume 1616 of Lecture Notes in Computer Science . Springer-Verlag,1999

[20] Wampler, D., Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces, Fifth AOSD Workshop on ACP4IS, Bonn 2006

[21] Wand, M., Kiczales G., et al. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming, LNCS 2196, 2001

On the Pursuit of Static and Coherent Weaving

Meng Wang
National University of Singapore, Singapore
wangmeng@comp.nus.edu.sg

Kung Chen
National Chengchi University, Taiwan
chenk@cs.nccu.edu.tw

Siau-Cheng Khoo
National University of Singapore, Singapore
khoosc@comp.nus.edu.sg

Abstract

Aspect-oriented programming (AOP) has been shown to be a useful model for software development. Special care must be taken when we try to adapt AOP to strongly typed functional languages which come with features like type inference mechanism, polymorphic types, higher-order functions and *type-scoped* pointcuts. Specifically, it is highly desirable that weaving of aspect-oriented functional programs can be performed statically and coherently. In [13], we showed a type-directed weaver which resolves all advice chainings coherently at static time. The novelty of this paper lies in the extended framework which supports static and coherent weaving in the presence of polymorphic recursive functions, advising advice bodies and higher-order advices.

1. Introduction

Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut the components of a software system [7]. In AOP, a program consists of many functional modules and some *aspects* that encapsulate crosscutting concerns. An aspect provides two specifications: A *pointcut*, comprising a set of functions, designates when and where to crosscut other modules; and an *advice*, which is a piece of code, that will be executed when a pointcut is reached. The complete program behavior is derived by some novel ways of composing functional modules and aspects according to the specifications given within the aspects. This is called *weaving* in AOP. Weaving results in the behavior of those functional modules impacted by aspects being modified accordingly.

Two highly desirable properties of weaving are it being *static* and *coherent*. Static weaving refers to making as many weaving decisions at compilation time as possible, usually by static translation to a "less-aspect-oriented" program. A direct benefit out of static weaving is that less run-time checking overhead is required. In addition, a weaver should allow different invocations of a function with inputs of the same type to be advised with the same set of advices. This property is known as coherence. Coherent weaving is also crucial as it ensures correct and understandable behavior of programmes. However, it is far from straightforward to bring these two properties together particularly under a strongly typed functional language setting. Let's consider a small example to have a feel of the intricacy involved.

FOAL'06

Example 1

```
n1@advice around {h} (arg::Int) = proceed (arg+1) in
n2@advice around {h} (arg) = proceed arg in
let h x = x in
let f x = h x in
(f 1) + (h 2)
```

This piece of code defines two pieces of advice namely `n1` and `n2`; it also defines a main program consisting of declarations of `f` and `h` and a main expression specifying applications of `f` and `h`. The first advice, `n1`, designates execution of `h` as its pointcut. It also contains a type constraint, which is called a *type scope*, attached to the first argument. `n1` is only triggered when `h` is executed with an Int argument. On the other hand, the pointcut of `n2` is not constrained by a type-scope. Thus all executions of function `h` match the pointcut. Consequently, pointcuts of `n1` and `n2` overlap in that the former is subsumed by the latter. In general, it is not possible to determine locally if a particular advice should be triggered. Let's consider the main program of the above example.

From a syntactic viewpoint, function `h` will be called in the body of `f`. If we naively infer that the argument `x` to function `h` in the RHS of `f`'s definition is of polymorphic type, we will be tempted to conclude that (1) advice `n2` should be triggered at the call, and (2) advice `n1` should not be called as its type scope is less general than $\forall a.a \rightarrow a$. As a result, `n2` will be statically chained to the call to `h`.

Unfortunately, this approach will cause incoherence behavior of `h` at run-time. Specifically, in the main expression, `(h 2)` will trigger both advices `n1` and `n2`. On the other hand, `(f 1)` in the main expression will actually pass integer argument `1` to `h`. There, triggering of `n1` is missed out since the weaver has mistakenly committed its choice in the definition of `f`. The only coherent behavior of a weaver in this case is to have `h` being advised by both `n1` and `n2`, during both invocations of `h`, *i.e.*, `(h 1)` and `(h 2)`.

It appears that the goals of achieving static weaving while ensuring coherent weaving are not in tandem here. In PolyAML [4], dynamic type checking is employed to handle matching of type-scoped pointcuts; on the other hand, Aspectual Caml [9] takes a syntactic approach which sacrifices coherence for static weaving.

In our earlier work [13], we designed a static weaving strategy that smoothly incorporates essential features of aspects into a core functional language with parametric polymorphism and higher-order functions. In contrast with the work done on PolyAML and Aspectual Caml, our strategy synthesizes functional core and aspects during compilation, thus successfully reconciling the desires to be static and to be coherent. The central idea there is to make full advantage of type information, both from the base program and

the type-scoped pointcuts, to guide the weaving of aspects. Specifically, it advocates a source-level type inference system for a higher-order, polymorphic language coupled with type-scoped pointcuts. A type-directed translation scheme is then devised to resolve all advice applications, thus eliminating any future need for dynamic type checking. The translation removes advice declarations from source programs and produces translated codes in an intermediate language which is essentially polymorphically typed lambda calculus with a small extension. The program in example 1 is translated as follows.

```
let n1 = \arg -> proceed (arg+1) in
let n2 = \arg -> proceed arg in
let h x = x in
let f dh x = dh x in
(f <h,{n1,n2}> 1) + (<h,{n1,n2}> 2)
```

Note that all advice declarations are translated into functions and are woven in. A special syntax $\langle \_ , \{\ldots\} \rangle$ is used to chain together advices and advised functions. For instance, $\langle$`h` , $\{$`n1`, `n2`$\}\rangle$ denotes the chaining of advices `n1` and `n2` to advised function `h`. In the above example, the two invocations of `h` in the original aspect program have been translated to an invocation of the chained function $\langle$`h` , $\{$`n1`, `n2`$\}\rangle$. This shows that our weaver respects the coherence property.

This coherent weaving of advices to `h` entails passing appropriate chained expressions of `h` to those function calls in the program text from which `h` may be called indirectly. This requirement is satisfied by allowing functions of those affected calls to carry extra parameters. In the code above, the translated definition of function `f` carries such an additional parameter, `dh`. The original `(f 1)` call is then translated to `(f <h,n1,n2> 1)`, in which the chained expression for `h` is passed.

In this paper, we re-engineer our type system and translation scheme to handle recursive functions, advising advice bodies and higher-order advices; this gives full-fledged support of aspects.[1] Previously, functions and advices in our framework were treated very differently. In particular, advices cannot be the targets of advising, neither directly by another advice nor indirectly through calls to advised functions in advice bodies. While completely blurring the distinction between functions and advices is not desirable, maintaining an unnecessary wide gap between them can also make aspect programming overly restrictive. As well argued by Rajan et al [11], two-layered models of advice and function cannot provide proper modularization for higher-order crosscutting concerns. Therefore, we refine our framework by devising new typing and translation rules that handle both advising advice bodies and higher-order advices, i.e., advices advising other advices.

The main contributions of this paper are:

- A translation scheme that enables *static and recursive weaving* of advices into recursive function definitions.
- A set of novel type rules with the support of an intermediate language that ensure static and coherent weaving of advanced aspect-oriented features. Specifically,
  - The weaving of *advices into other advices' bodies*. Static and coherent weaving of such advices has been challenging because the decision for weaving is only known after the context of invoking the underlying advice is known.
  - The weaving of *higher-order advices*. These are advices that advise other named advices. Such feature demands a uniform typing and translation scheme that not only infer the types of both functions and advices consistently but also weave in proper advices in a cascading manner according the type context.

The outline of the paper is as follows: Section 2 describes an aspect-oriented language and provides background information and terminologies used. In Section 3, we describe the intermediate language to be used as target of type-directed weaving. Section 4 describes our type-directed weaving algorithm, and presents our solutions to the handling of giving advice to both recursive functions and other advices, and the handling of higher-order advices. Section 5 surveys related work done in this field, and Section 6 concludes our work.

2. Aspect Language

In this section, we introduce an aspect-oriented functional language for our investigation. We shall focus on only some essential features of aspects, namely, *around* advice with *execution* pointcuts. Note that we drop the description of some features discussed previously in [13], namely higher-order functions, curried pointcuts and *any* pointcuts, as they are orthogonal to the discussion of this paper. However, it should be understood that they are still safely supported by the new system. The following syntax specifies the language.

Expressions	e	::=	$x \mid \lambda x.e \mid e\ e \mid$
			$\mathsf{let}\ f\ =\ e\ \mathsf{in}\ e \mid proceed \mid$
			$n@\mathsf{advice\ around}\ pc = e\ \mathsf{in}\ e$
Arguments	arg	::=	$x \mid x :: t$
Pointcuts	pc	::=	$\{\overline{jp}\}\ (arg)$
Joinpoints	jp	::=	$f \mid n$

We write $\bar{o}$ as an abbreviation for a sequence of objects $o_1, ..., o_n$ (e.g. expressions, types etc). Note that we generally assume $\bar{o}$ and o denotes non-related objects which should not be confused. The term $[\overline{o/a}]o'$ denotes simultaneous substitution of o_i for variables a_i in o', for $i = 1, \ldots, n$. We write $t_1 \sim t_2$ to specify equality between two types t_1 and t_2 (*a.k.a*, unification) to avoid confusing with assignment $=$. We write $fv(o)$ to denote the free variables in some object o.

For simplicity, we leave out type annotations, user defined data types, `if` statements and patterns but may make use of them in examples. Basic types such as booleans, integers, tuples and lists are predefined and their constructors are recorded in some initial environment.

In our language, an aspect is an advice declaration which includes a piece of advice and its target *pointcut*. Pointcuts are represented by $\{\overline{jp}\}\ (arg)$ where jp stands for joinpoints, comprising f, ranging over functions, and n, ranging over advices. A pointcut describes the point in time any function or advice from the set is executed. Usually function names are included in the pointcuts to designate the target functions for advice weaving. Since advices are also named, we allow advices advising other advices, i.e., higher-order advices. The argument variable *arg* is bound to the actual argument of the function execution and it may contain an annotated type.

Advice is a function-like expression that executes before, after, or around a pointcut. Note that *around* advice executes in place of the indicated pointcut, allowing a function to be replaced. A special function *proceed* may be called inside the body of an *around* advice. It is bound to a function that represents the rest of the computation at the advised pointcut. It is easy to see that both *before* advice and *after* advice can be simulated by *around* advice that always *proceeds*. Therefore, our aspect language only needs to support *around* advice.

[1] Previously supported features such as higher-order functions, curried pointcuts and any pointcut are compatibly supported in the new system even though a discussion of them are omitted from this paper.

There are two things about our pointcuts that merit further discussion. Firstly, the pointcut designator `around pc` represents the point in time when the functions or advices in `pc` are about to execute. Like the *execution* pointcuts of AspectJ, these pointcuts cover the cases when functions are explicitly invoked as well as those when they are implicitly called. They are necessary for languages with functions as first-class values, for, in such languages, functions can be applied directly through name-based invocation as well as indirectly through aliasing and functional arguments to a higher-order function. The following simple program illustrates the situations.

```
n@advice around {f} (arg) = e in
let f x = x in
let g = f in
let h k x = k x in
(f True, g 'a', h g 3)
```

Clearly, in this example, if we look for only the function calls made to `f`, following the call pointcuts of AspectJ, we will not be able to capture the applications of `f` through `g` and `k`. However, deriving a static weaving scheme for advices on execution pointcuts in a statically typed functional language is not as easy as it may appear. In AspectJ, the pointcut designator, `execution f (arg)`, will direct the weaver to insert the advice call into the body of `f`. As a result, the invocations of `f` through `g` and `k` will also trigger the advice. However, this naive approach will encounter great static typing difficulties when handling advices with additional type constraints, which is strongly related to the following discussion on type-scoped advices.

Secondly, as well demonstrated in [4] and [9], it is very often that we need to have advices with type constraints to confine the applicable scope of such advices. Our aspect language support such advices as it allows type constraints to be imposed on the arguments of those functions occurring in pointcuts. We call such pointcuts *type-scoped pointcuts*. Advices with type scoped pointcuts are henceforth called *type-scoped advices*. However, having such type-scoped advices in a statically typed language will pose significant challenges for advice weaving, since it calls for a smooth reconciliation between type-based advice dispatch and static weaving. In our opinions, previous work did not address this issue adequately. In designing Aspectual Caml, Masuhara et al. also suggest using execution pointcuts to handle indirect function calls. But they followed the weaving scheme of AspectJ by inserting a call to the associated advice in the advised function. Apparently, this scheme will only work for monomorphic functions; dynamic type-dispatch is needed to support polymorphic yet type-scoped advices. This may also partly explain why Dantas et al. include runtime type analysis mechanism in their design of PolyAML. By contrast, our aspect language supports type-scoped advices while retaining both static typing and static weaving.

The following syntax defines the type expressions in our aspect language.

Types	t	$::=$	$a \mid t \rightarrow t$
Type Schemes	σ	$::=$	$\forall \bar{a}.\rho$
Advised Types	ρ	$::=$	$(x:t).\rho \mid t$

Basic types such as booleans, integers, tuples and lists are predefined and their constructors are recorded in some initial environment. Central to our approach is the construct of *advised types*, inspired by the *predicated types* [12] used in Haskell's type classes. These advised types augment common type schemes with *advice predicates*, which are used to capture the need of future advice weaving dependent on the type context. For example, the type scheme for the function *g* in the above example will be $\forall a.(f : a \rightarrow a).a \rightarrow a$, which indicates that whenever *g* is applied in a specific context, the advices on *f* will also be triggered. We shall explain them in detail in Section 4.

In the next a few (sub)sections, we show how the features discussed in this paper are used when programming with aspects. The challenges in incorporating them into a static and coherent weaving framework are also outlined.

2.1 Recursive Functions

Recursive functions are widely used in functional programming. When type-scope advices are defined on a polymorphic recursive function, it may yield an interesting advised type which has a predicate refers the function itself. Let's consider a small example for illustration.

Example 2

```
let g x = x + 1 in
n@advice around {f} (arg:[Int])
  = Cons (g (head arg)) (proceed arg) in
let f x = if (length x) > 0 then f (tail x) else x
in f [1,2,3]
```

The function `f` above defines a generic traversal of an input list. When the input list contains *Int* elements, advice `n` intercepts the execution and applies function `g` to the list head. Thus, it simulates the behavior of the standard *map* function.

In an AO system which performs weaving by static translation, the definition of function `f` should be translated into an expression with relevant advices chained. However, because of the recursion, the translation of `f` requires a translated definition of itself which results into a cyclic process!

A syntactic weaver may sees this matter from a syntactic view by chaining advices into the type-annotated abstract syntax tree. In that case, coherence is lost as the execution of the recursive calls may be chained with a different set of advices other than those of the initial call even when the recursion is monomorphic.

Let's consider the program in Example 2. The typed AST annotates the initial `f`-call in the main expression with type $[Int] \rightarrow [Int]$ and the recursive `f`-calls in the definition of `f` with type $[a] \rightarrow [a]$. Thus, the former is chained with advise `n` whereas the later is not even though both receive arguments of type $[Int] \rightarrow [Int]$ during the actual execution.

In Section 4, we show how a fixed point combinator can be employed to achieve coherent weaving of recursive functions.

2.2 Nested Advices

Aspects are not limited to observing base programs. Inside the bodies of advice definitions, there may be calls to other functions that are advised. We call these *nested advices*.

The program in example 2 increments all the elements of a list by one through the interception of aspect `n`. However, when `f` is called with the empty list `[]`, the program crashes as the advice attempts to extract the head from `[]` before the test of list length in the body of `f` is performed.

To remedy this safety violation, we may implement a patch aspect by setting the head of `[]` to an invalid bit, say -1.

```
n1@advice around {head} (arg:[Int])
   = if arg == [] then -1 else proceed arg
```

Note that advice `n1` advises on a function called inside the body of an advice. In another words, `n1` is a nested advice.

This small example sheds light on a wide range of applications of nested advices. When aspects are used for enforcing safety and security concerns, it is important that the advices are applied to every execution of the target functions. Therefore, nested advices becomes the essential feature which supports this behavior.

Note that we do not allow circular *around* advices that apply to the execution of their own bodies, directly or indirectly. The reason for this restriction is that circular *around* advices together with potential recursive functions that they are advising may form a scenario similar to polymorphic mutual recursion which threatens decidability of type inference. We leave this for future investigation.

Even without circular *around* advices, weaving nested advices statically is a challenging task primary for the following two reasons.

1. Advice chainings only appear in the woven program which is not a subject for further weaving. A syntactic approach to solve this problem is to have an iterative process which repetitively feeds the woven program back to the weaver until no more advice can be woven. One side condition for this approach is that both input and output of weaver are from the same language.
2. The typing context where an advice `n` is chained may not be sufficiently specific for another advice to be chained to the calls inside `n`'s body. This complicates coherent weaving.

In Section 4, we show in details how our translation works coherently without the need of iteratively feeding the woven program back to the weaver.

2.3 Higher-Order Advices

The two-layered design of AspectJ like languages only allow advices to advise other advices in a very restricted way. The loss of expressiveness of such an approach has been well argued in [11]. The idea of a multi-layered design dates back to [5, 1, 10]; and this is sometimes called *higher-order advices*.

In Section 2.2, we use a piece nested advice to patch an unsafe program. However, the result is not completely satisfactory as `n1` always inserts an extra invalid bit into the result list. The root of the problem is the inability of advising an advise directly. In this section, we show a solution with a higher-order advice which cause no undesirable side effects such as the extra bit.

```
n2@advice around {n} (arg)
   = if arg == [] then [] else proceed arg
```

Advice `n2` advise directly on `n` which allows us to short circuit the `head`-call when the input is `[]`.

There has been some argument that higher-order advices can be simulated by nested-advices. Take AspectJ as an example. Advices are nameless in AspectJ, hence we cannot directly advise another advice. Instead, if we know there are such requirements in advance, we can shape the target advice for advice nesting as follows: Move its entire advice body into a help method, and write a piece of advice that advise this help method, thus achieving the effect of advising advices to a certain degree. But there are at least two shortcomings using this way of simulation. Firstly, this only works for *before* and *after* advices, because *proceed* will take effect only when it occurs inside an around advice. Thus, the example given above cannot be handled. Secondly. This scheme does not scale up well. What if later we want yet another third-order advice on the second-order one?

Besides being used as patches for other advices, higher-order advices are also useful as development aspects. Let's say we want to compute the total amount of a customer order and apply discount rates according to certain regular rules as follows.

Example 3

```
let calcPrice cart = sum (map discount cart) in
let discount item = (getRate item) * (getPrice item)
```

In addition to regular discount rules, there are also other ad-hoc sale discounts that may be put into effect on certain occasions, such as special holiday-sales, anniversary-sales, etc. Due to their ad-hoc nature, it is better to separate them from the functional modules and put them in aspects that advise on the discount rate query function.

```
n1@advice around {getRate} (arg) =
   (getHolidayRate arg) * (proceed arg)
n2@advice around {getRate} (arg) =
   (getAnniversaryRate arg) * (proceed arg)
```

Furthermore, it is common to have some business rules that govern all the sales promotions offered to customers. For example, there may be a rule stipulates the maximum discount rate that is applicable to any product item, regardless of the multiple discounts it qualifies. Such business rules can be realized using aspects of higher-order in a modular manner.

```
n3@advice around {n1,n2} (arg) =
   let finalRate = proceed arg
   in  if (finalRate < 0.5) then 0.5 else finalRate
```

Here the second-order advice `n3` has meta-control over advices `n1` and `n2`. The call to *proceed* gets the compounded discount rate and the rule that no products can be sold under 50% of their list prices is applied.

