

UNIVERSIDAD DE CHILE

FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SAFE AND PRACTICAL DECOUPLING OF ASPECTS WITH JOIN POINT INTERFACES

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN
COMPUTACIÓN

MILTON GALO PATRICIO INOSTROZA AGUILERA

PROFESOR GUÍA:
ÉRIC TANTER

PROFESOR CO-GUÍA:
ERIC BODDEN

MIEMBROS DE LA COMISIÓN:
JOHAN FABRY
ROMAIN ROBBES
JOCELYN SIMMONDS WAGEMANN

Este trabajo ha sido parcialmente financiado por CONICYT - 2011

SANTIAGO DE CHILE
ENERO 2013



University of Chile
Faculty of Physics and Mathematics
Graduate School

Safe and Practical Decoupling of Aspects with Join Point Interfaces

by

Milton Inostroza

Submitted to the University of Chile in fulfillment
of the thesis requirement to obtain the degree of
M.Sc. in Computer Science

Advisor :
ÉRIC TANTER

Co-Advisor :
ERIC BODDEN

Committee :
JOHAN FABRY
ROMAIN ROBBES
JOCELYN SIMMONDS WAGEMANN

This work is partially funded by CONICYT - 2011

Department of Computer Science - University of Chile
Santiago - Chile
January 2013

Resumen

La Programación Orientada a Aspectos (AOP) es una técnica avanzada para modularizar *crosscutting concerns* tales como análisis de perfiles, seguridad, monitoreo, entre otros. Para esto, AOP introduce una nueva unidad funcional llamada *aspecto*. En los sistemas actuales que implementan AOP, los aspectos contienen en su definición referencias explícitas al código base. Este tipo de referencias son frágiles y no permiten el cumplimiento de propiedades importantes de la ingeniería de software tales como el razonamiento modular y la evolución independiente entre los aspectos y el código base.

En esta tesis, se introduce una nueva abstracción llamada Join Point Interfaces, la cual, soporta razonamiento modular y evolución independiente a través del desacoplamiento de los aspectos del código base, mediante un algoritmo de verificación de tipos modular. Join Point Interfaces soporta que los *join points* sean anunciados tanto de forma implícita, mediante el uso de *pointcuts*, como explícita, mediante el uso de *closure join points*. Además, Join Point Interfaces ofrece una semántica de ejecución de *advice*, semejante a *multi-methods*, que permite ejecución polimorfa dependiendo del tipo de los join points. En este trabajo, se muestra como esta nueva abstracción resuelve diversos problemas observados en trabajos relacionados previos.

Se ha implementado Join Point Interfaces como una extensión del lenguaje de programación orientado a aspectos AspectJ. El diseño general de esta propuesta está validado por estudios realizados en proyectos reales de software. En una primera etapa de evaluación se detectaron ciertas limitaciones las cuales fueron solucionadas mediante la introducción de polimorfismo paramétrico y un mecanismo de cuantificación más permisivo. Como resultado, Join Point Interfaces permiten de una manera segura y práctica el desacoplamiento de los aspectos.

Abstract

Aspect-Oriented Programming (AOP) is an advanced technique for modularizing crosscutting concerns such as profiling, security, monitoring, among others. To do this, AOP introduces a new functional unit called *aspect*. In current aspect-oriented systems, aspects references to the base code. Those references are fragile and give up important software engineering properties such as modular reasoning and independent evolution of aspects and base code.

In this thesis, we introduce a novel abstraction called Join Point Interfaces that, by design, supports modular reasoning and independent evolution by decoupling aspects from base code and by providing a modular type-checking algorithm. Join point interfaces can be used both with implicit announcement through pointcuts, and with explicit announcement by using closure join points. Join point interfaces further offer polymorphic dispatch on join points, with an advice-dispatch semantics akin to multi-methods. In this work, we show how our proposal solves a large number of problems observed in previous, related approaches.

We have implemented join point interfaces as an open-source extension to AspectJ. An initial study on existing aspect-oriented programs supports our initial design in general, but also highlights some limitations which we then address by introducing parametric polymorphism and a more permissive quantification mechanism. As a result, join point interfaces are a safe and practical way of decoupling aspects.