Weaving higher-order advices involves allowing advices to be advised as functions. This adds in another layer of complexity to the translation. Again, we refer the readers to Section 4 for a detailed discussion of the solution.

3. Intermediate Language

Our type-directed weaving produces codes in an intermediate language, which explicitly expresses the chaining of advices. The intermediate language is based on a polymorphically typed lambda calculus plus let introductions, extended with chaining expressions which are used to model advice invocations triggered by function calls.

3.1 Operational Semantics

The syntax of the language is displayed below.

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & \lambda x.e \mid \mu f.e \mid \langle v, \{\bar{v}\}\rangle \\
\text{Expressions} & e & ::= & v \mid x \mid \mathit{proceed} \mid \lambda x.e \mid e\ e \mid \langle e, \{\bar{e}\}\rangle \\
 & & & \mid \mathsf{let}\ f\ =\ e\ \mathsf{in}\ e
\end{array}
$$

There is no notion of advices in this language as they are modelled straightforwardly as functions. Pointcuts are also not necessary since we assumes all advices are already woven in place. A chaining expression of the form $\langle e, \{\bar{e}\}\rangle$ consists of an expression e which evaluates to a function (or an advice) and a chain of expressions $\bar{e}$ which evaluates to advices to be triggered by the function call. We call e occurring at the left component the *active* function/advice, and $\{\bar{e}\}$ the *dormant* advices. When both e and $\bar{e}$ are values, the chaining expression is itself a value.

The set of β reductions are defined as follow:

$$
\begin{array}{lcl}
(\lambda x.e\ v) & \longmapsto_\beta & (e[v/x]) \\
(\mu f.e\ v) & \longmapsto_\beta & (e[\mu f.e/f]\ v) \\
(\mathsf{let}\ x = v\ \mathsf{in}\ e) & \longmapsto_\beta & (e[v/x]) \\
(\langle v, \{\}\rangle\ v') & \longmapsto_\beta & (v\ v') \\
(\langle v, \{v_1, \bar{v}\}\rangle\ v') & \longmapsto_\beta & (v_1[\langle v, \{\bar{v}\}\rangle/\mathit{proceed}]\ v')
\end{array}
$$

These rules specifies a call-by-value evaluation strategy which is orthogonal to the language design. The first three rules are standard β-rules for lambda calculus. In the fourth rule, when the advice sequence is empty, the chaining returns the original function. Otherwise, as shown in the last rule, the chaining replaces the *proceed* in the first advice in sequence by a value which chains the function v with the remainder of the advice sequence.

The substitution operation $e[v/x]$ performs the usual substitution of values for variables, with one exception: When the variable being substituted is *proceed*, and the expression is a chained expression, then the corresponding substitution is performed only on the active expression, but not the dormant expressions:

$$\langle e, \{\bar{e}\}\rangle[v/proceed] \equiv \langle e[v/proceed], \{\bar{e}\}\rangle.$$

Note that the dormant advices above, $\{\bar{e}\}$, have not been substituted, because the *proceed* is bound to the existing active expression e, not the dormant expression $\bar{e}$. This point is particularly important in the case of second-order advice, which will have different *proceed* value from the advice which the former is advising.

3.2 Type System

Programs produced in the intermediate language first undergo α-conversion. This frees the programs from scoping concerns. Consequently, the program can be type-checked for its correctness. The type system is defined in Figure 1.

$$(\textsc{Var})\quad \frac{x:\sigma \in \Gamma}{\Gamma \vdash_i x:\sigma} \qquad (\textsc{Chain})\quad \frac{\Gamma, proceed:t' \vdash_i \bar{e}:\bar{t} \quad \Gamma \vdash_i e:t' \quad t' \trianglelefteq \bar{t}}{\Gamma \vdash_i \langle e, \{\bar{e}\}\rangle : t'}$$

$$(\textsc{Abs})\quad \frac{\Gamma, x:t_1 \vdash_i e:t_2}{\Gamma \vdash_i \lambda x.e : t_1 \to t_2} \qquad (\textsc{Fix})\quad \frac{\Gamma, f:t \vdash_i e:t}{\Gamma \vdash_i \mu f.e : t}$$

$$(\forall\textsc{Elim})\quad \frac{\Gamma \vdash_i e:\forall a.\sigma}{\Gamma \vdash_i e:[t/a]\sigma} \qquad (\textsc{App})\quad \frac{\Gamma \vdash_i e_1 : t_1 \to t_2 \quad \Gamma \vdash_i e_2 : t_2}{\Gamma \vdash_i e_1\, e_2 : t_2}$$

$$(\forall\textsc{Intro})\quad \frac{\Gamma \vdash_i e:\sigma \quad a \notin \Gamma}{\Gamma \vdash_i e:\forall a.\sigma}$$

$$(\textsc{Let})\quad \frac{\Gamma, proceed:t', f:\sigma \vdash_i e_1:\sigma \quad \Gamma, f:\sigma \vdash_i e_2:t \quad \sigma \trianglerighteq t'}{\Gamma \vdash_i \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t}$$

Figure 1. Typing Rules

There is no introduction of new type syntax other than the one of the standard polymorphically typed lambda calculus.

$$\begin{array}{lllll} \text{Types} & t & ::= & a \mid t \to t \\ \text{Type Schemes} & \sigma & ::= & \forall \bar{a}.t \end{array}$$

The typing rules are presented in Figure 1 which are mostly standard except that type bindings of *proceed* is needed for function definitions introduced by let. The reason for this is that advices, which may contain *proceed*-calls, appear as functions in the intermediate language. In Rule (CHAIN), the advices are typed under the assumption that *proceed* is an instance of the function. We also require the advices have types more general than that of the function. We say a type scheme is more general than the other if it can be instantiated to the latter via variable substitutions. The relation is formally defined as:

$$(\textsc{Gen})\quad \frac{[\bar{t}/\bar{a}]t_1 \sim t_2}{\forall \bar{a}.t_1 \trianglerighteq \forall \bar{b}.t_2}$$

The type system enjoys the standard safety properties.

Theorem 1 (Progress) *If $\vdash_i e : \sigma$, then either e is a value or else there is some e' with $e \longmapsto_\beta e'$.*

Theorem 2 (Preservation) *If $\Gamma \vdash_i e : \sigma$ and $e \longmapsto_\beta e'$, then $\Gamma \vdash_i e' : \sigma$.*

4. Type Directed Weaving

As introduced in Section 2, $advised\ type$ denoted as ρ is used to capture function names and their types that may be required for advice resolution. For instance, in the main program given in Example 1, function `f` possesses the advised type $\forall a.(h : (a \to a)).a \to a$, in which $(h : a \to a)$ is called an *advice predicate*. It signifies that *the execution of any application of* `f` *may require advices of* `h` *applied with type which should be no more general than* $a \to a$.

Note that advised types are used to indicate the existence of some *indeterminate advices*. If a function contains only applications whose advices are completely determined, then the function will not be associated with an advised type; it will be associated with a normal (and possibly polymorphic) type. As an example, the type of the advised function `h` in Example 1 is $\forall a.a \to a$ since it does not contain any applications of advised functions in its definition.

$$(\textsc{AErase})\quad [\![\forall \bar{a}.(x:t).\rho]\!] = [\![\forall \bar{a}.\rho]\!] \quad [\![\forall \bar{a}.(x:t).t']\!] = \forall \bar{a}.t'$$

$$(\textsc{Gen}_F)\quad gen(\Gamma, \sigma) = \forall \bar{a}.\sigma \quad \text{where } \bar{a} = fv(\sigma) \backslash fv(\Gamma)$$

$$(\textsc{Card})\quad |o_1 ... o_k| = k \qquad (\textsc{Card}_p)\quad |\forall \bar{a}.\bar{p}.t|_{pred} = |\bar{p}|$$

Figure 2. Auxiliary Definitions

Figure 2 defines a set of auxiliary functions/relations that assists type inference. The letter t ranges over unification (type-)variables which are distinct from quantified rigid type variable a. Rule (AERASE) defines a function $[\![\cdot]\!]$ which removes all advice predicates from an advised type scheme. We also define, in rule ($\textsc{Gen}_F$), a generalization procedure which turns a type into a type scheme by quantifying type variables that do not appear free in the type environment. The (CARD) function, denoted by $|\cdot|$, returns the cardinality of a sequence of objects. The ($\textsc{Card}_p$) function returns the number of advice predicates in a type scheme.

The main set of type inference rules, as described in Figure 3, is an extension to the Hindley-Milner system. We introduce a judgment $\Gamma \vdash e : \sigma \rightsquigarrow e'$ to denote that expression e has type σ under type environment Γ and it is translated to e'. We assume that the advice declarations are preprocessed and all the names which appear in any of the pointcuts are recorded in an initial global store A. We also assume that the base program is well typed in Hindley-Milner and the type information of all the functions are stored in Γ_{base}.

The typing environment Γ contains not only the usual type bindings (of the form $x : \sigma \rightsquigarrow e$) but also *advice bindings* of the form $n : \sigma \bowtie \bar{x}$. This states that an advice with name n of type σ is defined on $\bar{x}$. We may drop the $\bowtie \bar{x}$ part when it is not relevant. When the bound variable is advised (i.e. $x \in A$), we use a different binding $:_*$ to distinguish from the non-advised case. We also use the notation $:_{(*)}$ to represent a binding which is either : or $:_*$.

Note that while it is possible to present the typing rules without the translation detail by simply deleting the '$\rightsquigarrow e$' portion, it is not possible to present the translation rules independently since typing controls the translation.

4.1 Predicating and Releasing

There are two rules for variable lookups. Rule (VAR) is standard. In the case that variable x is advised, rule (VAR-A) will check all advices defined on x (we do not distinguish : and $:_*$-binding for

$$\text{(VAR)}\quad \frac{x : \sigma \rightsquigarrow e \in \Gamma}{\Gamma \vdash x : \sigma \rightsquigarrow e} \qquad \text{(VAR-A)}\quad \frac{\begin{array}{c} x :_* \sigma_x \in \Gamma \quad [\![\bar{\sigma}]\!] \not\trianglelefteq [\![\sigma']\!] \quad \Gamma \vdash n_i : [\![\sigma']\!] \rightsquigarrow e_i \\ \bar{n} :_{(*)} \overline{\sigma} \bowtie x \rightsquigarrow \bar{n}' \in \Gamma \quad \{n_i \mid [\![\sigma_i]\!] \trianglerighteq [\![\sigma']\!]\} \\ |\bar{y}| = |\sigma_x|_{pred} \quad \bar{y}\ is\ fresh \quad \sigma_x \trianglerighteq \sigma' \end{array}}{\Gamma \vdash x : \sigma' \rightsquigarrow \lambda \bar{y}.\langle x\ \bar{y}\ , \{e_i\}\rangle}$$

$$\text{(}\forall\text{ELIM)}\quad \frac{\Gamma \vdash e : \forall a.\sigma \rightsquigarrow e'}{\Gamma \vdash e : [t/a]\sigma \rightsquigarrow e'} \qquad \text{(}\forall\text{INTRO)}\quad \frac{\Gamma \vdash e : \sigma \rightsquigarrow e' \quad a \notin \Gamma}{\Gamma \vdash e : \forall a.\sigma \rightsquigarrow e'} \qquad \text{(APP)}\quad \frac{\begin{array}{c} \Gamma \vdash e_1 : t_1 \to t_2 \rightsquigarrow e_1' \\ \Gamma \vdash e_2 : t_1 \rightsquigarrow e_2' \end{array}}{\Gamma \vdash e_1\ e_2 : t_2 \rightsquigarrow (e_1'\ e_2')}$$

$$\text{(ABS)}\quad \frac{\Gamma, x : t_1 \rightsquigarrow x \vdash e : t_2 \rightsquigarrow e'}{\Gamma \vdash \lambda x.e : t_1 \to t_2 \rightsquigarrow \lambda x.e'} \qquad \text{(LET)}\quad \frac{\begin{array}{c} \Gamma \vdash e_1 : \sigma \rightsquigarrow e_1' \\ \Gamma, f :_{(*)} \sigma \rightsquigarrow f \vdash e_2 : t \rightsquigarrow e_2' \end{array}}{\Gamma \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t \rightsquigarrow \mathsf{let}\ f = e_1'\ \mathsf{in}\ e_2'}$$

$$\text{(PRED)}\quad \frac{\begin{array}{c} x :_* \sigma_x \in \Gamma \quad t \trianglelefteq [\![\sigma_x]\!] \\ \Gamma, x : t \rightsquigarrow x_t \vdash e : \rho \rightsquigarrow e_t' \quad x \in A \end{array}}{\Gamma \vdash e : (x : t).\rho \rightsquigarrow \lambda x_t.e_t'} \qquad \text{(REL)}\quad \frac{\begin{array}{c} \Gamma \vdash e : (x : t).\rho \rightsquigarrow e' \\ \Gamma \vdash x : t \rightsquigarrow e'' \quad x \in A \quad x \neq e \end{array}}{\Gamma \vdash e : \rho \rightsquigarrow e'\ e''}$$

$$\text{(FIX)}\quad \frac{\Gamma, f :_{(*)} \rho \rightsquigarrow f \vdash e : \rho \rightsquigarrow e'}{\Gamma \vdash \mu f.e : \rho \rightsquigarrow e'} \qquad \text{(REL-F)}\quad \frac{\Gamma \vdash f : (f : t).\rho \rightsquigarrow e' \quad F\ fresh \quad f \in A}{\Gamma \vdash f : \rho \rightsquigarrow \mathsf{let}\ F = (e'\ F)\ \mathsf{in}\ F}$$

$$\text{(ADV)}\quad \frac{\begin{array}{c} \Gamma, proceed : t \vdash \lambda x.e_a : \bar{p}.t \rightsquigarrow e_a' \quad f_i : \sigma' \in \Gamma_{base} \\ \sigma' \trianglelefteq [\![\sigma]\!] \quad \Gamma, n :_{(*)} \sigma \bowtie \bar{f} \rightsquigarrow n \vdash e : t' \rightsquigarrow e' \quad \sigma = gen(\Gamma, \bar{p}.t) \end{array}}{\Gamma \vdash n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x) = e_a \mathsf{in}\ e : t' \rightsquigarrow \mathsf{let}\ n = e_a'\ \mathsf{in}\ e'}$$

$$\text{(ADV-AN)}\quad \frac{\begin{array}{c} \Gamma, proceed : t \vdash \lambda x : t_x.e_a : \bar{p}.t \rightsquigarrow e_a' \quad f_i : \forall \bar{a}.t_i \to t_i' \in \Gamma_{base} \\ t_x \trianglelefteq \forall \bar{a}.t_i \quad (t_i \to t_i') \sim t \quad \Gamma, n :_{(*)} \sigma \bowtie \bar{f} \rightsquigarrow n \vdash e : t' \rightsquigarrow e' \quad \sigma = gen(\Gamma, \bar{p}.t) \end{array}}{\Gamma \vdash n@\mathsf{advice\ around}\ \{\bar{f}\}\ (x :: t_x) = e_a \mathsf{in}\ e : t' \rightsquigarrow \mathsf{let}\ n = e_a'\ \mathsf{in}\ e'}$$

Figure 3. Type-directed Weaving by translation

these advices here) to see whether any of them has a more specific type than x's. This is to ensure that chaining of advices is only done in a sufficiently specific context. We call this check *sufficiently specific context check*, and it is expressed in the rule as the guard $[\![\bar{\sigma}]\!] \not\trianglelefteq [\![\sigma']\!]$ (the relation $\not\trianglelefteq$ is defined in Section 3.2). If the check succeeds (*i.e.*, no advice has a more specific type than x), x will be chained with the translated forms of all those advices defined on it, having the same or more general types than x has. We give all these selected advices a non-advised type in the translation of them $\Gamma \vdash n_i : [\![\sigma']\!] \rightsquigarrow e_i$. This ensures correct weaving of nested advices advising the bodies of the selected advices. The detail will be elaborated in Section 4.4. Finally, the final translated expression is *normalized* by bringing all the advice abstractions of x outside the chain $\langle \ldots \rangle$. This ensures type compatibility between the advised call and its advices as required by the type system of the intermediate language.

If the check for sufficiently specific context fails, there must exists some advices for x with more specific types, and rule (VAR-A) fails to apply. Since $x \in A$ still holds, rule (PRED) can be applied. This rule introduces an *advice parameter* to the program (through the corresponding translation scheme). This advice parameter enables concrete *advice-chained functions* to be passed in at a later stage, called *releasing*, through the application of rule (REL).

Before we describe rules (PRED) and (REL) in detail, we illustrate the application of these rules by derving the type and the woven code for the program shown in Example 1. During the derivation of the definition of f, we have:

$$\Gamma = \{\ h :_* \forall a.a \to a \rightsquigarrow h, n_2 : \forall a.a \to a \bowtie h, n_1 : I \to I \bowtie h\}$$

$$\text{(VAR)}\ \frac{h : t \to t \rightsquigarrow dh \in \Gamma_2}{\Gamma_2 \vdash h : t \to t \rightsquigarrow dh} \qquad \text{(VAR)}\ \frac{x : t \rightsquigarrow x \in \Gamma_2}{\Gamma_2 \vdash x : t \rightsquigarrow x}$$

$$\text{(APP)}\ \frac{\Gamma_2 \vdash h : t \to t \rightsquigarrow dh \qquad \Gamma_2 \vdash x : t \rightsquigarrow x}{\Gamma_2 = \Gamma_1, x : t \rightsquigarrow x \vdash (h\ x) : t \rightsquigarrow (dh\ x)}$$

$$\text{(ABS)}\ \frac{\Gamma_2 = \Gamma_1, x : t \rightsquigarrow x \vdash (h\ x) : t \rightsquigarrow (dh\ x)}{\Gamma_1 = \Gamma, h : t \to t \rightsquigarrow dh \vdash \lambda x.(h\ x) : t \to t \rightsquigarrow \lambda x.(dh\ x)}$$

$$\text{(PRED)}\ \frac{\Gamma_1 = \Gamma, h : t \to t \rightsquigarrow dh \vdash \lambda x.(h\ x) : t \to t \rightsquigarrow \lambda x.(dh\ x)}{\Gamma \vdash \lambda x.(h\ x) : (h : t \to t).t \to t \rightsquigarrow \lambda dh.\lambda x.(dh\ x)}$$

Next, for the derivation of the main expression, we have:

$$\Gamma_3 = \{\ h :_* \forall a.a \to a \rightsquigarrow h, n_2 : \forall a.a \to a \bowtie h,\ n_1 : I \to I \bowtie h, f : \forall a.(h : a \to a).a \to a \rightsquigarrow f\}$$

$$\text{(VAR)}\ \frac{f : \forall a.(h : a \to a).a \to a \rightsquigarrow f \in \Gamma_3}{\Gamma_3 \vdash f : (h : I \to I).I \to I \rightsquigarrow f}\ \textcircled{a} \qquad \text{(REL)}\ \frac{\Gamma_3 \vdash f : (h : I \to I).I \to I \rightsquigarrow f}{\Gamma_3 \vdash f : I \to I \rightsquigarrow (f\ \langle h\ , \{n_1, n_2\}\rangle)} \qquad \ldots$$

$$\text{(APP)}\ \frac{\Gamma_3 \vdash f : I \to I \rightsquigarrow (f\ \langle h\ , \{n_1, n_2\}\rangle) \qquad \ldots}{\Gamma_3 \vdash (f\ 1) : I \rightsquigarrow (f\ \langle h\ , \{n_1, n_2\}\rangle\ 1)}$$

$$\textcircled{a} = \text{(VAR-A)}\ \frac{h :_* \forall a.a \to a \rightsquigarrow h \in \Gamma_3 \quad \ldots}{\Gamma_3 \vdash h : I \to I \rightsquigarrow \langle h\ , \{n_1, n_2\}\rangle}$$

We note that rules (ABS),(LET), (APP), (∀INTRO) and (∀ELIM) are rather standard, with the tiny exception that rule (LET) will bind f with : when it is not in A; and with $:_*$ otherwise.

Rules (PRED) and (REL) respectively introduces and eliminates advice predicates just as (∀INTRO) and (∀ELIM) do to bound type variables. Rule (PRED) adds an advice predicate to a type (Note that we only allow sensible choices of t constrained by $t \trianglelefteq [\![\sigma_x]\!]$). Correspondingly, its translation yields a lambda abstraction with an advice parameter. At a later stage, rule (REL) is applied to release (*i.e.*,remove) an advice predicate from a type. Its translation

generates a function application with an advised expression as argument.

4.2 Advising Recursive Functions

Now let's consider Example 2 given in Section 2.1 where the advised function `f` is recursive. The code is reproduced below.

```
let g x = x + 1 in
n@advice around {f} (arg:[Int])
  = Cons (g (head arg)) (proceed arg) in
let f x = if (length x) > 0 then f (tail x) else x
in f [1,2,3]
```

In our type system, rule (FIX) is used to type and translate recursive functions. In this above example, our translation produces an interesting advised type $\forall a.(f : [a] \rightarrow [a]).[a] \rightarrow [a]$ for `f`. If Rule (REL) is applied to release this type, the translation will not terminate as the derivation of $\Gamma \vdash f : [a] \rightarrow [a]$ depends on itself. The solution is to break the loop by using a fixed point combinator as the translation result. This is manifested in Rule (REL-F), by which example 2 is translated to the following:

```
let g x = x + 1 in
let n = \arg.(Cons (g (head arg)) (proceed arg)) in
let f df x = if (length x) > 0
             then df (tail x) else x in
(let F = \y.<f y,{n}> F in F) [1,2,3]
```

By a simple Let-lifting, we lift the local definition of `F` to the top level. The final translation result is:

```
let g x = x + 1 in
let n = \arg.(Cons (g (head arg)) (proceed arg)) in
let f df x = if (length x) > 0
             then df (tail x) else x in
let F = \y.<f y,{n}> F in
F [1,2,3]
```

The fixed point combinator `F` correctly captures the desired behavior by chaining every execution of `f` with `n`. In the following, we sketch the evaluation steps for the main expression `F [1,2,3]` based on the operational semantics given in Section 3.

For the sake of presentation, some long expressions are renamed as follows.

$v_1 = \backslash$`x.if (length x) > 0 then F (tail x) else x`
$v_2 = \backslash$`arg.(Cons (g (head arg)) (`v_1` arg))`

We also use $\longrightarrow_\beta{}^*$ to represent multiple steps of β reduction.

$$
\begin{array}{ll}
 & \texttt{F}\ [1,2,3] \\
\longrightarrow_\beta & (\backslash\texttt{y.}\langle \texttt{f y , \{n\}} \rangle\ \texttt{F})\ [1,2,3] \\
\longrightarrow_\beta & \langle \texttt{f F , \{n\}} \rangle\ [1,2,3] \\
\longrightarrow_\beta & \langle v_1 \texttt{ , \{n\}} \rangle\ [1,2,3] \\
\longrightarrow_\beta & v_2\ [1,2,3] \\
\longrightarrow_\beta{}^* & \texttt{Cons 2 (F [2,3])} \\
\ldots &
\end{array}
$$

4.3 Handling Advices

There are two type-inference rules for handling advices. Rule (ADV) handles non-type-scoped advices, whereas rule (ADV-AN) handles type-scoped advices. In rule (ADV), we firstly infer the (possibly advised) type of the advice as a function $\lambda x.e_a$ under the type environment extended with *proceed*. The advice body is therefore translated. Note that this translation does not necessarily complete all the chaining because the most specific context condition may not hold. In this case, just like functions, the advice is parameterized. At the same time, an advised type is assigned to it and only released when it is chained in Rule (VAR-A).

After type inference of the advice, we ensure that all functions in the pointcut have type schemes that are not more general than the advice's. Note that the type information of all the functions are stored in Γ_{base}. Then, this advice is added to the environment. It does not appear in the translated program, however, as it is translated into a function awaiting for participation in advice chaining.

In rule (ADV-AN), variable x can only be bound to a value of type t_x such that t_x is no more general than the input type of those functions in the pointcut. We also require the type of all functions in the pointcut to be unifiable to the advice type, so that any bogus advices which can never be safely triggered will be rejected by our type system.

Note that we do not allow the annotated type t_x to be more general than the input type of any function in the pointcut, as this will be contrary to the intention of type-scoped advices.

4.4 Advising Advice Bodies

As mentioned in the previous (sub)section, the Rules (ADV) and (ADV-AN) make an attempt to translate advice bodies. However, just like the translation of function bodies, the local type contexts may not be specific enough to chain all the advices. We illustrate this with an example.

Example 4

```
n1@advice around {f} (arg::Int) = e1 in
n2@advice around {f} (arg) = e2 in
let f x = x in
n3@advice around {g} (arg) = f arg in
let g x = x in
let h x = g x in
h 1
```

Here, advice `n3` calls `f` which is in turn being advised. The goal of our translation is to chain advices which are applicable to the call of `f` inside an advice. Concretely, when a call to `g` is chained with advice `n3`, the body of `n3` must also be advised. Moreover, the choice of advices must be coherent.

At the time when the declaration of `n3` is translated, the body of the advice is translated. An advised type is given to it since the currently context is not sufficiently specific.

When the translation attempts to chain an advice in Rule (VAR-A), the judgment $\Gamma \vdash n_i : [\![\sigma']\!] \rightsquigarrow e_i$ in the premise forces the advice to have a non-advised type. This is to ensure that all the advice abstractions are fully released so that chaining can take effect.

In the case that this derivation fails, it signifies that the current context is not sufficiently specific for advising some of the calls in this advice's body, and chaining has to be delayed. In example 4, the call to `g` in the body of `h`'s definition is of type $a \rightarrow a$. This is sufficiently specific for advising `g`, since `n3` is the only candidate. Consequently, the call to `f` inside the body of `n3` is also of type $a \rightarrow a$. However, this type is not sufficiently specific for advising `f`. As a result, we have to give `h` an advised type and it is translated as follows.

```
let n1 = \arg.e1 in
let n2 = \arg.e2 in
let f x = x in
let n3 = \df.\arg.df arg in
let g x = x in
let h dg x = dg x in
h <g,{n3 <f,{n1,n2}>}> 1
```

`n3` is only chained in the main expression where the context is sufficiently specific for both the calls to `g` and `f`.

4.5 Higher-Order Advices

In our system, we show that, just like functions, advices can be advised liberally. An example is given below.

Example 5

```
n1@advice around {f} (arg::Int) = e1 in
n2@advice around {n1} (arg::Int) = e2 in
let f x = x in
let g x = f x in
g 1
```

The second advice declaration is higher-order as it advises another advice `n1`. The advising mechanism in our language does not prejudice functions over advices. The translation $\Gamma \vdash n_i : [\![\sigma']\!] \rightsquigarrow e_i$ in the premise of Rule (VAR-A) not only translates bodies of advices but also chains n_i with advices defined on it.

In the premises of Rule (ADV) and (ADV-AN), we note that typing information of advices is not stored in Γ_{base}. Thus, we replace $f_i : \sigma' \in \Gamma_{base}$ by $n_i :_* \sigma' \in \Gamma$.[2] Consequently, the check $\sigma' \trianglelefteq [\![\sigma]\!]$ in (ADV) becomes $[\![\sigma']\!] \trianglelefteq [\![\sigma]\!]$ as σ' may be an advised type. By doing this, we assume advised advices are translated before the advices defined on them. This is valid because circular cases are precluded.

Thus, example 5 is translated into

```
let n1 = \arg.e1 in
let n2 = \arg.e2 in
let f x = x in
let g df x = df x in
g <f,{<n1,{n2}>}> 1
```

Note that advice `n1` is chained with `n2` before the chaining to `f`.

4.6 Correctness of Translation

One of the desirable properties of our type-directed weaving algorithm is its reliance on a type-inference system that is a conservative extension of the Hindley-Milner Type System. (Note that the notation $[\![\cdot]\!]$ is defined in Figure 2.)

Theorem 3 (Conservative Extension) *Given a program P consisting of a set of advices and a closed base program e. If*

$$\vdash P : \sigma \rightsquigarrow P',$$

then

$$\vdash e : [\![\sigma]\!].$$

Our main theorem is to ensure that our translated program preserves the type of the original program. When the original program is of an advised type, the translated scheme will concretize the advice predicates into advice parameters, which constitute part of the translated program. To this end, we define a function η that translates advised type to normal polymorphic type.

$$\begin{array}{rcl} \eta(\forall \bar{a}.\rho) & = & \forall \bar{a}.\eta(\rho) \\ \eta((x:t).\rho) & = & t \rightarrow \eta(\rho) \\ \eta(t) & = & t \end{array}$$

This main theorem ensures that the type-directed weaving is type-safe.

Theorem 4 (Type Preservation) *Given a program P consisting of a set of advices and a closed base program. If*

$$\vdash P : \sigma \rightsquigarrow P',$$

then

$$\vdash_i P' : \eta(\sigma).$$

[2] Advices defined on functions cannot be treated this way because of possible recursiveness of the functions.

5. Related Works and Discussions

Since the introduction of aspect-oriented paradigm [7], researchers have been developing its semantic foundations. Most of the works in this aspect were done in object-oriented context in which type inference, higher-order functions and parametric polymorphism are of little concern. Recently, researchers in functional languages have also started to study various issues of adding aspects to functional languages. Two notable works in this area, PolyAML [4] and Aspectual Caml [9], have made many significant results in supporting polymorphic pointcuts and advices in strongly typed functional languages such as ML. While these works have introduced some expressive aspect mechanisms into the underlying functional languages, they have not successfully reconciled aspects with parametric polymorphism and higher-order functions – two essential features of modern functional languages. Neither have they adequately addressed the issues of advising advices, which we have discussed in this paper.

PolyAML advocates first-class join points for constructing generic aspect libraries [4]. It allows programmers to define polymorphic advices using type-annotated pointcuts. Unfortunately, PolyAML in [4] does not support *around* advice. The authors are currently extending the language to remedy this [14, 3]. In order to support non-parametric polymorphic advice, PolyAML includes case-advices which are subsumed by our type-scoped advices. Its type system is a conservative extension to Hindley-Milner type inference algorithm with a form of local type inference based on the required annotation on pointcuts. A type-preserving translation inserts labels which serve as marks of control-flow points. During execution, advices are looked-up through the labels and runtime type analysis are performed to handle the matching of type-scoped pointcuts, through which *execution* pointcuts with higher-order functions are supported. It is worth mention that this translation has little resemblance to ours as it does not strive to make weaving decisions at static time. Lastly, advices are anonymous in PolyAML and apparently not intended to be the targets of advising, *aka.* no higher-order advices.

Aspectual Caml [9], on the other hand, does not require annotations on pointcuts. It gives pointcuts the most general types available in context and ensures that the types of the advices hinged on the pointcut are consistent with the type of the pointcut. Similar to PolyAML, it also allows a restricted form of type-scoped advices. Yet, unlike our approach, the types of the functions specified in a pointcut are not checked against the type of the pointcut during type inference. Type safety of advice application is considered later in the weaving process. After type inference, its weaver goes through all type-annotated functions to insert advice calls. For each expression, it looks for advice definitions which have pointcuts that match this expression. If the type of the pointcut is more general than the type of the matched expression, the expression will be replaced by an application to the advice function. This syntactic approach makes it easy to advise anonymous functions. However, for polymorphic functions invoked indirectly through aliases or functional arguments, this approach cannot achieve coherent weaving results. It is also not clear how to extend the syntactic weaving scheme to handle nested advices or higher-order advices.

The current work is a conservative extension of our previous work [13], where we developed a type-directed weaving strategy for functional languages featuring higher-order functions, curried pointcuts and overlapping type-scoped advices. *Around* advices are woven into the base program based on the underlying type context using a Hindley-Milner type inference system extended with advised types and source translation. Coherent translations are

achieved without using any dynamic typing mechanisms. However, in that work, advices and functions are still kept in two completely different levels: advices can neither invoke advised functions nor advise other advices. It was also not clear how to weave advice into polymorphic recursive functions properly. All these shortcomings are fully addressed in this paper by re-designing our type inference system and translation scheme.

In contrast to AspectJ's direct translation into a non-aspect-oriented language, our targeted intermediate language requires addition of chaining expressions. This has been designed for the purpose of presentation clarity. There are many well known schemes such as inlining and closure [2] which can be directly applied to translate the intermediate language into a main stream non-aspect-oriented language. For the purpose of this paper, we omit discussions on this aspect as the added complexity does not contribute any further insights into the static and coherent weaving problem addressed here. Another advantage of our intermediate language is that it supports incremental weaving. Note that a chaining expression $\langle f\ , \{\bar{e}\}\rangle$ has the same static semantics as f. Therefore, it is straightforward to extend our current system to incorporate chaining expressions of the form $\langle f\ , \{\bar{e}\}\rangle$ as the targets for chaining any future advices defined on *f*.

Our type-directed translation was originally inspired by the dictionary translation of Haskell type classes [12]. A number of subsequent applications of it [8, 6] also shares some similarities. However, the issues discussed in this paper are unique, which makes our translation substantially different from the others.

6. Conclusion

Static typing, static and coherent weaving are our main concerns in investigating how to incorporate the essential features of aspects into a core functional language with higher-order functions and parametric polymorphism. As a sequel to our previous results, this paper has advanced our investigation in a variety of ways. Firstly, the target language of our translational semantics of advice weaving has been refined and given a neater formalization. Secondly, we have devised new typing and translation rules to handle the weaving of advices on polymorphic recursive functions. Thirdly, while the basic structure of our type system remains the same, the typing rules have been significantly refined and extended beyond the two-layered model of functions and advices. Consequently, advices can also be advised, either directly or indirectly. All these are accomplished by fully exploring the type information available in context and a novel technique of threading the types of matching advice chains; it is truly a type-directed weaving.

Moving ahead, we shall continue this line of investigation in a few directions. Currently the operational semantics of the intermediate language is purely reduction-based and hence we need to perform α-conversions to avoid name clashes. We plan to look into a closure-based semantics for the intermediate language that should be free of such intricacies. At the aspect language side, some extensions of the pointcuts are worth further investigation. Specifically, we shall consider how to support the control-related *Cflow* pointcuts available in many Java-based aspect-oriented languages. Finally, a prototype implementation is surely a necessary means for us to explore potential applications of our type-scoped advices [13].

References

[1] Aspectwerkz project. http://aspectwerkz.codehaus.org.

[2] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising aspectj. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[3] Daniel S. Dantas, January 2006. personal communication.

[4] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. Polyaml: a polymorphic aspect-oriented functional programmming language. In *Proc. of ICFP'05*. ACM Press, September 2005.

[5] Jboss aop project. http://www.jboss.org/products/aop.

[6] Mark P. Jones. Exploring the design space for type-based implicit parameterization. Technical report, Oregon Graduate Institute of Science and Technology, 1999.

[7] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[8] Jeffrey R. Lewis, Mark Shields, John Launchbury, and Erik Meijer. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages*, pages 108–118, 2000.

[9] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual caml: an aspect-oriented functional language. In *Proc. of ICFP'05*. ACM Press, September 2005.

[10] Harold Ossher and Peri Tarr. Aspectwerkz projectmulti-dimensional separation of concerns in hyperspace, 1999. IBM research report.

[11] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[12] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[13] Meng Wang, Kung Chen, and Siau-Cheng Khoo. Type-directed weaving of aspects for higher-order functional languages. In *PEPM '06: Workshop on Partial Evaluation and Program Manipulation*. ACM Press, 2006.

[14] Geoffrey Washburn, February 2006. personal communication.

A. Sample Derivations

In this section, we present the typing/translation derivation of the examples given in the paper. We use I as a short hand for Int to save space. Some obvious details are also omitted.

A.1 Example 1

The derivation of the definition of f is:

$$\Gamma = \{h :_* \forall a.a \to a \rightsquigarrow h, n_2 : \forall a.a \to a \bowtie h,\\ n_1 : I \to I \bowtie h\}$$

$$\dfrac{\dfrac{\dfrac{h : t \to t \rightsquigarrow dh \in \Gamma_2}{\Gamma_2 \vdash h : t \to t \rightsquigarrow dh}\ (\textsc{Var}) \qquad \dfrac{x : t \rightsquigarrow x \in \Gamma_2}{\Gamma_2 \vdash x : t \rightsquigarrow x}\ (\textsc{Var})}{\Gamma_2 = \Gamma_1, x : t \rightsquigarrow x \vdash (h\ x) : t \rightsquigarrow (dh\ x)}\ (\textsc{App})}{\dfrac{\Gamma_1 = \Gamma, h : t \to t \rightsquigarrow dh \vdash \lambda x.(h\ x) : t \to t \rightsquigarrow \lambda x.(dh\ x)}{\Gamma \vdash \lambda x.(h\ x) : (h : t \to t).t \to t \rightsquigarrow \lambda dh.\lambda x.(dh\ x)}\ (\textsc{Pred})}\ (\textsc{Abs})$$

The derivation of the main expression is:

$$\Gamma_3 = \{h :_* \forall a.a \to a \rightsquigarrow h, n_2 : \forall a.a \to a \bowtie h,$$

$$n_1 : I \to I \bowtie h, f : \forall a.(h : a \to a).a \to a \rightsquigarrow f\}$$

$$\dfrac{\dfrac{\dfrac{f : \forall a.(h : a \to a).a \to a \rightsquigarrow f \in \Gamma_3}{\Gamma_3 \vdash f : (h : I \to I).I \to I \rightsquigarrow f}\ (\text{VAR}) \quad @ \quad \dfrac{\dots}{\dots}}{\Gamma_3 \vdash f : I \to I \rightsquigarrow (f\ \langle h\ , \{n_1, n_2\}\rangle)}\ (\text{REL})}{\Gamma_3 \vdash (f\ 1) : I \rightsquigarrow (f\ \langle h\ , \{n_1, n_2\}\rangle\ 1)}\ (\text{APP})$$

$$@ = \ (\text{VAR-A}) \quad \dfrac{h :_* \forall a.a \to a \rightsquigarrow h \in \Gamma_3 \quad \dots}{\Gamma_3 \vdash h : I \to I \rightsquigarrow \langle h\ , \{n_1, n_2\}\rangle}$$

A.2 Example 2

The derivation of the definition of f is:

$$\Gamma = \{g : I \to I \rightsquigarrow g, head : \forall a.[a] \to a \rightsquigarrow head,\ tail : \forall a.[a] \to [a] \rightsquigarrow tail\}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{f : [a] \to [a] \rightsquigarrow df \in \Gamma_2}{\Gamma_2 \vdash f : [a] \to [a] \rightsquigarrow df}\ (\text{VAR}) \quad \dots}{\Gamma_2 = \Gamma_1, x : [a] \vdash f\ (tail\ x) : [a] \rightsquigarrow df\ (tail\ x)}\ (\text{APP})}{\Gamma_1 = \Gamma, f : [a] \to [a] \rightsquigarrow df \vdash \lambda x....then\ f\ (tail\ x)... : [a] \to [a] \rightsquigarrow \lambda df.\lambda x....then\ df\ (tail\ x)...}\ (*)}{\Gamma \vdash \lambda x....then\ f\ (tail\ x)... : (f : [a] \to [a]).[a] \to [a] \rightsquigarrow \lambda df.\lambda x....then\ df\ (tail\ x)...}\ (\text{PRED})$$

The derivation of the main expression is:

$$\Gamma_3 = \{g : I \to I \rightsquigarrow g, head : \forall a.[a] \to a \rightsquigarrow head,\ tail : \forall a.[a] \to [a] \rightsquigarrow tail, n : I \to I \bowtie f,\ f :_* \forall a.(f : [a] \to [a]).[a] \to [a] \rightsquigarrow f\}$$

$$\dfrac{\dfrac{\dfrac{f :_* \forall a.(h : [a] \to [a]).[a] \to [a] \rightsquigarrow f \in \Gamma_3}{\Gamma_3 \vdash f : (f : [I] \to [I]).[I] \to [I] \rightsquigarrow \lambda y.\langle f\ y\ , \{n\}\rangle}\ (\text{VAR-A}) \quad \dots}{\Gamma_3 \vdash f : [I] \to [I] \rightsquigarrow \mathsf{let}\ F = \lambda y.\langle f\ y\ , \{n\}\rangle\ F}\ (\text{REL-F})}{\Gamma_3 \vdash (f\ [1,2,3]) : [I] \rightsquigarrow (\mathsf{let}\ F = \lambda y.\langle f\ y\ , \{n\}\rangle\ F)\ [1,2,3]}\ (\text{APP})$$

A.3 Example 4

The derivation of the definition of n_3 is:

$$\Gamma = \{f :_* \forall a.a \to a \rightsquigarrow f, n_1 : I \to I \bowtie f,\ n_2 : \forall a.a \to a \bowtie f\}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : a \to a \rightsquigarrow df \in \Gamma_2}{\Gamma_2 \vdash f : a \to a \rightsquigarrow df}\ (\text{VAR}) \quad \dots}{\Gamma_2 = \Gamma_1, arg : a \rightsquigarrow arg \vdash f\ arg : a \rightsquigarrow df\ arg}\ (\text{APP})}{\Gamma_1 = \Gamma, f : a \to a \rightsquigarrow df \vdash \lambda\, arg.f\ arg : a \to a \rightsquigarrow \lambda\, arg.df\ arg}\ (\text{ABS})}{\Gamma_1 = \Gamma, prd : a \to a \vdash \lambda\, arg.f\ arg : (f : a \to a).a \to a \rightsquigarrow \mathsf{let}\ n = \lambda\, df.\lambda\, arg.df\ arg}\ (\text{PRED})}{\Gamma \vdash n3@advice\ around\ g\ (arg) = f\ arg\ \mathsf{in}\ \dots : \dots \rightsquigarrow \mathsf{let}\ n = \lambda\, df.\lambda\, arg.df\ arg\ \mathsf{in}\dots}\ (\text{ADV})$$

Similarly, h is inferred to have type $(g : a \to a).a \to a$. The reason for this advised type is that n_3 fails to be chained with the g-call in that context as the sub-derivation $\Gamma \vdash n_3 : a \to a$ in (VAR-A) fails.