Agradecimientos

Llegué al programa de magíster luego que el comité académico rechazara mi postulación al programa de doctorado. Este rechazo fue una de las mejores cosas que me ha sucedido en la vida; comprendí que no solo bastaba demostrar la pasión que me impulsaba a seguir en la academia sino que también tenía que demostrar mis capacidades con hechos concretos sin importar lo logrado en el pasado. Al terminar este proceso, puedo decir que fue maravilloso todo lo que viví y que ciertamente todo esto cambió mi forma de ver y vivir la vida.

Durante estos tres años conocí y llegué a estimar a muchas personas. A todos ustedes, sepan que los llevaré en mis recuerdos por ayudarme a sobreponerme a la adversidad y por estar junto a mí en los buenos y malos momentos.

A mis profesores guías Éric y Eric, gracias por todo el tiempo invertido en mí. Me siento un privilegiado de haber trabajado con ustedes. Éric, gracias por tratarme como a un *juven colega* mas que como a un estudiante; nunca olvidaré tu sinceridad profesional. Eric, Danke dass du immer bereit warst, mir bei meinen Fragen und Zweifeln weiter zu helfen. Weißt du, ich werde nie vergessen, was du mir bei unserem ersten Treffen in Darmstadt gesagt hast: *das ist kein Problem, das ist Etwas, an dem wir zusammen arbeiten müssen*. Diese Wörter waren sehr ermutigend und entscheidend für diesen langen Prozess.

A mis compañeros Ismael, Teresa, Héctor, Pablo y Manuel. Vivimos juntos jornadas duras y fines de semanas inexistentes; gracias por su compañía. En especial, agradecerte Ismael por tu sinceridad y empuje al final de este camino. Mis especiales agradecimientos a Mario por compartirme tu actitud ganadora y a Daniel por todas aquellas conversaciones acerca de la vida. A mis amigos Hernán, Jorge y Andrés. Hernán, te quiero agradecer por tus palabras de aliento al inicio de este proceso y por recibirme en tu casa. Jorge, nunca olvidaré tus palabras: *si lo puedes hacer sencillo, no lo compliques más*. Andrés, quizás no te imaginas lo que tu ayuda significó para mí; mil gracias.

A mis padres y hermana, gracias por mostrarme el camino y dejarme recorrerlo a mi modo. A Cecilia, mi prima, gracias por darme ese calor de hogar en aquellas noches oscuras y frías de la capital. A Ivania, mi compañera, gracias por tu bondad, generosidad y amor. De no haber sido por tí no hubiera viajado a Alemania. Gracias por darme las fuerzas cuando ya no quedaban; se que has vivido tan intensamente este magíster como yo.

A Sandra y Angélica, las amigas que siempre estuvieron ahí; fueron un soporte emocional muy importante en todo este proceso.

Contents

1	Introduction	1
1.1	Goals	2
1.2	Contribution of the thesis	3
1.3	Organization of the thesis	3
2	Background	5
2.1	Aspect-Oriented Programming	5
2.1.1	AspectJ in a nutshell	6
2.2	Modular Reasoning	8
2.2.1	Aspect-aware interfaces	8
2.2.2	Crosscutting interfaces	8
2.2.3	Open modules	8
2.2.4	Join point types	9
2.2.5	Ptolemy	11
2.2.6	EScala	11
2.3	Type soundness and aspects	12
2.4	Summary	12
3	Join Point Interfaces	13
3.1	A First Tour of Join Point Interfaces	13
3.1.1	Defining Join Point Interfaces	14
3.1.2	Advising JPIs	15
3.1.3	Implicit Announcement with Pointcuts	15
3.1.4	Explicit Announcement with Closure Join Points	16
3.1.5	Join Point Polymorphism	17
3.2	Core Semantics of JPIs	18
3.2.1	Syntax	18
3.2.2	Static Semantics	20
3.2.3	Dynamic Semantics	24
3.3	Summary	25
4	A First Evaluation of JPIs	26
4.1	Benefits of Joint Point Interfaces	26
4.2	Join Point Polymorphism	27
4.2.1	Subtyping Patterns	28
4.2.2	Depth of Subtyping Hierarchies	29