The derivation of the main expression is:

$$\Gamma_3 = \{f :_* \forall a.a \to a \rightsquigarrow f, n_1 : I \to I \bowtie f,\ n_2 : \forall a.a \to a \bowtie f, n_3 : \forall a.(f : a \to a).a \to a \bowtie g\ g :_* \forall a.a \to a \rightsquigarrow g, h :_* \forall a.(g : a \to a).a \to a \rightsquigarrow h\}$$

$$\dfrac{\dfrac{\dfrac{h : \forall a.(g : a \to a).a \to a \rightsquigarrow h \in \Gamma_3}{\Gamma_3 \vdash h : (g : I \to I).I \to I \rightsquigarrow h}\ (\text{VAR}) \quad @ \quad \dots}{\Gamma_3 \vdash h : I \to I \rightsquigarrow (h\ \langle g\ , \{n_3\ \langle f\ , \{n_1, n_2\}\rangle\}\rangle)}\ (\text{REL})}{\Gamma_3 \vdash (h\ 1) : I \rightsquigarrow (h\ \langle g\ , \{n_3\ \langle f\ , \{n_1, n_2\}\rangle\}\rangle\ 1)}\ (\text{APP})$$

$$@ = \ (\text{VAR}) \ \dfrac{\dots}{\Gamma_3 \vdash n_3 : (f : I \to I).I \to I \rightsquigarrow n_3} \quad (\text{VAR-A}) \ \dfrac{\dots}{\Gamma_3 \vdash f : I \to I \rightsquigarrow \langle f\ , \{n_1, n_2\}\rangle}$$
$$(\text{REL}) \ \dfrac{\qquad\qquad}{\Gamma_3 \vdash n_3 : I \to I \rightsquigarrow n_3\ \langle f\ , \{n_1, n_2\}\rangle}$$
$$(\text{VAR-A}) \ \dfrac{\qquad\qquad}{\Gamma_3 \vdash g : I \to I \rightsquigarrow \langle g\ , \{n_3\ \langle f\ , \{n_1, n_2\}\rangle\}\rangle}$$

A.4 Example 5

The derivation of the main expression is:

$$\Gamma = \{f :_* \forall a.a \to a \rightsquigarrow f, n_1 :_* I \to I \bowtie f,\ n_2 : I \to I \bowtie n_1, g : \forall a.(f : a \to a).a \to a \rightsquigarrow g\}$$

$$\dfrac{\dfrac{\dfrac{g : \forall a.(h : a \to a).a \to a \rightsquigarrow g \in \Gamma}{\Gamma \vdash g : (f : I \to I).I \to I \rightsquigarrow g}\ (\text{VAR}) \quad @ \quad \dots}{\Gamma \vdash g : I \to I \rightsquigarrow g\ \langle f\ , \{\langle n1\ , \{n2\}\rangle\}\rangle}\ (\text{REL})}{\Gamma \vdash (g\ 1) : I \rightsquigarrow (g\ \langle f\ , \{\langle n1\ , \{n2\}\rangle\}\rangle\ 1)}\ (\text{APP})$$

$$@ = \ (\text{VAR-A}) \ \dfrac{n_1 :_* I \to I \rightsquigarrow n_1 \in \Gamma \quad \dots}{\Gamma \vdash n_1 : I \to I \rightsquigarrow \langle n1\ , \{n2\}\rangle}$$
$$(\text{VAR-A}) \ \dfrac{\qquad\qquad}{\Gamma \vdash f : I \to I \rightsquigarrow \langle f\ , \{\langle n1\ , \{n2\}\rangle\}\rangle}$$

AOP and the Antinomy of the Liar

Florian Forster
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Universitätsstraße 1
58097 Hagen, Germany
florian.forster@fernuni-hagen.de

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Universitätsstraße 1
58097 Hagen, Germany
steimann@acm.org

ABSTRACT
Unless explicitly prevented, aspects can apply to themselves and can therefore change their own behaviour. This self-adaptation can lead to syntactically correct programs that express antinomies, i.e., that are meaningless (have no intuitive semantics). Drawing the parallel to mathematical logic, we suggest adopting the classical solution presented by Russell and Tarski, i.e., the separation of language into different levels. We propose a simple static type system for AOP that is based on such stratification and that not only helps avoid certain common programming errors, but also reflects on its inherent nature.

Categories and Subject Descriptors
D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *Semantics, Syntax*

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Recursion*

General Terms
Languages, Theory, Verification.

Keywords
Aspects, aspect-oriented programming, meta-programming, self-referentiality, antinomy, paradox, types

1. INTRODUCTION
AOP [4] [11] evolved out of meta programming [12]. It packs intercession, i.e., the possibility to intercept certain events in the course of a program and to insert event-specific behaviour, into a new language construct, the aspect.

Aspects are extremely powerful. In fact, they are so powerful that most contemporary implementations restrict their expressive power through certain syntactical constraints. For instance, most AOPLs do not let aspects advise other aspects (or even themselves). AspectJ [1][10], which has a primitive pointcut `adviceexecution()` that covers all executions of advices, provides constructs such as `cflow(.)` and `within(.)` (or, rather, `!within(.)`) that can be used to prevent self-reference and hence infinite recursion. However, these restrictions and by-passes are usually ad hoc in nature and not argued for on conceptual grounds; in fact, the general approach of language development seems to be that AOPLs are evolved according to their users' needs, and problems are fixed once they are discovered.

In this paper, we take a more principled approach to restricting the expressive power of AOPLs by revisiting a famous series of problems in logic and drawing the analogy to AOP. For this, we briefly recapitulate an ancient paradox known as the antinomy of the liar, and present certain variations of it that can be transformed into aspect-oriented programs (Section 2). Following the reasoning of the logicians who first solved the problem, we argue that any formal language allowing the expression of such antinomies is unsound, and needs mending (Section 3). In Section 4 we present several technical variants of a surprisingly simple solution that not only avoids all paradoxes of the discussed kind, but also other unwanted recursion of aspect application that until today can only be avoided by explicitly introducing certain run-time checks. In the discussion we compare our approach to related work, and find that it sheds some light on the nature of AOP.

2. FAMOUS ANTINOMIES AND THEIR TRANSLATION TO AOP
One of the oldest and also best known antinomies is that of the liar: when Epimenides the Cretan said that all Cretans are liars, and everything else they said was in fact untrue[1], he begged the question whether he himself told the truth, or lied. While the antinomy in Epimenides' utterance depends on certain assumption concerning the meaning of words, the paradox in it its simplest reduction,

"This sentence is false."

is fairly obvious: if the sentence is true, then by its meaning it must be false, and if it is false, the opposite of its meaning must be true, i.e., it must be true, thereby contradicting the presupposition.

This antinomy, which could not be resolved for some 2,500 years, has many incarnations. For instance, consider the following two sentences which, each one for itself being easy to understand, form an unpleasant loop ([8], p. 21):

[1] quoted after Bertrand Russell [13]. The original statement of Epimenides does not appear to have been formulated to provoke a contradiction.

FOAL '06, March 21, 2006, Bonn, Germany.

1. The following sentence is false.
2. The preceding sentence is true.

Interpreting the first sentence as true makes the second sentence false which, assuming a binary (Boolean) logic, would make the first sentence false, thus making the second sentence true. Interpreting the second sentence as true makes the first one true and thus makes itself, the second sentence, false, and so on. There is no way out of this.

2.1 Formulations in AOP

Translating the above two sentences to an AspectJ program is almost straightforward. All we have to do is to replace the truth values *true* and *false* with execution and non-execution, respectively. Sentence 1 then translates to

```
public aspect S1 {
  void around(): adviceexecution() && within(S2) {
  }
}
```

i.e., the advice of `S1` negates the execution of `S2`'s advice (because it contains no `proceed()`). Accordingly, sentence 2 translates to

```
public aspect S2 {
  void around(): adviceexecution() && within(S1) {
    proceed();
  }
}
```

i.e., the advice of `S2` confirms the execution of `S1`'s advice. The intuitive semantics of these two aspects would imply that whenever the advice of `S1` is to be executed, it does not get executed, because the `proceed()` in the advice of `S2` (which would commence its execution) is cancelled by `S1`. Now if one accepts that execution of `S1`'s advice is cancelled, the advice of `S2` (the `proceed()`) does not get cancelled (by `S1`), so that there is not reason why `S1` should not get executed in the first place.

Starting the loop with the advice of `S2`, the picture is not much different: before `S2`'s advice can get executed, that of `S1` is executed, which cancels the execution `S2`'s advice and with it, through cancellation of `proceed()`, also cancels the execution of `S1`'s advice.

The operational semantics of AspectJ (as implemented by its compiler) has a simple solution to this paradox: since it calls the advices of both aspects in alternating order *before* it does anything (i.e., call or not call `proceed()`), it never comes to the core of the problem, but rather causes a stack overflow.

One might argue that `S2` is really a non-aspect, since it does not do anything other than intercept an invocation of `S1`'s advice and then continue with it. In fact, the following reduced aspect `S` could be thought of as inlining `S2` in `S1`:

```
public aspect S {
  void around(): adviceexecution() && within(S) {
    // do something, but do not proceed
  }
}
```

It could be interpreted as the programmatic form of "This sentence is false". Its intuitive semantics again would imply that whenever the advice of `S` is to be executed, it does not get executed, because the `proceed()` in the advice of `S` is lacking. Without a non-executed `proceed()`, however, there is no reason why `S` should not get executed. Admittedly, this is taking intuition a little far, but on the other hand, what is aspect `S` to express? Should it "do something", do nothing, or recur infinitely?[2]

Finally, the antinomy of the liar can be paraphrased in programming terms beginning with "all routines returning a truth value are always (i.e., for all calls) wrong". The passionate AO programmer might believe that this could easily be corrected by introducing a repair aspect, namely by

```
aspect Negate {
  Object around(): execution(* *(..))
      || adviceexecution() {
    Object c = proceed();
     if (c instanceof Boolean && c!= null)
        return !((Boolean) c);
     else
       return c;
   }
}
```

However, since the aspect would also have to correct itself, it is unclear what it should return in this case: upon execution, the above AspectJ code does the best it can — it runs into an infinite recursion, thus refusing to give an answer to the question.

2.2 Antinomies That Currently Cannot Be Expressed

There are also variations of the antinomy that cannot be expressed in AspectJ. Among the most famous is the barber who shaves all and only the people who do not shave themselves: assuming the barber shaved himself he would disregard the condition to shave only the people who do not shave themselves; assuming that he did not shave himself on the other hand he would, by the premise of his job description, have to shave himself. One way or another, the barber fails to meet the requirements of his task.

At first glance, this antinomy can be easily transcribed to AOP, namely to the following, informally defined aspect:

> "Aspect `Barber` advises all and only the aspects that do not advise themselves."

In AspectJ, that a concrete aspect `A` advises itself, i.e. its own pieces of advice, is expressed by the following pointcut:

```
adviceexecution() && within(A)
```

Conversely, that the aspect `A` does not advise itself is expressed by

```
adviceexecution() && !within(A)
```

The difficulty comes from generically expressing *all* aspects that advise, or do not advise, themselves. Due to existing language restrictions, AspectJ currently has no means of checking if an aspect advises itself. Whether intentional or not, this restriction saves AspectJ from being able to express the Barber's antinomy.

2.3 Non-Paradoxical Recursion

That the `adviceexecution()` pointcut designator can lead to infinite recursion is a well-known problem. In fact, in [6] it is stressed that

[2] As it turns out, it will recur infinitely as the advice is executed ("called") even though it does nothing. An optimizing aspect compiler might however change this semantics.

> *[t]he preferred way to use the* `adviceexecution()` *pointcut is to pair it with* `within(YourAspect)`*, thus limiting its scope to advice appearing in the body of* `YourAspect`*.*

"The AspectJ Programming Guide" [3] gives a concrete example of this and shows how to avoid it:

```
aspect TraceStuff {
  pointcut myAdvice(): adviceexecution() &&
    within(TraceStuff);
  before(): call(* *(..)) && !cflow(myAdvice()) {
    // do something matching call(* *(..))
  }
}
```

However, the recursion that would occur in the application of `TraceStuff` if `!cflow(myAdvice())` were not included in the pointcut of the before advice can be considered a plain programming error.[3] In particular, it does not give rise to antinomies of the above kind, and its interpretation by the AspectJ compiler is not at conflict with its intuitive semantics. On the other hand, it is a programming error that is easily overlooked, one that would be nice if the language definition prevented the programmer from. We will return to this issue in Section 4.

2.4 Aspect Recursion Not Involving the Advising of Advice

Finally, we point the reader to the fact that there is a form of (usually unintended, i.e., erroneous) recursion that is caused by aspect application, but that does not involve the advising of aspects. The following gives an example of this:

```
public class Innocent {
   public void someMethod() {
     ...
   }
}

public aspect Naughty {
   before(Innocent a):
       execution(void Innocent.someMethod())
       && target(a) {
     a.someMethod();
   }
}
```

Note that recursion does not involve an `adviceexecution()` pointcut.

This kind of problem occurs when aspects access elements of the base program, thereby triggering (other) aspects including themselves. This however is of a different quality than the problems induced by the self-referentiality of aspects discussed above, and we make no proposals suggesting how to avoid such problems.

3. GREAT ESCAPES

It was one of the most significant mathematical discoveries of the early 20th century that antinomies of the presented kind are not the result of some linguistic sophistry, but rather question the fundamentals of all mathematical reasoning. In fact, mathematicians of that time (including Russell) seriously considered abandoning set theory altogether (and with it the concept of classes and relationships). Luckily for us, they did not, but instead came up with several solutions that avoided these problems. One of the earliest was formulated by Russell himself as his "theory of types", the essential idea of which, the distinction of different levels of propositions, was later repeated in Tarski's contemplations regarding the notion of truth. As it turns out, Russell's and Tarski's solution makes a useful contribution to AOP, but before transferring it to our problem, we briefly revisit the original works, one by one.

3.1 Russell's Theory of Types

In the year 1901 Russell discovered a fundamental problem in the naïve form of set theory that at that time was thought to be the basis of mathematics. In [13] he formulated "the class of all those classes which are not members of themselves":

$$M = \{X \mid X \notin X\}$$

The problem with this definition is that whichever of the two possible alternatives $M \in M$ and $M \notin M$ one assumes, the opposite follows:

$$M \in M \Rightarrow M \notin M$$
$$M \notin M \Rightarrow M \in M$$

Russell noted that the problem can only be avoided by agreeing that "[w]hatever involves *all* of a collection must not be one of the collection". However, the problem is that it is unobvious how to specify such a condition, since

> *[w]e cannot say: "When I speak of all propositions, I mean all except those in which 'all propositions' are mentioned"; for in this explanation we have mentioned the propositions in which all propositions are mentioned, which we cannot do significantly. […] The exclusion [therefore] must result naturally and inevitably from our positive doctrines, which must make it plain that "all propositions" and "all properties" are meaningless phrases.* [13]

Russell solved this problem constructively by introducing a "hierarchy of types":

> *A* type *is defined as the range of significance of a propositional function, that is, as the collection of arguments for which the said function has values. Whenever an apparent variable occurs in a proposition, the range of values of the apparent variable is a type, the type being fixed by the function of which "all values" are concerned. The division of objects into types is necessitated by the reflexive fallacies which otherwise arise. These fallacies, as we saw, are to be avoided by what may be called the "vicious-circle principle", that is, "no totality can contain members defined in terms of itself". This principle, in our technical language, becomes: "Whatever contains an apparent variable must not be a possible value of that variable". Thus whatever contains an apparent variable must be of a different type from the possible values of that variable; we will say that it is of a* higher *type. Thus the apparent variables contained in an expression are what determines its type. This is the guiding principle in what follows.* [13]

Transferred to our problem of self-reference in AOP, the function

advice(*joinpoint*)

defines as a type the set of possible values the variable *joinpoint* may adopt. The value of *advice*(*joinpoint*) however must be of a

[3] In fact, in [2] the authors note that "circular `adviceexecution()` applications are very rare, and usually pathological and a symptom of an error in the program."

higher type, so that it cannot be a value of *joinpoint*. It follows that no advice can serve as its own join point or, phrased differently, that no advice can advise itself. We will exploit this in our typing system for AOP described in Section 4.

It is interesting to note that Russell's type theory was only later generalized into sorted (and also order-sorted) predicate logic, whose sorts map closely to the types we know from typed programming languages. Since logic is usually restricted to first order, its sorts are all of Russell's type 1, i.e., they are sets of individuals (the objects). Our type system suggested in Section 4 lifts this restriction.

3.2 Tarkski's Distinction between Object Language and Meta-Language

In his discussion of the semantic conception of truth [17] Tarski analyzed the assumptions which lead to the antinomy of the liar, and observed the following:

I. We have implicitly assumed that the language in which the antinomy is constructed contains, in addition to its expressions, also the names of these expressions, as well as semantic terms such as the term "true" referring to sentences of this language; we have also assumed that all sentences which determine the adequate usage of this term can be asserted in the language. A language with these properties will be called "semantically closed."