4.2.3	Per-Kind Advice Overriding	30
4.3	Threats to Validity	30
4.4	Perspectives on the Flexibility of JPIs	31
5	A More Flexible Language	33
5.1	Generic Join Point Interfaces	33
5.1.1	Motivation	33
5.1.2	Defining and Using a Generic JPI	35
5.1.3	Static Semantics	36
5.1.4	Evaluation	37
5.2	Controlled Global Quantification	37
5.2.1	Motivation	37
5.2.2	Defining and Using a Global JPI	38
5.2.3	Refining global pointcuts	39
5.2.4	Protecting from Global Quantification	40
5.2.5	Static Semantics	40
5.2.6	Evaluation	40
5.2.7	Discussion	41
5.3	Summary	41
6	Join Point Interfaces in AspectJ	42
6.1	AspectBench Compiler	42
6.1.1	JastAddJ as frontend	44
6.2	Syntax extension	48
6.3	Advice-dispatch Semantics	50
6.4	Invariant Pointcut Designators	51
6.5	Static Overloading	53
6.6	Modular Typechecking	54
6.7	Generic Interfaces	55
6.8	Global pointcuts	56
6.9	Reuse of implementation for closure join points	58
6.10	Summary	58
7	Conclusion	59
7.1	Perspectives on quantification	59
	Bibliography	60

List of Listings

2.1	An existing e-commerce application	7
2.2	Discount aspect in plain AspectJ	7
2.3	Shopping example with join point types (JPTs)	10
3.1	Shopping session with discount aspect	14
3.2	Example from Figure 3.1 with closure join points	16
3.3	Advice overriding	18
5.1	LoD Check aspect (excerpt) in plain AspectJ	34
5.2	LoD Check aspect with (non-generic) JPIs	35
5.3	LoD Check aspect with a generic JPI	36
5.4	Base code advised by the generic JPI version of Check aspect	38
5.5	global pointcut designator	39
6.1	New keywords: jpi and exhibits	48
6.3	New AST nodes: JPITypeDecl , ExhibitBodyDecl , and CJPAdviceDecl	49
6.2	New productions for jpi , exhibits , and cjpadvice declaration.	49
6.4	Inlined exhibits clause which use global	57

List of Figures

1.1	Dependencies in traditional AOP and with join point interfaces.	2
3.1	Inheritance between Join Point Interfaces	17
3.2	Syntactic extension for Join Point Interfaces (AspectJ syntax is shown in gray)	19
4.1	A Potential Hierarchy of Join Point Interfaces in Glassbox.	29
6.1	High-level overview of the components of the abc compiler (adapted from Avgustinov et al. [2005])	43
6.2	JastAdd AST representation	45

List of Tables

5.1	Number of advices defined in each version	37
5.2	Number of exhibits clauses defined in each version	40
7.1	Summary of features and their purpose	60

Chapter 1

Introduction

“Modular reasoning means being able to make decisions about a module while looking only at its implementation, its interface and the interfaces of modules referenced in its implementation or interface. For example, the type-correctness of a method can be judged by looking at its implementation, its signature (i.e. interface), and the types (i.e. interfaces) of any other code called by the method.” [Kiczales and Mezini, 2005]

While Aspect-Oriented Programming (AOP) [Filman et al., 2005] aids in obtaining localized implementations of crosscutting concerns, its impact on modular reasoning is not that positive. Indeed, the emblematic mechanism of AOP is pointcuts and advice, where *pointcuts* are predicates that denote *join points* in the execution of a program where *advices* are executed. With such an implicit-invocation mechanism, it is not usually possible to reason about an aspect or an advised module in isolation. As we show in Figure 1.1a, an aspect contains direct textual references to the base code via its pointcuts—with detrimental effects. These dependencies make programs fragile, and hinder aspect evolution and reuse. Changes in the base code can unwittingly render aspects ineffective or cause spurious advice applications. Conversely, a change in a pointcut definition may cause parts of the base program to be non-intended advice, breaking some implicit assumptions made by base code programmers. The fact that independent evolution is compromised is particularly worrisome considering that programming aspects requires a higher level of expertise, and is hence likely to be done by specialized programmers. Therefore, to be widely adopted, AOP is in great need of mechanisms to support separate development in a well-defined manner.