II. We have assumed that in this language the ordinary laws of logic hold.

[…] Since every language which satisfies both of these assumptions is inconsistent, we must reject at least one of them. [17]

Because the ordinary laws of logic are hard to renounce, it seems that semantic closedness cannot be upheld. Now if we agree

> *not to employ semantically closed languages, we have to use two different languages in discussing the problem of the definition of truth and, more generally, any problems in the field of semantics. The first of these languages is the language which is "talked about" and which is the subject matter of the whole discussion; the definition of truth which we are seeking applies to the sentences of this language. The second is the language in which we "talk about" the first language, and in terms of which we wish, in particular, to construct the definition of truth for the first language. We shall refer to the first language as "the object language," and to the second as "the meta-language."* [17]

Tarski further argues that in order to make statements about statements formulated in the object language, "the meta-language must be rich enough to provide possibilities of constructing a name for every sentence of the object language." Regarding truth, the meta-language must also contain terms of general logic such as AND, OR and NOT.

It springs to mind that the meta-language of Tarski and aspect languages (AspectJ in particular) have a lot in common. Quite obviously, since AspectJ extends Java, every sentence of the object language (Java) can occur in the meta-language (AspectJ). Names for expressions in the object language can be constructed by using pointcuts (the fact that it is not possible to construct a pointcut for every element of the object language is merely a limitation of the implementation). Last but not least, the meta-language contains logical terms for the formulation of pointcuts. Because we were able to reconstruct the antinomies in AspectJ, we conclude that it is semantically closed; in order to avoid them, we have to introduce a clear distinction between object language and meta-language.

4. TYPE-SAFE AOP

We will start the presentation of our solution with a practical example. It contains a recursion analogous to those presented in Section 2.3 and [3], but no antinomy. However, as we will elaborate later our solution is powerful enough to also avoid all antinomies we were able to express in Section 2.1, as well as ones that cannot (yet) be formulated, enabling certain future language extensions that seem too risky today.

One of the best known (and most often cited) applications of aspects is tracing: if the execution paths of a program become unobvious, a trace may help to find out what exactly is going on. However, because of its obliviousness property AOP comes with its very own tracing demands: the programmer might be particularly interested when a certain aspect (or all aspects) are executed or, more challenging, in which order certain conflicting pieces of advice are executed on the same join point[4].

Writing an advice that traces all method executions and advice executions seems an easy exercise. The first solution a programmer might propose, namely

```
public aspect Tracing {
  void around(): adviceexecution()
    || execution[5] (* *(..)) {
    System.out.println("Entering:" +
      thisJoinPoint);
    proceed();
    System.out.println("Leaving: " +
      thisJoinPoint);
  }
}
```

as a tracing aspect that traces both method and advice executions, as for instance

```
public aspect Worker {
  void around(): execution(* *(..)) {...}
}
```

and

```
public class Base {
  public void doSomething() {...}
}
```

does not work. Taking a closer look reveals that the pointcut attached to the tracing advice selects the tracing advice itself (by means of the unrestricted primitive pointcut `adviceexecution()`), sending AspectJ into infinite recursion. This is clearly a programming error, which has to be fixed somehow.

---

[4] Note that by the current definition of advice precedence in AspectJ this order might be impossible to determine. Even worse, it may change in between two compiler runs. [5]

[5] Although we consistently use the pointcut designator `execution(.)` for referring to the base program throughout the following, it should be understood that it could be replaced by other pointcut designators such as `set(.)` or `get(.)`.

An immediate solution would appear to be using the pointcut designator `within(<TypePattern>)`, where `<TypePattern>` identifies a number of classes, interfaces and/or aspects. The formerly unrestricted pointcut `adviceexecution()` can then anded with (restricted by) `!within(Tracing)`, i.e. only pieces of advice which are not in the lexical scope of the aspect `Tracing` are selected, as the following example shows.

```
public aspect Tracing {
  void around(): (adviceexecution()
    && !within(Tracing))
    || execution (* *(..)) {
    ...
  }
}
```

As it turns out, however, this construction cannot avoid indirect recursion. In fact, when applying it to aspect `S1` from Section 2.1, it must remain ineffective, since `within(S2)` implies `!within(S1)`. Therefore, one has to check explicitly whether `S1` has already been activated, a test that can be performed with the aid of the `cflow()` function. Hence, in order to be sure that self application is under all circumstances avoided, one has to include the verbose construct presented in Section 2.3. Thus, our tracing aspect becomes the clumsy

```
public aspect Tracing {
  pointcut guard(): adviceexecution() &&
    within(Tracing);
  void around(): (adviceexecution()
    || execution (* *(..))) &&
    !cflow(guard()) {
    ...
  }
}
```

It seems that the introduction of `adviceexecution()` as a means to let aspects apply to aspects has made necessary a programming pattern that serves to fix the resulting problems. However, this pattern means that the infinite recursion introduced by `adviceexecution()` has to be *explicitly* detected and broken, and this *at runtime*. What would be desirable instead is that `adviceexecution()`, while allowing certain (wanted) recursion, can never mean the (generally nonsensical) infinite recursion to itself.[6] In the following, we build such a solution on a theory of types as proposed by Russell or, equivalently, on a theory of object language and meta-language as proposed by Tarski. We develop the solution in a stepwise manner, by first presenting a programming pattern using annotations to introduce type (or meta) levels, then sketching a preprocessor utility for AspectJ that frees the programmer from the coding overhead and error-proneness of the pattern, and finally by suggesting the addition of a new keyword `meta` to AspectJ whose semantics does the job all automatically.

4.1 Step 1: Using Annotations and a Simple Programming Pattern

The basic idea of the solutions of Tarski and Russell was the introduction of different levels of language. What we need, therefore, is a way to organize the elements of an aspect-oriented program into several levels. With the new annotation feature of Java 5.0 and AspectJ5 one can introduce such stratification, thereby simulating the distinction of Java as the object language and AspectJ as the meta-language, or the type levels of Russell.

We declare the annotation needed for this purpose as follows:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface TypeLevel {
   int value() default 0;
}
```

Note that our annotating `TypeLevel` with the built-in meta-annotation `@Retention(RetentionPolicy.SOURCE)` implies that we evaluate the annotations statically (in contrast to `cflow()`, which can only be evaluated dynamically!). A second built-in meta-annotation, `@Target`, is set to `ElementType.TYPE`; it prevents the annotation of elements other than types, i.e. classes, interfaces, and aspects, by prompting a corresponding compilation error.

Our `TypeLevel` annotation has one argument which represents the meta-level of the annotated element. By definition the elements (class or interface) of the object language will have a meta-level of 0, meaning that they must be annotated with `@TypeLevel(0)`, and the elements (aspects) of the meta-language addressing elements of the object language (i.e., advice without an `adivceexecution()` in its pointcut) will have a meta-level of 1, meaning that they must be annotated with `@TypeLevel(1)`.[7] Thus, the base of our aspect `Tracing` must be annotated as

```
@TypeLevel(0)
public class Base {
  public void doSomething() {...}
}

@TypeLevel(1)
public aspect Worker {
  void around(): execution(* *(..)) {...}
}
```

When moving to the next higher level, the (former) meta-language becomes the (new) object language, so that the (new) meta-language ranges at level 2: advice with an `adviceexecution()` in its pointcut is to be annotated with `@TypeLevel(2)` or higher:

```
@TypeLevel(2)
public aspect Tracing {
  void around(): adviceexecution() {...}
}
```

The problem that remains is how to restrict the scope of the `adviceexecution()` pointcut to aspects of levels lower than that of its enclosing aspect. As it turns out, AspectJ 5 is equipped with the `@within(Annotation)` pointcut designator that matches only join points belonging to a type annotated with `Annotation`. By adding `@within(TypeLevel)` plus the explicit guard `if(TypeLevel.value() < 2)` to the pointcut, our tracing aspect can be formulated as

```
@TypeLevel(2)
public aspect Tracing {
  void around(): (adviceexecution()
    && @within(TypeLevel)
    && if(TypeLevel.value() < 2))
    || execution (* *(..)) {
    System.out.println("Entering: " +
```

[6] Note that this cannot be achieved simply by excluding every occurrence of `adviceexecution()` from its own scope, since the recursion may be indirect. *Cf.* also Russell's comment on the impossibility of explicit avoidance of self-reference in Section 3.1.

[7] Note that both Russell and Tarski introduced no absolute, but only relative levels. However, since our domain is AOP, the level of the (non-aspect) base program is as low as we can get.

```
      thisJoinPoint);
    proceed();
    System.out.println("Leaving: " +
      thisJoinPoint);
  }
}
```

without limiting its meaning unduly.

Unfortunately, things are not as simple with the current implementation of AspectJ, as the following example shows:

```
@TypeLevel(2)
public aspect Tracing {
  void around(): (adviceexecution()
    && @within(TypeLevel)
    && if(TypeLevel.value() < 2))
    || execution (* *(..)) {
      helpMethod();
    }
  }
  void helpMethod() {...}
}

@TypeLevel(1)
public aspect Worker {
  void around(): execution(* *(..)) {...}
}
```

When including in the aspect `Tracing` an arbitrary helper method (here: `helpMethod()`) and calling it from the aspect's advice, aspect `Worker` (which advises *all* method executions) advises the execution of this method, and therefore indirectly also the aspect `Tracing` even though it is of a higher type level. This leads to an infinite recursion as the execution of `Worker`'s advice triggers the advice of `Tracing`'s which executes the method `helpMethod()` again. The only way out of this (without checking the call stack) is to exclude `execution()` from applying to methods defined within aspects.[8] To achieve this, `execution(.)` pointcuts also have to be guarded:

```
@TypeLevel(2)
public aspect Tracing {
  void around(): (adviceexecution()
    || execution (* *(..)))
    && @within(TypeLevel)
    && if(TypeLevel.value() < 2) {
    helpMethod()
  }
  void helpMethod() {...}
}

@TypeLevel(1)
public aspect Worker {
  void around(): execution(* *(..))
    && @within(TypeLevel)
    && if(TypeLevel.value() < 1) {
    ...
  }
}
```

The accidental recursion is thus removed. It follows immediately that annotating base type (classes and interfaces) with `@TypeLevel(0)` cannot be avoided, although at first glance this seems to be redundant (because the base has `execution` and other exclusive pointcut designators that cannot apply to pieces of advice, and because it cannot be caught by `adviceexecution()`).[9]

Unfortunately, this solution has several problems. First, it only works if all types are tagged with their corresponding annotation, because if a type (class, interface, or aspect) is not annotated, a guarded pointcut will not select its join points, voiding all its aspects. Second, the programmer is responsible for ensuring the constraint that the value of the type guard of a pointcut is always lower than its own aspect's type level (because there are no means to instruct the compiler to check annotation values). Last but not least, the required code is highly stereotypical (it is in fact a coding pattern), and experience teaches that the implied programming overhead will not be welcomed by practicing programmers, particularly if workarounds requiring less coding (the `within(.)/ !cflow(.)` pattern) are available. Since annotating types and guarding advice cannot be enforced by the compiler, it will most likely not be used. On the other hand, much of the task is so stereotypical that it can be delegated to a pre-processor, as discussed next.

4.2 Step 2: Using a New Built-in Annotation

The next major Java release (codenamed "Mustang") will allow user-defined annotations to be included into the compilation process by means of a special interface to the compiler [9]. Once available, this pre-processing facility should allow us to extend the compiler with a pre-processor reducing the work and responsibility of the developer, and thus the likelihood of making errors. In this section we will therefore sketch such a pre-processor which, in concert with a correctly annotated program, statically ensures that the typing conditions of our language are satisfied.

In our description, we assume a procedure for pre-processing described in the Annotation Preprocessing Tool Manual [16]. The pre-processor for the tagging task, making sure that every aspect is appropriately annotated, is straightforward to write:

```
foreach type in program
  if isTypeTagged(type)
    do nothing
  else
    if (type == Class || type == Interface)
      type.tagWithLevel(0)
    if (type == Aspect)
      type.tagWithLevel(1)
endfor
```

Therefore, when feeding an untagged program to the pre-processor, it assumes that it consists of only base program and level 1 aspects, but no aspects advising aspects.[10] After this pre-processing step every type is, either by the pre-processor or by the developer, tagged with a `TypeLevel` annotation.

In the next step the pre-processor must generate the guards which are required to complete stratification of our language. The following pseudo code attaches the code pattern `@within(TypeLevel) && if(TypeLevel.value() < t)`, where `t` is the type level of the enclosing aspect, to every pointcut (un-

[8] One may ask oneself why AspectJ, while granting advice execution a different status ("higher level") than method execution, does not extend this to the methods defined within the aspect, in particular since inlining these methods should not change the meaning of the aspect.

[9] Note that if one insists that `helpMethod()` in `Tracing` is of type level 0 (the AspectJ view; *cf*. Footnote 8), i.e., part of the base, the resulting recursion is analogous to that discussed in Section 2.4, meaning that it cannot be broken by our type system.

[10] As we will see below, occurrence of an `adviceexecution()` pointcut in such a program will flag an error.

named or named) in the lexical scope of the aspect under investigation.

```
foreach aspect in program
  foreach pointcut in aspect
    t := getTypeLevel(aspect)
    attachGuard(pointcut, t)
  endfor
endfor
```

The generated guard allows the pointcut to select only join points occurring in the lexical scope of types annotated with a type level below its own. Thus our tracing advice

```
@TypeLevel(2)
aspect Tracing {
  void around(): adviceexecution()
    || execution (* *(..)) {...}
}
```

will be automatically extended to

```
@TypeLevel(2)
aspect Tracing {
  void around(): (adviceexecution()
    || execution (* *(..)))
    && @within(TypeLevel)
    && if(TypeLevel.value() < 2) {...}
```

whereas

```
@TypeLevel(1)
aspect Tracing {
  void around(): adviceexecution()
    || execution (* *(..)) {...}
}
```

will be extended to

```
@TypeLevel(1)
aspect Tracing {
  void around(): (adviceexecution()
    || execution (* *(..)))
    && @within(TypeLevel)
    && if(TypeLevel.value() < 1) {...}
```

which has an empty pointcut, because `adviceexecution()` only matches to program elements in the scope of type level 1 or higher. At this point, the pre-processor should flag a type level mismatch error.

The annotation-based pre-processing suffers from one rather subtle problem: it assumes that all pointcuts are intended to refer to program elements of *any* lower level. However, a programmer might want to specify that `adviceexecution()` should match advice at a particular level (and no other), and this level need not even be precisely 1 below itself. In such a case, an explicit guard (involving "=" rather than "<") will be required. We will present a more elegant solution for this in the next step.

4.3 Step 3: Extending the Language with the `Meta` Modifier

Although the exploitation of "semantic" (i.e., built-in, but user-defined) annotations reduces the programming overhead and the possibility to make mistakes, it is still only a precursor to full language support. In particular, it would be desirable for the compiler to detect and flag all errors related to the typing of aspects, just as it discovers other, conventional typing errors. Also, we believe that our suggested typing of aspects deserves the status of a new, native language construct, since it addresses a fundamental problem inherent in AO languages. Therefore, we propose a small, yet very effective extension to AspectJ which equips it with a type system à la Russell (not to be confused with the type system of Java) and Tarski, allowing the safe advising of advice.

Frankly, in our extended language attempting to compile

```
public aspect Tracing {
  void around(): adviceexecution() ...
}
```

would result in an error message "type level mismatch error: consider modifying aspect Tracing with meta", because `adviceexecution()` may refer to itself. The keyword `meta` preceding an aspect definition lifts the so-declared aspect up one level, i.e., it declares it as an aspect of both aspects and base programs (where the former must themselves be aspects of base programs, not of aspects). For instance,

```
meta aspect Tracing {...}
```

makes `Tracing` a meta aspect that can apply to the base program and aspects (`Base` and `Worker` in the above example), but not meta aspects, thereby excluding self-reference. The pointcut definition of `Tracing` can remain as is; in particular, it need not be explicitly guarded: it can refer only to lower levels by the definition of the language.

Now the lifting procedure can be applied repeatedly, raising the meta level of aspects even further. Although there will most likely be no need for having aspects on a level higher than 3 (given the usual four-layer architecture), there seems to be no obvious theoretical bound to meta levels. Therefore, rather that introducing ever new `meta` modifiers, we propose to denote the meta-meta level with `meta^2`, and generally concatenation of *n* metas by `meta^n`. `meta` is then simply shorthand for `meta^1`, and absence of `meta` is shorthand for `meta^0`. However, it is important to note that theoretically, for $n > 0$ each `meta^n` represents a different keyword of our language, and our shorthand notation is only introduced to allow the compiler to accept them as they are used in a program. We will return to this subtlety in Section 5.4. Here, we note that an aspect that is to apply to `Tracing` would have to be declared as

```
meta^2 aspect GodAspect {...}
```

or higher.

Following Russell's theory of types, we allow meta-aspects to apply to aspects as well as to base programs, rather than to aspects alone. We have no particular reason for this other than that we do not want to place unnecessary restrictions on the formalism. Had we decided that aspects can exclusively apply to program elements one level below them, no distinction between the `execution()` and the `adviceexecution()` pointcut designator would have been necessary: `execution()` would have sufficed (denoting base code or aspect execution, depending on the level of the defining aspect).

With the possibility of `adviceexecution()` pointcuts spanning arbitrary levels, we may wish to have increased precision available for expressing specifically to which level a pointcut applies. For instance, while we can already distinguish between base program (`execution(.)`) and aspect (`adviceexecution()`) and thus between type level 0 and higher levels, we may wish to be able to differentiate in our pointcuts between type level 1 and 2. Therefore, we allow that the pointcut designator `adviceexecution()` can also be modified with the keyword `meta`, specifying the exact type level to which the so-modified pointcut is to apply. The pointcut

```
pointcut metaAdvice(): meta adviceexecution();
```

would thus select only advice defined in aspects of type (or meta) level 2, like our aspect `Tracing` from above. The meaning of the (unmodified) pointcut designator `adviceexecution()` is then restricted to aspects of type level 1, i.e., those that are not meta aspects. Our tracing aspect can thus be rewritten as

```
public meta aspect Tracing {
  void around(): adviceexecution()
    || execution (* *(..)) {
    ...
  }
}

public aspect Worker {
  void around(): execution(* *(..)) {...}
}

public class Base {
  public void doSomething() {...}
}
```

Note that, as mentioned at the beginning of this subsection, using the pointcut designator `adviceexecution()` in an ordinary (i.e., non-meta) aspect or, generally, `meta[^n] adviceexecution()` in any aspect declared as `meta[^m]` with $m \leq n$, would result in a compilation error, since it violates the typing rules of our language extension. The following table summarizes what is possible.

Aspect level	allowed pointcut designators
`aspect`	`execution()`
`meta aspect`	`execution(), adviceexecution()`
`meta^2 aspect`	`execution(),adviceexecution(), meta adviceexecution()`
`meta^3 aspect`	`execution(),adviceexecution(), meta adviceexecution(), meta^2 adviceexecution()`
…	…

This so modified AspectJ is now type safe in terms of the type theory of Russell, and the antinomies presented in Section 2 cannot be formulated in it, as the following demonstrates.

4.4 Resolving the Antinomies

With our new type system implemented, the code translation of the two contradictory sentences from Section 2.1 would now result in a type level mismatch (compilation) error, because the included (indirect) self-reference, i.e. `adviceexecution()`, while applying to type level 1, is in the lexical scope of an aspect of type level 1. In order to be well-typed, both `S1` and `S2` must be modified with `meta` as in

```
public meta aspect S1 {
  void around: adviceexecution() && within(S2) {
  }
}

public meta aspect S2 {
  void around: adviceexecution() && within(S1) {
    proceed();
  }
}
```

This however automatically prevents the self-reference and thus the infinite recursion. In fact, both pointcuts do not select any join point, since `adviceexecution()` implicitly applies to type level 1 whereas `within(S2)` and `within(S1)` apply to type level 2, so that the conjunction is always false. A corresponding compiler-generated error, or at least a warning, to notify the developer of this problem would seem desirable.

In the same vein, the recursions of all other paradoxical aspects presented above are naturally resolved. For instance, by modifying the declaration of the `Negate` aspect to `meta aspect Negate` eliminates the possible self-reference, and thus the need to restrict `adviceexecution()` by means of other pointcuts like `within(.)` and `cflow(.)`. The same holds for the unwanted recursion warned of in Refs. [3] and [6].

4.5 Handling of Aspect Members and Inter-Type Declarations

In Section 4.1 we mentioned that the weaving of AspectJ treats methods extracted from an advice as ordinary (base) methods, and showed how this can lead to indirect recursion. To solve this problem in our proposed extension of AspectJ, we assign to every join point in an aspect the type level of that aspect (cf. Footnote 8). Therefore, it cannot be matched by pointcuts of the same or lower levels, breaking the recursion.

To allow selective matching of the join points of an aspect exposed by its members (methods and fields), we further extend AspectJ to allow modification of all other pointcut designators (i.e., `call(.)`, `execution(.)`, `set(.)`, `get(.)`, etc.) with `meta^n`. `meta^n <pointcut>` will select only join points occurring in the lexical scope of aspects of the corresponding level. Using `meta^n <pointcut>` in an aspect declared as `meta^m` with $m < n$ will result in a (statically discovered) type level mismatch error. The complete table of admissible pointcut designators in each aspect type level is the following:

Aspect level	allowed pointcut designators
`aspect`	current AspectJ pointcut designators excluding `adviceexecution()`
`meta aspect`	same as above plus `adviceexecution()` plus every other AspectJ pointcut designator modified with `meta`
`meta^2 aspect`	same as above plus `meta adviceexecution()` plus every other AspectJ pointcut designator modified with `meta^2`
…	…

In order to catch all method executions in the base program and its (type level 1) aspects, our tracing aspect has to be rewritten as

```
public meta aspect Tracing {
  void around(): adviceexecution()
    || meta execution (* *(..))
    || execution (* *(..)) {
    ...
  }
}
```

It traces both the advice and the (helper) method of

```
public aspect Worker {
  void around(): execution(* *(..)) {
    someMethod();
  }
  void someMethod() {...}
}
```

For the convenience of the programmer it might prove useful to allow modification through `meta^n` also for defining the scope of named pointcuts. Instead of writing

```
void around(): meta get(* *)
  && meta set(* *) {...}
```

one could then write

```
meta pointcut accessors(): set(* *) && get(* *);
void around(): accessors() {...}
```

One remaining issue is that of how AspectJ's inter-type declarations are to be integrated into our typing system. Returning to our tracing example once more, we extend the aspect `Tracing` with an introduction affecting the aspect `Worker`.