The above issues have been identified early on [Gudmundson and Kiczales, 2001] and have triggered a rich discussion in the community [Kiczales and Mezini, 2005; Steimann, 2006]. In particular, several proposals have been developed to enhance the potential for modular reasoning by introducing a notion of *interface* between aspects and advised code (e.g. [Gudmundson and Kiczales, 2001; Aldrich, 2005; Sullivan et al., 2010; Steimann et al., 2010]). However, as shown in this work, while those proposals do enhance the situation over traditional AOP, none of these proposals manages to fully support independent evolution through modular type checking, mostly because the interfaces are not expressive enough. This is especially troublesome because the existence of a concrete modular type checker is generally considered the first solid evidence of modular reasoning.

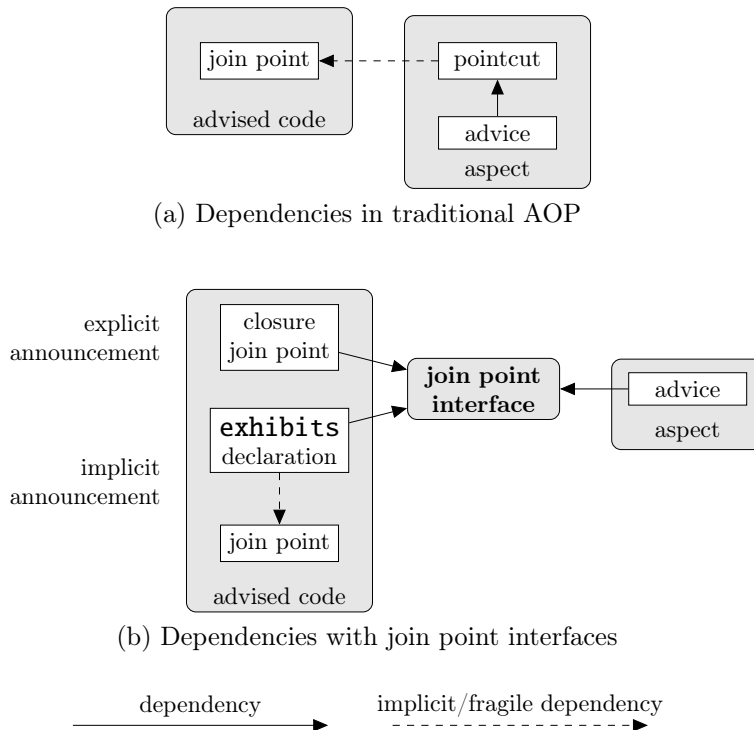


Figure 1.1: Dependencies in traditional AOP and with join point interfaces.

1.1 Goals

In order to address the above issues we set out to define an abstraction layer between aspects and advised code (Figure 1.1b) which allows independent software evolution in a safe and practical way.

The goal of this thesis is to enable fully modular type checking for aspects through *Join point interfaces* (JPIs), which are type-based contracts between aspects and advised code. The specific goals of this work are:

- Design of Join point interfaces: design the syntax and both static and dynamic semantics. The static semantics is intended to provide a type system which supports modular type checking of both aspects and classes. On the other hand, the dynamic semantics specifies the advice-dispatch mechanism and how join points are announced.
- Implementation of Join point interfaces: develop join point interfaces as an extension of the aspect-oriented programming language AspectJ. This extension will be implemented by using AspectBench Compiler [Avgustinov et al., 2005].
- Evaluation of Join point interfaces: migrate real software projects to assess the impact of join point interfaces and its practicability in order to be adopted by the practitioners.

1.2 Contribution of the thesis

The major contribution of this thesis is that by using Join point interfaces, aspects and advised code can be developed and evolve independently in a safe and practical manner. JPIs do allow for strict separate compilation thanks to modular type checking, conversely to previous work [Aldrich, 2005; Sullivan et al., 2010; Steimann et al., 2010]. When programmers in charge of advised code compile their module, they need to include JPI definitions, but no aspect code. Likewise, when aspect experts compile their aspects, they only include the join point interface definitions, but no base code. This is similar to what Java interfaces offer to support independent evolution of object-oriented code. The static semantics of JPIs guarantee that programmers can always safely compose aspects and advised modules, even when they were separately developed and compiled.

Also, this thesis presents the following original contributions:

- the design of join point interfaces, a novel mechanism for decoupling aspect definitions from base code, implemented as an extension of AspectJ;
- a detailed explanation of why join point interfaces are the first abstraction to allow for modular type checking in combination with implicit announcements;
- a novel semantics for advice dispatch in the presence of join point polymorphism, based on multiple dispatch;
- the design and implementation of generic JPIs, to improve on flexibility while retaining type safety;
- the design and implementation of a controlled global pointcut mechanism which can be used to support highly-crosscutting aspects;
- an extensive case study of existing aspect-oriented projects that is used to justify and assess our design decisions.

1.3 Organization of the thesis

This thesis is structured as follows:

Chapter 2 gives basic definitions, introduces the fundamental topics in the field, and reviews the related work.

Chapter 3 shows through an example how programmers can use join point interfaces. It then, shows the syntax and explains our type system and the dynamic semantics in full generality.

Chapter 4 presents the results of the first evaluation of join point interfaces in real software projects. As this evaluation shows, the core version of join point interfaces provides type safety, but lacks flexibility.

Chapter 5 explains how generic JPIs and global pointcuts restore flexibility in our language

design. Also, this chapter presents an evaluation of this new language's features.

Chapter 6 discusses the implementation of join point interfaces as an extension of the AspectBench Compiler for AspectJ.

Chapter 7 distills the most important conclusions of this thesis and the perspectives for future work.

The notion of JPIs was first presented in the New Ideas track of ESEC/FSE [Inostroza et al., 2011]. This work is the result of the development and maturation of that idea; syntax and semantics have evolved, and both implementation and evaluation are completely new, as are the ideas of generic JPIs and global pointcuts. The work of Chapters 2, 3, 4, and 5 was published as a Technical Report TUD-CS-2012-0106, Technical University of Darmstadt [Bodden et al., 2012].

Chapter 2

Background

This chapter introduces most of the concepts used in this thesis: we first present the fundamental concepts of Aspect-Oriented Programming. Then, we illustrate those concepts in practice by means of AspectJ, an aspect-oriented language. We conclude this chapter discussing the most salient and most related proposals in areas such as modular reasoning and type soundness.

2.1 Aspect-Oriented Programming

Separation of concerns [Parnas, 1972] can provide many software engineering benefits like simpler evolution and reduce system complexity. Current programming paradigms such as object-oriented, procedural, and functional are able to decompose a system by using one *dominant* dimension of separation at a time [Tarr et al., 1999]. This decomposition generates that certain concerns are encapsulated in modules, leaving the others, those which cannot be modularized, tangled and scattered in the program. These concerns, such as profiling, security, monitoring, among others, are the so-called *crosscutting concerns*.

Aspect-Oriented Programming (AOP) is an advanced technique for modularizing crosscutting concerns. AOP introduces a new modular unit called aspect, which describes the conditions under which the crosscutting behavior is to be applied. There are different mechanisms to implement aspects in a programming language [Masuhara and Kiczales, 2003]: *Pointcut/Advice*, *Traversal specifications*, *Class composition*, and *Open classes*. The most emblematic mechanism is Pointcut/Advice. In this mechanism, aspects are defined in terms of pointcuts and advices. A pointcut is a predicate which is evaluated over join points. A join point is a well defined point in the program execution (advised code), i.e.: method call, method execution, field accessors, among others. An advice is the crosscutting behavior which is executed when its associated pointcut matches the given join points.

AOP has been successfully applied to many software projects to obtain localized implementations of crosscutting concerns. Khatchadourian et al. [2009] show more than 20 projects which benefited from the use of aspects. In this corpus we can find very different kinds of

applications: figure editors, photo and music manipulators, software design pattern checkers, among others.