```
public meta aspect Tracing {
  void around(): adviceexecution()
    || meta execution (* *(..))
    || execution (* *(..)) {
    ...
  }
  void Worker.doGood() {...}
}

public aspect Worker {
  void around(): execution(* *(..)) {...}
}

public class Base {
  public void doSomething() {...}
}
```

According to the current semantics of AspectJ the introduction `Worker.doGood()` is considered to be a member of the type it is introduced to, i.e., at least as regards join point matching it is equivalent to defining the method in the aspect `Worker` directly. This is in accord with our typing system: any aspect introducing elements to lower level types can also watch over their execution. For instance, in the above example the tracing aspect traces all executions of `doGood()` in `Worker`. Currently, we can see no need to restrict introductions to lower levels, i.e., introduction to same or higher levels should also be possible, with the restriction that these introductions can not be covered by pointcuts of the introducing aspect.

4.6 Enabled Language Extensions

With our type-level language extension defined as above, we are now ready to extend AspectJ safely with constructs allowing the expression of aspects that advise, or do not advise, themselves.[11] For instance, a special variable `targetaspect` can now be introduced that refers to the (instance of) the aspect whose join point (as captured by an `adviceexecution()` pointcut) is currently handled. Another special variable `thisaspect` can be added that refers to the (instance of) the current (handling) aspect. Note that the type (level) of `thisaspect` is always the same as that of the advice in whose context (lexical scope) the variable occurs, while that of `targetaspect` is necessarily of a lower level; therefore, the expression of the aspect from Section 2.2 that advises all aspects that do not advise themselves,

```
aspect Barber {
  void around(): adviceexecution() {
    if (targetaspect != thisaspect) {
      proceed();
    else {}
  }
}
```

causes a type level mismatch error in line 3.

[11] A similar request for language extension has been formulated in [14].

5. DISUSSION AND RELATED WORK

5.1 Dependence of the Antinomies on a Declarative Interpretation

When reconstructing the logical antinomies in AspectJ in Section 2.1, we relied on what we called "intuitive semantics". This assumed intuitive semantics is basically a declarative one, i.e., we read the programs as assertions rather than as sequences of instructions. When looking at it with procedural glasses on, the advice of aspect `s` in Section 2.1 would read as "before executing any advice, call the advice of `s`". Since "any advice" includes itself, the advice of `s` is called recursively *before* anything else is (not) done. Therefore, one might argue that there is no antinomy in the program, just an infinite recursion. However, the reader will agree that this procedural semantics (as defined by the AspectJ compiler) is non-obvious at best, and that in a well designed language, intuitive semantics should match the operational one (the principle of least surprise).

As an aside, it is interesting to note that the procedural semantics of aspects allows them to avoid self-reference through the `cflow()` construct. In fact, in a purely declarative interpretation, particularly without a notion of sequentiality and without having access to the history of execution, an exclusion of self-reference cannot be formulated (as noted by Russell in the quote of Section 3.1). The price for this check is that it has to be done at runtime (and is in fact very expensive); by contrast, our type system allows a static check, which (in terms of runtime overhead) is entirely free.

5.2 Typing to Prevent Programming Errors

Even if one denies the existence of antinomies in the aspects constructed in Section 2.1, one will agree that a well-designed programming language should save its programmers from programming errors. In fact, type systems are generally accepted as serving this purpose; they discover many possible type mismatches at compilation time. However, the aspects of AspectJ, although syntactically similar to classes, are untyped; therefore, current typing systems cannot prevent any errors related to advice application. We have adapted a typing system well-suited for this purpose from Russell's theory of types and Tarski's theory of object language and meta-language; although it looks very different from that base language's (i.e., Java's) type system, it serves the same purpose: it prohibits the construction of illegal programs, and it prevents programming errors.

5.3 The Orderedness of AOPLs

It has been noted elsewhere that AOPLs are necessarily second-order languages [15]. Second-orderedness by definition excludes self-referentiality, so that in all AOPLs that are true second order languages (as are all those languages that exclude aspects from being applied to aspects), the above antinomies cannot be expressed. However, as we have demonstrated here, at least AspectJ as its stands is an unordered language; like unordered logic, it allows the construction of Russell's "vicious circles". From all we can see, making AspectJ a well-ordered ("typed") language as proposed here fixes the problems without imposing any undue restrictions.

5.4 Aspects of Aspects and the Closure of Languages

The notion of aspects of aspects has stirred some theoretical contemplation concerning the closedness of aspect languages. In [7], the authors state that the goal of any aspect language should be that it be "closed with respect to aspectification (aspect closure)". This is expressed by the equation $A(L) = L$, meaning that expressing aspect application to the language elements of L would make do with L, that is, would not require additional language elements. From this, they deduce that AspectJ as an instance of A(Java) is currently not closed, since obviously AspectJ ≠ Java, but also (currently) A(AspectJ) ≠ AspectJ. They argue in favour of such a closure since they observed that certain languages incorporating meta-programming, such as Smalltalk or CLOS, are also self-contained, i.e., that there $M(L) = L$. However, they ignore that this is only possible because these languages resort to certain tricks: for instance, in Smalltalk the class `MetaClass` is an instance of itself. This however forbids the semantic interpretation of classes as sets and instances as elements of sets, since then the set of `MetaClass` would have to contain itself. At the same time, it is at odds with Tarski's fundamental observation that the meta-language must be richer than its object language, meaning that it cannot be "semantically closed".

Returning to the generic `meta^n` keyword discussion from Section 4.3, we note that our language, although handled by a single compiler, is not semantically closed, since every new meta-level, i.e., $A(L)$ where the highest type level in L is n, requires a new keyword `meta^n`. In a similar vein, the syntax of predicate logic can encompass various orders (i.e., first order predicate logic, second order predicate logic, and so forth).

5.5 Testing Advice

In Ref. [14] it is argued that testing is a crosscutting concern, i.e., that testing code spreads across the whole system, and thus lends itself to being extracted to an aspect. All testing code can be encapsulated into one module which has privileged access to the original source without needing to modify it. Furthermore, the testing code is easily removed, by excluding it from compilation. As AOP seems to be well suited for testing object-oriented programs, the question arises whether it is also well suited for testing aspect-oriented programs.

To focus the discussion, the authors distinguish between the "application aspects", i.e. aspects applying to the base program for some application-specific purpose, and "testing aspects", i.e. aspect applying to the base program or application aspects for the purpose of testing them. With the aid of our type levels, one can syntactically separate these levels, by modifying testing aspects with `meta`. Also, our type-safeness opens the door for the extension of reflection mechanisms requested in [14] "so that information about actual join points, pointcuts and advices can be obtained", without worrying about new problems (*cf.* Section 4.6).

6. CONCLUSION

As a form of meta-programming, aspects and AOP are so powerful concepts that their use must be regulated. In particular, possible self-application of aspects is a severe problem, since it cannot only cause infinite recursion, but also allows nonsensical expressions. While the former should merely be prevented, the latter must be forbidden by any sound language definition. Based on the groundbreaking work of Russell and Tarski, we have proposed a simple language extension that equips AspectJ with a (formerly unavailable) typing system suitable to eliminate both kinds of problems through a simple static type check. Even though we based our argumentation on one specific implementation of AOP, the problem and the solution presented in this paper should be applicable to AOP in general.

7. REFERENCES

[1] AspectJ homepage, http://www.aspectj.org, 2006

[2] P. Avgustinov et al: "Optimising AspectJ", *Technical Report*, 2004

[3] http://www.eclipse.org/aspectj/doc/released/progguide/

[4] R.E. Filman et al: "Aspect-Oriented Software Development", *Addison-Wesley*, 2004

[5] F. Forster: "Points-To Analyse und darauf basierende Interferenzanalyse für eine Kernsprache von AspectJ", *Master Thesis*, University of Passau, 2005

[6] J.D. Gradecki, N. Lesiecki: "Mastering AspectJ", *John Wiley & Sons*, 2003

[7] K.B. Graversen and K. Østerbye: "Aspects of Aspects – a framework for discussion", *European Interactive Workshop on Aspects in Software*, 2004

[8] D.R. Hofstadter: "Gödel, Escher, Bach: an Eternal Golden Braid", *Basic Books Inc.*, 1979.

[9] http://www.jcp.org/en/jsr/detail?id=269

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold: "An overview of AspectJ", *Proceedings of ECOOP '01*, 2001

[11] G. Kiczales et al. : "Aspect-oriented programming", *Proceedings of ECOOP '97*, 1997

[12] http://en.wikipedia.org/wiki/Metaprogramming_(programming)

[13] B. Russell: "Mathematical Logic as Based on the Theory of Types", *American Journal of Mathematics Vol. 30*, 222-262, 1908.

[14] D. Sokenou and S. Herrmann: "Aspects for Testing Aspects?", *Workshop on Testing Aspect Oriented-Programms AOSD 2005*, 2005

[15] F. Steimann: "Domain Models are Aspect Free", *MoDELS/UML 2005 (Springer 2005)*, 171–185, 2005

[16] http://java.sun.com/j2se/1.5.0/docs/guide/apt/GettingStarted.html

[17] A. Tarski: "The Semantic Conception of Truth and the Foundations of Semantics", *Philosophy and Phenomenological Research 4*, 341-375, 1944

Interference of Larissa Aspects

David Stauch
Verimag/INPG, Grenoble
Centre Equation – 2, avenue
de Vignate, F 38610 GIERES
David.Stauch@imag.fr

Karine Altisen
Verimag/INPG, Grenoble
Centre Equation – 2, avenue
de Vignate, F 38610 GIERES
Karine.Altisen@imag.fr

Florence Maraninchi
Verimag/INPG, Grenoble
Centre Equation – 2, avenue
de Vignate, F 38610 GIERES
Florence.Maraninchi@imag.fr

ABSTRACT

Aspect Oriented Programming is a programming language concept for expressing cross-cutting concerns. A key point when dealing with aspects is the notion of interference. Applying several aspects to the same program may lead to unintended results because of conflicts between the aspects. In this paper, we study the notion of interference for Larissa, a formally defined language. Larissa is the aspect extension of Argos, a StateChart-like automata language designed to program reactive systems. We present a way to weave several aspects in a less conflict-prone manner, and a means to detect remaining conflicts statically, at a low complexity.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Theory, Verification

Keywords

reactive systems, aspect-oriented programming, formal semantics, synchronous languages, aspect interference

1. INTRODUCTION

Aspect oriented programming (AOP). AOP has emerged recently. It aims at providing new facilities to implement or modify existing programs: it may be the case that implementing some new functionality or property in a program P can not be done by adding a new module to the existing structure of P but rather by modifying every module in P. This kind of functionality or property is then called an *aspect*. AOP provides a way to define aspects separately from the rest of the program and then to introduce or "weave" them automatically into the existing structure. Many programming languages have been, now, extended with aspects.

Larissa. In this paper, we study the notion of aspect interferences for a formally defined language called Larissa. Larissa [1] is an aspect extension for Argos [13], a language used to program reactive systems. Argos pertains to the synchronous languages family; it is based on Mealy machines that communicate via Boolean signals, plus a simple set of atomic operators on the machines. The semantics of Argos programs is formally defined. Larissa defines aspects that are woven into Argos programs. The selection of join points is based on temporal pointcuts and the advice is some modification of the transitions of the basic machines. The weaving of Larissa aspects preserves the equivalence between programs.

Interferences. A key point when dealing with aspects is the notion of interferences. If A_1 and A_2 are aspects, and weaving first A_1 and then A_2 yields a different program than weaving first A_2 and then A_1, A_1 and A_2 are said to interfere. Aspect interference may depend on how the weaver proceeds: if it sequentially weaves first A_1 into a program P and then A_2 into the result — denoted by $P \triangleleft A_1 \triangleleft A_2$ — or if it weaves A_1 and A_2 together into P — denoted by $P \triangleleft \{A_1, A_2\}$.

In general, sequential weaving often causes interference. This may be one reason why languages such as AspectJ do not proceed sequentially. As an explanation, let us look at the following example. The class `Test` has a method `foo` and a method `main`, which calls `foo` on some `Test` object.

```
class Test {
  public void foo () { ... }
  public static void main (String [] args)
    { (new Test ()).foo (); }              }
```

We then define the aspect `A1` that has a method `bar()` and adds a call to `bar` at the end of every call to every method named `foo`.

```
aspect A1 {
    void bar () { ... }
    after(): call(* foo (..)) { bar ();}   }
```

After compiling together the class `Test` and the aspect `A1` (`Test`$\triangleleft$ `A1`), the execution of `main` executes `foo ()` and then `bar ()`. Let us introduce another aspect `A2` which adds some code at the end of every method named `bar`.

```
aspect A2 { after(): call(* bar (..)) { XXX } }
```

FOAL '06 Bonn, Germany

If the class `Test` is compiled with `A2` only (`Test`$\triangleleft$ `A2`), nothing changes (the class `Test` is unchanged, since no method `bar` exists, we have `Test` = `Test`$\triangleleft$ `A2`).

Now imagine that a weaver for AspectJ produces Java code as a backend, and that for weaving two aspects, we first weave the first one, obtain some Java code, and weave the second aspect into the result. We call this sequential weaving of aspects. Larissa works this way: as the other Argos operators, aspect weaving is defined as the transformation of an Argos program into another Argos program.

Sequentially weaving `A1` into `Test` and then `A2` into the result provides a different program from weaving first `A2` and then `A1`. If we execute the `main` method in both cases, (`Test`$\triangleleft$ `A1`)$\triangleleft$ `A2` executes `foo`, `bar` and then the code `XXX` added by `A2`, whereas (`Test`$\triangleleft$ `A2`)$\triangleleft$ `A1` only executes `foo` and `bar`. `A2` is activated in the first case, but not in the second.

AspectJ does not work this way. Join points are defined as points in the execution of the woven program, including those contained in advice. Thus, aspects affect each other, and cannot be woven sequentially. They must be woven together, i.e., for the example, `Test`$\triangleleft$ {`A1`, `A2`}. In the example, this produces the same result as (`Test`$\triangleleft$ `A1`)$\triangleleft$ `A2`.

In this paper we propose a weaving mechanism for Larissa that weaves several aspects together into a program, and thus eliminates some cases of interference. As opposed to AspectJ, pointcuts do not capture join points in the woven program, but in the base program. In Larissa, advice only affects the base program, whereas in AspectJ, advice also affects advice.

Aspects in AspectJ may still interfere. This is illustrated by the second example:

```
aspect A3 {
  declare precedence : A4, A3;     // (**)
  before(): call(* foo (..)) { ... }  }
aspect A4 {
  before(): call(* foo (..)) { ... }  }
```

The sets of join points selected by `A3` and by `A4` are the same. The interference here is unavoidable since the advice programs have to be executed sequentially. In such a case, AspectJ allows to describe the order of application of the advice (see line `(**)`).

Likewise, aspects in Larissa may still interfere if they share join points, even if they are woven together into a program. We further analyze interference for aspects that are woven together. We present sufficient conditions to prove non-interference, either for two aspects in general or two aspects and a specific program.

Section 2 presents the Argos language and the Larissa extension, Section 3 illustrates the language on an example, Section 4 deals with interferences for Larissa, Section 5 explores some related work and Section 6 gives some conclusions and perspectives.

2. LANGUAGE

This section presents a restriction of the Argos language [13], and the Larissa extension [1]. The Argos language is defined as a set of operators on complete and deterministic input/output automata communicating via Boolean signals. The semantics of an Argos program is given as a trace semantics that is common to a wide variety of reactive languages.

2.1 Traces and trace semantics

Definition 1 (Traces) *Let $\mathcal{I}$, $\mathcal{O}$ be sets of Boolean input and output variables representing signals from and to the environment. An* input trace*, it, is a function: it* : $\mathbb{N} \longrightarrow [\mathcal{I} \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$. *An* output trace*, ot, is a function: ot* : $\mathbb{N} \longrightarrow [\mathcal{O} \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$. *We denote by $InputTraces$ (resp. $OutputTraces$) the set of all input (resp. output) traces. A pair (it, ot) of input and output traces (i/o-traces for short) provides the valuations of every input and output at each instant $n \in \mathbb{N}$. We denote by $it(n)[i]$ (resp. $ot(n)[o]$) the value of the input $i \in \mathcal{I}$ (resp. the output $o \in \mathcal{O}$) at the instant $n \in \mathbb{N}$.*

A set of pairs of i/o-traces $S = \{(it, ot) \mid it \in InputTraces \wedge ot \in OutputTraces\}$ is deterministic *iff* $\forall (it, ot), (it', ot') \in S \,.\, (it = it') \implies (ot = ot')$.

A set of pairs of i/o traces $S = \{(it, ot) \mid it \in InputTraces \wedge ot \in OutputTraces\}$ is complete *iff* $\forall it \in InputTraces \,.\, \exists ot \in OutputTraces \,.\, (it, ot) \in S$.

A set of traces is a way to define the semantics of an Argos program P, given its inputs and outputs. From the above definitions, a program P is *deterministic* if from the same sequence of inputs it always computes the same sequence of outputs. It is *complete* whenever it allows every sequence of every eligible valuations of inputs to be computed. Determinism is related to the fact that the program is indeed written with a programming language (which has deterministic execution); completeness is an intrinsic property of the program that has to react forever, to every possible inputs without any blocking.

2.2 Argos

The core of Argos is made of input/output automata, the synchronous product, and the encapsulation.

The synchronous product allows to put automata in parallel which synchronize on their common inputs. The encapsulation is the operator that expresses the communication between automata with the synchronous broadcast: if two automata are put in parallel, they can communicate via some signal s. This signal is an input of the first automaton and an output of the second. The encapsulation operator computes this communication and then hides the signal s.

The semantics of an automaton is defined by a set of traces, and the semantics of the operators is given by translating expressions into flat automata.

Definition 2 (Automaton) *An* automaton $\mathcal{A}$ *is a tuple $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ where $\mathcal{Q}$ is the set of states, $s_{init} \in \mathcal{Q}$ is the initial state, $\mathcal{I}$ and $\mathcal{O}$ are the sets of Boolean input and output variables respectively, $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ is the set of transitions. $\mathcal{B}ool(\mathcal{I})$ denotes the set of Boolean formulas with variables in $\mathcal{I}$. For $t = (s, \ell, O, s') \in \mathcal{T}$, $s, s' \in \mathcal{Q}$ are the source and target states, $\ell \in \mathcal{B}ool(\mathcal{I})$ is the triggering condition of the transition, and $O \subseteq \mathcal{O}$ is the set of outputs emitted whenever the transition is triggered. Without loss of generality, we consider that automata only have complete monomials as input part of the transition labels.*

Complete monomials are conjunctions that for each $i \in \mathcal{I}$ contain either i or $\bar{i}$. Requiring complete monomials as input labels makes the definition of the operators easier. A transition with an arbitrary input label can be easily converted in a set of transitions with complete monomials, and

can thus be considered as a macro notation. We will use such transitions in the examples.

The *semantics* of an automaton $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ is given in terms of a set of pairs of i/o-traces. This set is built using the following functions:

$$S\_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times \mathbb{N} \longrightarrow \mathcal{Q}$$
$$O\_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times \mathbb{N} \setminus \{0\} \longrightarrow 2^{\mathcal{O}}$$

$S\_step(s, it, n)$ is the state reached from state s after performing n steps with the input trace it; $O\_step(s, it, n)$ are the outputs emitted at step n:

$$n = 0 : \; S\_step_{\mathcal{A}}(s, it, n) = s$$
$$n > 0 : \; S\_step_{\mathcal{A}}(s, it, n) = s' \qquad O\_step_{\mathcal{A}}(s, it, n) = O$$
$$\text{where } \exists (S\_step_{\mathcal{A}}(s, it, n-1), \ell, O, s') \in \mathcal{T}$$
$$\wedge\ \ell \text{ has value true for } it(n-1)\ .$$

We note $Traces(\mathcal{A})$ the set of all traces built following this scheme: $Traces(\mathcal{A})$ defines the semantics of $\mathcal{A}$. The automaton $\mathcal{A}$ is said to be *deterministic* (resp. *complete*) iff its set of traces $Traces(\mathcal{A})$ is deterministic (resp. complete) (see Definition 1). Two automata $\mathcal{A}_1$, $\mathcal{A}_2$ are *trace-equivalent*, noted $\mathcal{A}_1 \sim \mathcal{A}_2$, iff $Traces(\mathcal{A}_1) = Traces(\mathcal{A}_2)$.

Definition 3 (Synchronous Product) *Let* $\mathcal{A}_1 = (\mathcal{Q}_1, s_{init1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1)$ *and* $\mathcal{A}_2 = (\mathcal{Q}_2, s_{init2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2)$ *be automata. The* synchronous product *of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *is the automaton* $\mathcal{A}_1 \| \mathcal{A}_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, (s_{init1} s_{init2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T})$ *where* $\mathcal{T}$ *is defined by:*

$$(s_1, \ell_1, O_1, s_1') \in \mathcal{T}_1 \wedge (s_2, \ell_2, O_2, s_2') \in \mathcal{T}_2 \Longleftrightarrow$$
$$(s_1 s_2, \ell_1 \wedge \ell_2, O_1 \cup O_2, s_1' s_2') \in \mathcal{T}\ .$$

The synchronous product of automata is both commutative and associative, and it is easy to show that it preserves both determinism and completeness.

Definition 4 (Encapsulation) *Let* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ *be a set of inputs and outputs of* $\mathcal{A}$. *The* encapsulation *of* $\mathcal{A}$ *w.r.t.* Γ *is the automaton* $\mathcal{A} \setminus \Gamma = (\mathcal{Q}, s_{init}, \mathcal{I} \setminus \Gamma, \mathcal{O} \setminus \Gamma, \mathcal{T}')$ *where* $\mathcal{T}'$ *is defined by:*

$$(s, \ell, O, s') \in \mathcal{T} \wedge \ell^+ \cap \Gamma \subseteq O \wedge \ell^- \cap \Gamma \cap O = \emptyset \Longleftrightarrow$$
$$(s, \exists \Gamma\ .\ \ell, O \setminus \Gamma, s') \in \mathcal{T}'$$

ℓ^+ *is the set of variables that appear as positive elements in the monomial* ℓ *(i.e.* $\ell^+ = \{x \in \mathcal{I} \mid (x \wedge \ell) = \ell\}$*).* ℓ^- *is the set of variables that appear as negative elements in the monomial* l *(i.e.* $\ell^- = \{x \in \mathcal{I} \mid (\neg x \wedge \ell) = \ell\}$*).*

Intuitively, a transition $(s, \ell, O, s') \in \mathcal{T}$ is still present in the result of the encapsulation operation if its label satisfies a local criterion made of two parts: $\ell^+ \cap \Gamma \subseteq O$ means that a local variable which needs to be true has to be emitted by the same transition; $\ell^- \cap \Gamma \cap O = \emptyset$ means that a local variable that needs to be false should *not* be emitted in the transition.

If the label of a transition satisfies this criterion, then the names of the encapsulated variables are hidden, both in the input part and in the output part. This is expressed by $\exists \Gamma\ .\ \ell$ for the input part, and by $O \setminus \Gamma$ for the output part.

In general, the encapsulation operation does not preserve determinism nor completeness. This is related to the so-called "causality" problem intrinsic to synchronous languages (see, for instance [4]).

An example

Figure 1 (a) shows a 3-bits counter. Dashed lines denote parallel compositions and the overall box denotes the encapsulation of the three parallel components, hiding signals `b` and `c`. The idea is the following: the first component on the right receives `a` from the environment, and sends `b` to the second one, every two `a`'s. Similarly, the second one sends `c` to the third one, every two `b`'s. `b` and `c` are the carry signals. The global system has `a` as input and `d` as output; it counts `a`'s modulo 8, and emits `d` every 8 `a`'s. Applying the semantics of the operator (first the product of the three automata, then the encapsulation) yields the simple flat automaton with 8 states (Figure 1 (b)).

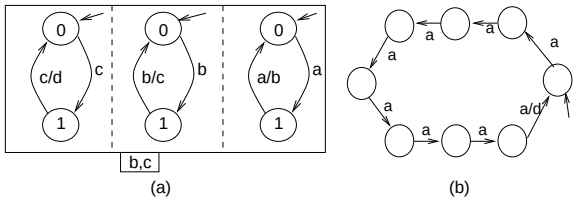


Figure 1: A 3-bits counter. Notations: in each automaton, the initial state is denoted with a little arrow; the label on transitions are expressed by "`triggering cond. / outputs emitted`", e.g. the transition labelled by "`a/b`" is triggered when `a` is true and emits `b`.

2.3 Larissa

Argos operators are already powerful. However, there are cases in which they are not sufficient to modularize all concerns of a program: some small modifications of the global program's behavior may require that we modify all parallel components, in a way that is not expressible with the existing operators.

Therefore, we proposed Larissa [1], an aspect-oriented extension for Argos, which allows the modularization of a number of recurrent problems in reactive programs. As most aspect languages, Larissa contains a pointcut and an advice construct. The pointcut selects a number of transitions, called *join point transitions*, from an automaton, and the advice modifies these transitions, both their outputs and their target state.

In this paper, we only present a simple kind of advice, in which the target state is the same for all join point transitions. In [1, 2] we present more sophisticated kinds of advice, which allow to jump forward or backward from the join point transition, and to replace join point transitions with complete automata. We believe that the ideas presented in this paper can be extended easily to the other kinds of advice, as the join point mechanism is the same.

We ensure the semantic properties that make it possible to introduce aspects as a normal operator into Argos. Specifically, as shown in [1], determinism and completeness are preserved, as well as semantic equivalence between programs: when we apply the same aspect to two trace-equivalent programs, we obtain two trace-equivalent programs.

Specifying pointcuts. Because we want to preserve trace equivalence, we cannot express pointcuts in terms of the internal structure of the base program. For instance, we do not allow pointcuts to refer explicitly to state names (as AspectJ [9] can refer to the name of a private method). As a consequence, pointcuts may refer to the observable behavior of the program only, i.e., its inputs and outputs. In the family of synchronous languages, where the communication between parallel components is the synchronous broadcast, *observers* [8] are a powerful and well-understood mechanism which may be used to describe pointcuts. An observer is a program that may observe the inputs and the outputs of the base program, without modifying its behavior, and compute some safety property (in the sense of safety/liveness properties as defined in [11]).

We use an observer P_{JP} that emits a special output `JP` to describe a pointcut. Whenever P_{JP} emits `JP`, "we are in" a join point, and the woven program executes the advice.

Join Point Weaving. If we simply put a program P and an observer P_{JP} in parallel, P's outputs $\mathcal{O}$ will become synchronization signals between them, as they are also inputs of P_{JP}. They will be encapsulated, and are thus no longer emitted by the product. We avoid this problem by introducing a new output o' for each output o of P: o' will be used for the synchronization with P_{JP}, and o will still be visible as an output. First, we transform P into P' and P_{JP} into P'_{JP}, where $\forall o \in \mathcal{O}$, o is replaced by o'. Second, we duplicate each output of P by putting P in parallel with one single-state automaton per output o defined by: $dupl_o = (\{q\}, q, \{o'\}, \{o\}, \{(q, o', o, q)\})$. The complete product, where $\mathcal{O}$ is noted $\{o_1, ..., o_n\}$, is given by:

$$\boldsymbol{\mathcal{P}}(P, P_{JP}) = (P' \| P'_{JP} \| dupl_{o_1} \| \ ... \ \| dupl_{o_n}) \setminus \{o'_1, ..., o'_n\}$$

The program $\boldsymbol{\mathcal{P}}(P, P_{JP})$ is first transformed into the single trace-equivalent automaton by applying the definition of the operators. We use the same notation, $\boldsymbol{\mathcal{P}}(P, P_{JP})$, for the program and its transformation. Then, the join points are selected: they are the *transitions* of $\boldsymbol{\mathcal{P}}(P, P_{JP})$ that emit JP.

Specifying the advice. A piece of advice modifies the join point transitions: it redefines their target states and their outputs. The only advice presented in this paper specifies the target state of the join point transition globally, by a finite input trace σ. When executing σ on $\boldsymbol{\mathcal{P}}(P, P_{JP})$ from its initial state, this leads to some state of $\boldsymbol{\mathcal{P}}(P, P_{JP})$, *targ*. *targ* is the unique new target state for any join point transition.

Advice Weaving. Advice weaving consists in changing the target state of the join point transitions to the single state specified by the finite input trace σ, and in replacing their outputs by the *advice outputs* O_{adv}.

Definition 5 (Advice weaving) *Let* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $adv = (O_{adv}, \sigma)$ *a piece of advice, with* $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$ *a finite input trace of length* $\ell_\sigma + 1$. *The advice weaving operator,* $\lhd_{JP}$, *weaves asp on* $\mathcal{A}$ *and returns the automaton* $\mathcal{A} \lhd_{JP} adv = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O} \cup O_{adv}, \mathcal{T}')$, *where* $\mathcal{T}'$ *is defined as follows, with* $targ = S\_step_{\mathcal{A}}(s_{init}, \sigma, \ell_\sigma)$ *being the new target state:*

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}' \quad (1)$$

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \in O\big) \implies (s, \ell, O_{adv}, targ) \in \mathcal{T}' \quad (2)$$

Transitions (1) are not join point transitions and are left unchanged. Transitions (2) are the join point transitions, their final state *targ* is specified by the finite input trace σ. $S\_step_{\mathcal{A}}$ (which has been naturally extended to finite input traces) executes the trace during ℓ_σ steps, from the initial state of $\mathcal{A}$.

General aspect definition. Putting together the pointcut and the advice, we define an aspect as follows:

Definition 6 (Larissa aspect) *An aspect, for a program* P *on inputs* $\mathcal{I}$ *and outputs* $\mathcal{O}$, *is a tuple* (P_{JP}, adv) *where*

- $P_{JP} = (\mathcal{Q}_{pc}, s_{init}, \mathcal{I} \cup \mathcal{O}, \{JP\} \cup O_{pc}, \mathcal{T}_{pc})$ *is the pointcut program, and JP occurs nowhere else in the environment.*
- $adv = (O_{adv}, \sigma)$ *is the advice, which contains two items:*
  - O_{adv} *is the set of outputs emitted by the advice transitions, which may contain fresh variables as well as elements of* $\mathcal{O}$.
  - $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$ *is a finite input trace of length* $\ell_\sigma + 1$. *It defines the single target state of the advice transitions by executing the trace from the initial state.*

An aspect is woven into a program by first determining the join point transitions and then weaving the advice.

Definition 7 (Aspect weaving) *Let* P *be a program and* $asp = (P_{JP}, adv)$ *an aspect for* P. *The weaving of asp on* P *is defined as follows:*

$$P \lhd asp = \boldsymbol{\mathcal{P}}(P, P_{JP}) \lhd_{JP} adv \ .$$

3. EXAMPLE

As an example, we present a simplified view of the interface of a complex wristwatch, implemented with Argos and Larissa. The full case study was presented in [2]. The interface is a modified version of the Altimax[1] model by Suunto[1].

3.1 The Watch

The Altimax wristwatch has an integrated altimeter, a barometer and four buttons, the `mode`, the `select`, the `plus`, and the `minus` button. Each of the main functionalities (time keeping, altimeter, barometer) has an associated main mode, which displays information, and a number of submodes, where the user can access additional functionalities. An Argos program that implements the interface of the watch is shown in Figure 2. For better readability, only those state names, outputs and transitions we will refer to are shown.

In a more detailed model (as in [2]) the submode states would contain behavior using the refinement operator of Argos (see [13] for a definition). We choose not to present this

[1]Suunto and Altimax are trademarks of Suunto Oy.

operator in this paper since we do not need it to define aspect weaving. Adding refinement changes nothing for the weaving definition, as it works directly on the transformation of the program into a single trace-equivalent automaton. For the same reason, the interference analysis presented in Section 4 is also the same.

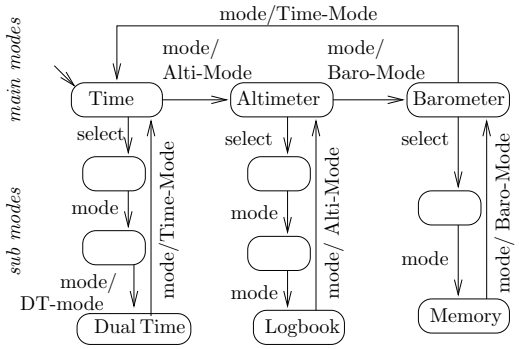


Figure 2: The Argos program for the Altimax watch.

The buttons of the watch are the inputs of the program. The `mode` button circles between modes, the `select` button selects the submodes. There are two more buttons: the `plus` and the `minus` button which modify current values in the submodes, but their effect is not shown in the figure. The buttons have different meanings depending on the mode in which the watch is currently.

The interface component we model here interprets the meaning of the buttons the user presses, and then calls a corresponding function in an underlying component. The outputs are commands to that component. E.g., whenever the program enters the Time Mode, it emits the output `Time-Mode`, and the underlying component shows the time on the display of the watch.

3.2 Two Shortcut Aspects

The `plus` and the `minus` buttons have no function consistent with their intended meaning in the main modes: there are no values to increase or decrease. Therefore, they are given a different function in the main modes: when one presses the `plus` or the `minus` button in a main mode, the watch goes to a certain submode. The role of the `plus` and `minus` buttons in the main modes are called *shortcuts* since it allows to quickly activate a functionnality, which would have needed, otherwise, a long sequence of buttons.

Pressing the `plus` button in a main mode activates the `logbook` function of the altimeter, and pressing the `minus` button activates the 4-day `memory` of the barometer. These functions are quite long to reach without the shortcuts since the logbook is the third submode of the altimeter, and the 4-day memory is the second submode of the barometer.

These shortcuts can be implemented easily with Larissa aspects. Figure 3 (a) shows the pointcut for the logbook aspect, and Figure 3 (b) the pointcut for the memory aspect. In both pointcuts, state `main` represents the main modes and state `sub` represents the submodes. When, in a main mode, `plus` (resp. `minus`) is pressed, the pointcut emits JP_l (resp. JP_m), thus the corresponding advice is executed; when `select` is pressed, the pointcut goes to the `sub` state, so as to record that the shortcuts are no longer active and

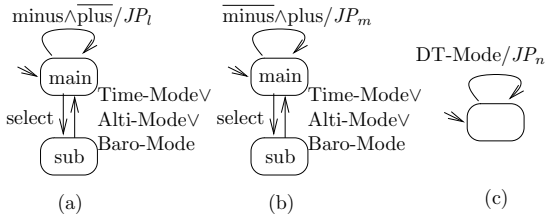


Figure 3: The pointcuts for the aspects.

that the `plus` and `minus` buttons have their usual meaning. As advice, we specify the trace that leads to the functionality we want to reach, i.e. σ_l =`mode.select.mode.mode` for the logbook aspect and σ_m =`mode.mode.select.mode` for the 4-day memory aspect, and the output that tells the underlying component to display the corresponding information.

3.3 The No-DTM Aspect

We want to reuse the interface program of the Altimax to build the interface of another wristwatch, which differs from the Altimax in that it has no Dual-Time mode (third submode of the main mode Time). Therefore, we write an aspect that removes the Dual-Time mode from the interface: all incoming transitions are redirected to another state. Figure 3 (c) shows the pointcut for the No-DTM aspect. It selects all transitions that emit `DT-Mode`, the output that tells the underlying component to show the information corresponding to the Dual-Time mode on the display. Because the Dual-Time mode is the last submode of the Time mode, we want the join point transitions to point to the Time main mode, i.e. the initial state of the program. Thus, as advice, we specify `Time-Mode` as output and an empty trace, which points to the Time main mode.

4. INTERFERENCE

This section identifies problems that occur when several aspects are applied to a program, and, as a solution, proposes to weave several aspects at the same time. A mechanism to prove that no interferences remain is also proposed.

4.1 Applying Several Aspects