Aspect taxonomy An interesting point to notice about these projects is the character of the implemented aspects, which can be categorized in either specific or highly-crosscutting aspects. Specific aspects define pointcuts that quantify over a restricted set of join points. Conversely, highly-crosscutting aspects define pointcuts that quantify over several join points which, commonly, belong to several, if not all, modules in a system. A representative application of the former category is the AJHotDraw [AJHotDraw, 2003] project: the aspect version of JHotDraw which is a Java framework for technical and structured 2D graphics. AJHotDraw uses aspects to implement undo actions for commands such as align and delete. For each command there is an aspect that intercepts the corresponding command execution to support the undo behavior. On the other hand, a representative application of the latter category is the LawOfDemeter project: a small set of aspects checking the compliance to the Law of Demeter programming rules [Lieberherr et al., 1988]. LawOfDemeter uses the **Check** aspect which intercepts each method execution in the whole program to analyze whether the rules are respected or not.

2.1.1 AspectJ in a nutshell

To illustrate the concepts of AOP in practice we use the AspectJ [Laddad, 2003] language. AspectJ, which implements the pointcut/advice mechanism, is a widely used aspect-oriented programming language which is implemented as an extension of the well-known Java programming language. AspectJ extends Java with support for two kinds of crosscutting implementation: *dynamic crosscutting* and *static crosscutting*.

Dynamic crosscutting lets programmers introduce additional behavior in well-known points in the program execution by means of different kinds of advices such as *before*, *around*, *after*, among others. An advice is a method-like mechanism used to declare that certain code should be executed at each of the join points in a pointcut. Both *before* and *after* advice are executed in that order whenever the pointcut associated with them matches a certain join point. An around advice has the special capability of selectively preempting the normal computation at the join point. At the body of an around advice, a call to **proceed** invokes the next most specific around advice, or, if no around advice remains, computes the original join point.

Static crosscutting lets programmers define new operations for existing types. In this thesis work, we only consider the dynamic crosscutting mechanism of AspectJ.

As an example, let us consider an existing e-commerce application (Listing 2.1), in which a customer can check out a product by either buying or renting the product. A business rule states that, on his/her birthday, the customer is given a 5% discount when checking out a product.

In order to implement the discount rule we introduce the **Discount** aspect. As Listing 2.2


```

1 class ShoppingSession {
2     ShoppingCart cart;
3     double totalValue;
4
5     void checkOut(Item item, double price, int amount, Customer cus){
6         cart.add(item, amount); //fill shopping cart
7         cus.charge(price); //charge customer
8         totalValue += price; //increase total value of session
9     }
10 }

```

Listing 2.1: An existing e-commerce application

```

1 aspect Discount {
2     pointcut CheckingOut(double pPrice, Customer pCustomer):
3         execution(* ShoppingSession.checkOut(..))
4         && args(*,pPrice,*,pCustomer);
5
6     void around (double aPrice, Customer aCustomer) :
7         CheckingOut(aPrice, aCustomer) {
8         double factor = aCustomer.hasBirthdayToday() ? 0.95 : 1;
9         proceed(aPrice*factor, aCustomer);
10    }
11 }

```

Listing 2.2: Discount aspect in plain AspectJ

shows, we define a pointcut named **CheckingOut** (Lines 2 - 4), which captures every execution of the **checkOut** method (Listing 2.1, Line 5). This pointcut is defined in terms of two built-in AspectJ pointcuts: **execution** and **args**. The **execution(* ShoppingSession.checkOut(..))** means that we want to pick out every join point that is a method execution defined on **ShoppingSession** whose name is **checkOut** regardless of its return type (the ***** wildcard) and parameters (the **..** wildcard). The **args(*, pPrice, *, pCustomer)** means that we want to pick out every join point that has four arguments and also the second argument must be of type **double** and the fourth argument must be of type **Customer**. To implement the birthday discount rule, we define an advice (Lines 6 - 10) that is associated with the pointcut **checkingOut** (Line 7). As it can be noticed, we use an *around* advice because we need to take full control over the join point computation: the total amount and customer values are modified accordingly to the discount rule and then the **checkOut** method is invoked via the call to **proceed()**.