If we apply first the logbook aspect and then, sequentially, the memory aspect to the watch program, the aspects do not behave as we would expect. If, in the woven program, we first press the `minus` button in a main mode, thus activating the logbook aspect, and then the `plus` button, the memory aspect is activated, although we are in a sub mode. This behavior was clearly not intended by the programmer of the memory aspect.

The problem is that the memory aspect has been written for the program without the logbook aspect: the pointcut assumes that the only way to leave a main mode is to press the `select` button. However, the logbook aspect invalidates that assumption by adding transitions from the main modes to a submode. When these transitions are taken, the pointcut of the memory aspect incorrectly assumes that the program is still in a main mode.

Furthermore, for the same reason, applying first the memory aspect and then the logbook aspect produces (in terms of trace-equivalence) a different program from applying first the logbook aspect and then the memory aspect:

`watch` $\triangleleft$ `logbook` $\triangleleft$ `memory` $\nsim$ `watch` $\triangleleft$ `memory` $\triangleleft$ `logbook`.

As a first attempt to define *aspect interference*, we say that two aspects $\mathcal{A}_1$ and $\mathcal{A}_2$ interfere when their application on a program P in different orders does not yield two trace-equivalent programs: $P \triangleleft \mathcal{A}_1 \triangleleft \mathcal{A}_2 \nsim P \triangleleft \mathcal{A}_2 \triangleleft \mathcal{A}_1$. We say that two aspects that do not interfere are *independent*.

With interfering aspects, the aspect that is woven second must know about the aspect that was applied first. To be able to write aspects as the ones above independently from each other, we propose a mechanism to weave several aspects at the same time. The idea is to first determine the join point transitions for all the aspects, and then apply the advice.

Definition 8 (Joint weaving of several aspects) *Let $\mathcal{A}_1 \ldots \mathcal{A}_n$ be some aspects, with $\mathcal{A}_i = (P_{JP_i}, adv_i)$, and P a program. We define the application of $\mathcal{A}_1 \ldots \mathcal{A}_n$ on P as follows:*

$$P \triangleleft (\mathcal{A}_1, \ldots, \mathcal{A}_n) = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \triangleleft_{JP_n} adv_n \ldots \triangleleft_{JP_1} adv_1$$

Jointly weaving the logbook and the memory aspect leads to the intended behavior, and the weaving order does not influence the result, because both aspects first select their join point transitions in the main modes, and change the target states of the join point transitions only afterwards.

Note that Definition 8 does not make sequential weaving redundant. We still need to weave aspects sequentially in some cases, when the second aspects must be applied to the result of the first. For instance, imagine an aspect that adds an additional main mode to the watch (with a kind of advice not presented in this paper). Then, the shortcut aspects must be sequentially woven *after* this aspect, so that they can select the new main mode as join point.

Definition 8 does not solve all conflicts. Indeed, the $\mathcal{A}_i$ in $P \triangleleft (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ do not commute, in general, since the advice weaving is applied sequentially. We define aspect interference for the application of several aspects.

Definition 9 (Aspect Interference) *Let $\mathcal{A}_1$ ... $\mathcal{A}_n$ be some aspects, and P a program. We say that $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$* interfere *for P iff*

$$P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_i, \mathcal{A}_{i+1} \ldots \mathcal{A}_n) \nsim P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_{i+1}, \mathcal{A}_i \ldots \mathcal{A}_n)$$

As an example for interfering aspects, assume that the condition of the join point transition of the pointcut of the logbook aspect (Figure 3 (a)) is only `minus` and the condition of the join point transition of the pointcut of the logbook aspect (Figure 3 (b)) is only `plus`. In this case, the two aspects share some join point transitions, namely when both buttons are pressed at the same time in a main mode. Both aspects then want to execute their advice, but only one can, thus they interfere. Only the aspect that was applied last is executed.

In such a case, the conflict should be made explicit to the programmer, so that it can be solved by hand. Here, it was resolved by changing the pointcuts to the form they have in Figure 3, so that neither aspect executes when both buttons are pressed.

4.2 Proving Non-Interference

In this section, we show that in some cases, non-interference of aspects can be proven, if the aspects are woven jointly, as defined in Definition 8. We can prove non-interference of two given aspects either for any program, or for a given program. Following [6], we speak of *strong independence* in the first case, and of *weak independence* in the second.

We use the operator *advTrans* to determine interference between aspects. It computes all the join point transitions of an automaton, i.e. all transitions with a given output *JP*.

Definition 10 *Let $A = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and $JP \in \mathcal{O}$. Then,*

$$advTrans(A, JP) = \{t | t = (s, \ell, O, s') \in \mathcal{T} \wedge JP \in O\} .$$

The following theorem proves strong independence between two aspects.

Theorem 1 (Strong Independence) *Let $\mathcal{A}_1 \ldots \mathcal{A}_n$ be some aspects, with $\mathcal{A}_i = (P_{JP_i}, adv_i)$. Then, the following equation holds:*

$$\begin{aligned} &advTrans(P_{JP_i} \| P_{JP_{i+1}}, JP_i) \\ &\cap\, advTrans(P_{JP_i} \| P_{JP_{i+1}}, JP_{i+1}) = \emptyset \\ &\Rightarrow P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_i, \mathcal{A}_{i+1} \ldots \mathcal{A}_n) \sim P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_{i+1}, \mathcal{A}_i \ldots \mathcal{A}_n) \end{aligned}$$

See appendix A for a proof. Theorem 1 states that if there is no transition with both JP_i and JP_{i+1} as outputs in the product of P_{JP_i} and $P_{JP_{i+1}}$, $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ are independent and thus can commute while weaving their advice. Theorem 1 defines a sufficient condition for non-interference, by looking only at the pointcuts. When the condition holds, the aspects are said to be *strongly independent*.

Theorem 2 (Weak Independence) *Let $\mathcal{A}_1 \ldots \mathcal{A}_n$ be some aspects, with $\mathcal{A}_i = (P_{JP_i}, adv_i)$, and $P_{pc} = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n})$. Then, the following equation holds:*

$$\begin{aligned} &advTrans(P_{pc}, JP_i) \cap advTrans(P_{pc}, JP_{i+1}) = \emptyset \\ &\Rightarrow P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_i, \mathcal{A}_{i+1} \ldots \mathcal{A}_n) \sim P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_{i+1}, \mathcal{A}_i \ldots \mathcal{A}_n) \end{aligned}$$

See appendix B for a proof. Theorem 2 states that if there is no transition with both JP_i and JP_{i+1} as outputs in P_{pc}, $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ do not interfere. This is weaker than Theorem 1 since it also takes the program P into account. However, there are cases in which the condition of Theorem 1 is false (thus it yields no results), but Theorem 2 allows to prove non-interference. See Section 4.4 for an example.

Theorem 2 is a sufficient condition, but, as Theorem 1, it is not necessary: it may not be able to prove independence for two independent aspects. The reason is that it does not take into account the effect of the advice weaving: consider two aspects such that the only reason why the condition for Theorem 2 is false is a transition sourced in some state s, and such that s is only reachable through another join point transition; if the advice weaving makes this state unreachable, then the aspects do not interfere.

The results obtained by both Theorems are quite intuitive. They mean that if the pointcut mechanism does not select any join points common to two aspects, then these aspects do not interfere. This condition can be calculated on the pointcuts alone, or can also take the program into account.

Note that the detection of non-interference is a static condition that does not add any complexity overhead. Indeed, to weave the aspects, the compiler needs to build first

$P_{JP_1} \| \ldots \| P_{JP_n} = P_{\text{all } JP}$: the condition of Theorem 1 can be checked during the construction of $P_{\text{all } JP}$. Second, the weaver builds $P_{\text{pc}} = \boldsymbol{\mathcal{P}}(P, P_{\text{all } JP})$. Afterwards, it can check the condition of Theorem 2. Thus, to calculate the conditions of both Theorems, it is sufficient to check the outputs of the transitions of intermediate products during the weaving. The weaver can easily emit a warning when a potential conflict is detected.

To have an exact characterization of non-interference, it is still possible to compute the predicate $P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_i, \mathcal{A}_{i+1} \ldots \mathcal{A}_n) \sim P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_{i+1}, \mathcal{A}_i \ldots \mathcal{A}_n)$, but calculating semantic equality is very expensive for large programs.

Note that the interference presented here only applies to the joint weaving of several aspects, as defined in Definition 8. Sequentially woven aspects may interfere even if their join points are disjoint, because the pointcut of the second aspects applies to the woven program. A similar analysis to prove non-interference of sequential weaving would be more difficult, because the effect of the advice must be taken into account. Moreover, it is not clear when such an analysis makes sense: sequential weaving should be used only if one aspect depends on the other, and interference is unavoidable.

4.3 Interference between the Shortcut Aspects

Figure 4 (a) shows the product of the pointcuts of the logbook and the memory aspect. There are no transitions that emit both JP_l and JP_m, thus, by applying Theorem 1, we know that the aspects do not interfere, independently of the program they are applied to.

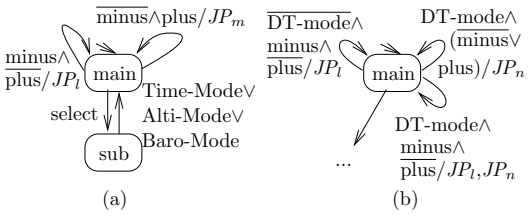


Figure 4: Interference between pointcuts.

Let us assume again that the condition of the join point transition of the pointcut of the logbook aspect (Figure 3 (a)) is only `minus` and the condition of the join point transition of the pointcut of the logbook aspect (Figure 3 (b)) is only `plus`. In this case, the state main in Figure 4 (a) would have another loop transition, with label `minus∧plus/`JP_l`,`JP_m. Thus, Theorem 1 not only states that the aspects potentially interfere, but it also gives a means to determine where: here, the problem is that when both `minus` and `plus` are pressed in a main mode, at the same time, both aspects are activated. Larissa thus emits a warning and the user is invited to solve the conflict if needed.

4.4 Interference between a Shortcut and the No-DTM Aspect

Figure 4 (b) shows the initial state of the product of the pointcuts of the logbook (Figure 3 (a)) and the No-DTM aspect (Figure 3 (c)). There is a transition that has both JP_l and JP_n as outputs. Theorem 1 states that the aspects may interfere, but when applied to the wristwatch controller, they do not. This is because the `DT-mode` is an output of the controller and is never emitted when the watch is in a main mode, where the logbook aspect can be activated. As the `DT-mode` is always false in the main modes, the conflicting transition is never enabled. When applied to another program, however, the aspects may interfere.

In this example, the use of Theorem 2 is thus needed to show that the aspects do not interfere when applied to the wristwatch controller. Its condition is true, as expected, because JP_l is only emitted in the main modes, and JP_n only in the Time submodes.

5. RELATED WORK

Some authors discuss the advantages of sequential vs. joint weaving. Lopez-Herrejon and Batory [12] propose to use sequential weaving for incremental software development. Colyer and Clement [5, Section 5.1] want to apply aspects to bytecode which already contains woven aspects. In AspectJ, this is impossible because the semantics would not be the same as weaving all aspects at the same time.

Sihman and Katz [15] propose SuperJ, a superimposition language which is implemented through a preprocessor for AspectJ. They propose to combine superimpositions into a new superimposition, either by sequentially applying one to the other, or by combining them without mutual influence. Superimpositions contain assume/guarantee contracts, which can be used to check if a combination is valid.

A number of authors investigate aspect interference in different formal frameworks. Much of the work is devoted to determining the correct application order for interfering aspects, whereas we focus on proving non-interference.

Douence, Fradet, and Südholt [6] present a mechanism to statically detect conflicts between aspects that are applied in parallel. Their analysis detects all join points where two aspects want to insert advice. To reduce the detection of spurious conflicts, they extend their pointcuts with shared variables, and add constraints that an aspect can impose on a program. To resolve remaining conflicts, the programmer can then write powerful composition adaptors to define how the aspects react in presence of each other.

Pawlak, Duchien, and Seinturier [14] present a way to formally validate precedence orderings between aspects that share join points. They introduce a small language, CompAr, in which the user expresses the effect of the advice that is important for aspect interaction, and properties that should be true after the execution of the advice. The CompAr compiler can then check that a given advice ordering does not invalidate a property of an advice.

Durr, Staijen, Bergmans, and Aksit [7] propose an interaction analysis for Composition Filters. They detect when one aspect prevents the execution of another, and can check that a specified trace property is ensured by an aspect.

Balzarotti, Castaldo D'Ursi, Cavallaro and Monga [3] use program slicing to check if different aspects modify the same code, which might indicate interference.

Clean interfaces which take aspects into account can also help to detect interferences. E.g., aspect-aware interfaces [10] indicate where two aspects advise the same methods in a system.

6. CONCLUSION

We present an analysis for aspect interference for a simple

but significant part of Larissa. We expect that it can be applied without major modifications to the rest of Larissa. First, we introduced an additional operator which jointly weaves several aspects together into a program, closer to the way AspectJ weaves aspects. Because Larissa is defined modularly, we only had to rearrange the building steps of the weaving process. Then, we could analyze interference with a simple parallel product of the pointcuts.

When a potential conflict is detected, the user has to solve it by hand, if needed. In the examples we already studied, the conflicts were solved by simple modifications of the pointcut programs. We plan to further explore this problem, but we believe no new language construct will be needed.

It seems that the interference analysis for Larissa is quite precise, i.e. we can prove independence for most independent aspects. One reason for that are Larissa's powerful pointcuts, which describe join points statically, yet very precisely, on the level of transitions. Another reason is the exclusive nature of the advice. Two pieces of advice that share a join point transition never execute sequentially, but there is always one that is executed while the other is not. If the two pieces of advice are not equivalent, this leads to a conflict. Thus, as opposed to [6], assuming that a shared join point leads to a conflict does not introduce spurious conflicts.

In addition, because our language is connected to formal verification tools, we can check whether different aspect orderings result in trace-equivalent automata. This, however, is only possible because Argos is restricted to Boolean signals; otherwise trace-equivalence is not decidable. It would be interesting to design an approximate interference analysis for Larissa aspects in the presence of valued signals.

7. REFERENCES

[1] K. Altisen, F. Maraninchi, and D. Stauch. Aspect-oriented programming for reactive systems: a proposal in the synchronous framework. *Science of Computer Programming, Special Issue on Foundations of Aspect-Oriented Programming*, 2006. To appear.

[2] K. Altisen, F. Maraninchi, and D. Stauch. Larissa: Modular design of man-machine interfaces with aspects. In *5th International Symposium on Software Composition*, Vienna, Austria, Mar. 2006. To appear.

[3] D. Balzarotti, A. C. D'Ursi, L. Cavallaro, and M. Monga. Slicing AspectJ woven code. In G. T. Leavens, C. Clifton, and R. Lämmel, editors, *Foundations of Aspect-Oriented Languages*, Mar. 2005.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Programming*, 19(2):87–152, 1992.

[5] A. Colyer and A. Clement. Large-scale AOSD for middleware. In K. Lieberherr, editor, *AOSD-2004*, pages 56–65, Mar. 2004.

[6] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *AOSD-2004*, pages 141–150, Mar. 2004.

[7] P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In K. Gybels, M. D'Hondt, I. Nagy, and R. Douence, editors, *2nd European Interactive Workshop on Aspects in Software (EIWAS'05)*, Sept. 2005.

[8] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology, AMAST'93*, June 1993.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–353, 2001.

[10] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, 2005.

[11] L. Lamport. Proving the correctness of multiprocess programs. *ACM Trans. Prog. Lang. Syst.*, SE-3(2):125–143, 1977.

[12] R. E. Lopez-Herrejon and D. Batory. Improving incremental development in AspectJ by bounding quantification. In L. Bergmans, K. Gybels, P. Tarr, and E. Ernst, editors, *Software Engineering Properties of Languages and Aspect Technologies*, Mar. 2005.

[13] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1/3):61–92, 2001.

[14] R. Pawlak, L. Duchien, and L. Seinturier. Compar: Ensuring safe around advice composition. In *FMOODS 2005*, volume 3535 of *lncs*, pages 163–178, jan 2005.

[15] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, Sept. 2003.

APPENDIX

A. PROOF FOR THEOREM 1

Theorem 1 is a consequence of Theorem 2. We show that

$$advTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_i) \cap$$
$$advTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_{i+1}) = \emptyset$$

follows from

$$advTrans(P_{JP_i} \| P_{JP_{i+1}}, JP_i)$$
$$\cap\, advTrans(P_{JP_i} \| P_{JP_{i+1}}, JP_{i+1}) = \emptyset$$

JP_i and JP_{i+1} can only occur in P_{JP_i} and $P_{JP_{i+1}}$. Thus, if a transition that has both of them as outputs in $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n})$, there must already exist a transition with both of them as outputs in $P_{JP_i} \| P_{JP_{i+1}}$. □

B. PROOF FOR THEOREM 2

Because the parallel product is commutative $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_i} \| P_{JP_{i+1}} \| \ldots \| P_{JP_n})$ and $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_{i+1}} \| P_{JP_i} \| \ldots \| P_{JP_n})$ are the same.

Let $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \triangleleft_{JP_n} \mathrm{adv}_n \ldots \triangleleft_{JP_{i+2}} \mathrm{adv}_{i+2} = (\mathcal{Q}, s_{\mathrm{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T}) = P_{i+2}$. Then $P_{i+2} \triangleleft_{JP_{i+1}} \mathrm{adv}_{i+1}$ yields an automaton $P_{i+1} = (\mathcal{Q}, s_{\mathrm{init}}, \mathcal{I}, \mathcal{O} \cup O_{\mathrm{adv}_{i+1}}, \mathcal{T}')$, where $\mathcal{T}'$ is defined as follows:

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}'$$
$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O\big) \implies$$
$$(s, \ell, O_{\mathrm{adv}_{i+1}}, S\_step_{P'}(s_{\mathrm{init}}, \sigma_{i+1}, l_{\sigma_{i+1}})) \in \mathcal{T}'$$

and $P_{i+1} \lhd_{JP_i} \text{adv}_i$ yields an automaton $P_i = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O} \cup O_{\text{adv}_{i+1}} \cup O_{\text{adv}_i}, \mathcal{T}'')$, where $\mathcal{T}''$ is defined as follows:

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \notin O \wedge JP_i \notin O\big) \implies (s,\ell,O,s') \in \mathcal{T}' \tag{3}$$

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \notin O\big) \implies (s,\ell,O_{\text{adv}_{i+1}}, S\_step_{P'}(s_{\text{init}},\sigma_{i+1},l_{\sigma_{i+1}})) \in \mathcal{T}' \tag{4}$$

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \notin O \wedge JP_i \in O\big) \implies (s,\ell,O_{\text{adv}_i}, S\_step_{P'}(s_{\text{init}},\sigma_i,l_{\sigma_i})) \in \mathcal{T}' \tag{5}$$

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \in O\big) \implies (s,\ell,O_{\text{adv}_{i+1}}, S\_step_{P'}(s_{\text{init}},\sigma_{i+1},l_{\sigma_{i+1}})) \in \mathcal{T}' \tag{6}$$

If we calculate $P_{i+2} \lhd_{JP_i} \text{adv}_i \lhd_{JP_{i+1}} \text{adv}_{i+1}$, we obtain the same automaton, except for transitions (6), which are defined by

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \in O\big) \implies (s,\ell,O_{\text{adv}_i}, S\_step_{P'}(s_{\text{init}},\sigma_i,l_{\sigma_i})) \in \mathcal{T}'\big)$$

Transitions (6) are exactly the join point transitions that are in $advTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_i) \cap advTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_{i+1})$. By precondition, there were no such transitions in $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n})$. Because we require that all the JP_j outputs occur nowhere else, JP_i and JP_{i+1} cannot be contained in a O_{adv_j}, thus no transition of type (6) has been added by the weaving of $\lhd_{JP_n} \text{adv}_n \ldots \lhd_{JP_{i+2}} \text{adv}_{i+2}$.

Thus, we have $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \lhd_{JP_n} \text{adv}_n \ldots \lhd_{JP_{i+2}} \text{adv}_{i+2} \lhd_{JP_{i+1}} \text{adv}_{i+1} \lhd_{JP_i} \text{adv}_i = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \lhd_{JP_n} \text{adv}_n \ldots \lhd_{JP_{i+2}} \text{adv}_{i+2} \lhd_{JP_i} \text{adv}_i \lhd_{JP_{i+1}} \text{adv}_{i+1}$. Weaving $\lhd_{JP_{i-1}} \text{adv}_{i-1} \ldots \lhd_{JP_1} \text{adv}_1$ trivially yields the same result. □