2.2 Modular Reasoning

There is a very large body of work that is concerned with modularity issues raised by the form of implicit invocation with implicit announcement provided by aspect-oriented programming languages like AspectJ, starting with Gudmundson and Kiczales [Gudmundson and Kiczales, 2001]. In the AOP literature, many proposals have been formulated, some aiming at providing more abstract pointcut languages (e.g. [Gybels and Brichau, 2003]), and others—as we do here—introducing some kind of interface between aspects and advised code. A detailed discussion of all these approaches is outside the scope of this thesis, so we concentrate on the most salient and most related proposals. A recent exhaustive treatment of this body of work and neighbor areas can be found in [Steimann et al., 2010].

2.2.1 Aspect-aware interfaces

In their ICSE 2005 paper, Kiczales and Mezini argue that when facing crosscutting concerns, programmers can regain modular reasoning by using AOP [Kiczales and Mezini, 2005]. Doing so requires an extended notion of interfaces for modules, called aspect-aware interfaces, that can only be determined once the *complete* system configuration is known. While the argument points to the fact that AOP provides a better modularization of crosscutting concerns than non-AOP approaches, it does not do anything to actually enable modular type checking and independent development. Aspect-aware interfaces are the conceptual backbone of current AspectJ compilers and tools, which resort to whole program analysis, and perform checks at weave time.

2.2.2 Crosscutting interfaces

Sullivan et al. [2010] formulated a lightweight approach to alleviate coupling between aspects and advised code. Crosscutting interfaces, XPIs for short, are design rules that aim at establishing a contract between aspects and base code by means of plain AspectJ. With the XPI approach, aspects advising the base code only define advices, not pointcuts. The pointcuts, in turn, are defined in another aspect representing the XPI. Sullivan et al. argue that this additional layer of indirection improves the system evolution because the resulting XPI is a separate entity and hence can be agreed upon as a contract. The authors also show how parts of such a contract can be checked automatically using static crosscutting or contract-checking advices in the XPI aspect itself. However, without a language-enforced mechanism, XPIs cannot provide any strong guarantees on modularity.

2.2.3 Open modules

In the same period, Aldrich formulated the first approach for language-enforced modularity, Open Modules [Aldrich, 2005]. Here, modules are properly encapsulated and protected from

being advised from aspects. A module can then open itself up by exposing certain join points, described through pointcuts that are now part of the module’s interface. The advantage is that aspects now rely on pointcuts for which the advised code is explicitly responsible. Aldrich formally proves that this allows replacing an advised module with a functionally equivalent one (but with a different implementation) without affecting the aspects that depend on it. Ongkingco et al. have implemented a variant of Open Modules for AspectJ [Ongkingco et al., 2006].

2.2.4 Join point types

Join point types (JPTs) [Steimann et al., 2010] are a further (and the most recent) step in the line of Open Modules. Also a language-enforced mechanism, JPTs provide a higher level of abstraction than pointcuts: join point types, which can be organized in a subtype hierarchy, provide a more natural way to deal with complexity, just like interfaces in Java help classify object behaviors. Also, join point types open opportunities for advice overriding.

JPTs were the starting point of our work. The general idea of introducing a typed interface between base and aspects with support for both implicit and explicit announcement is the same, but the realization differs in a number of fundamental ways, which make JPIs both safe (while JPTs are not), and more flexible than JPTs.

First, as an example, consider Listing 2.3, which corresponds to the birthday discount example (Listing 2.2), implemented using JPT. As we can see, join point types are data structures with mutable fields (line 1 - 4). This has a number of problems: most importantly, it makes advice and base code fragile, as both depend on the actual name of a JPT field. On the side of the base code, this dependency is even prohibitively strong: the example in Listing 2.3 only works because the programmer has named the local variables `price` and `c` equal to those names used in the join-point type definition. Our join point interfaces are instead like method signatures, so name dependencies for context information are avoided; matching of arguments happens purely by argument position, just like standard procedural abstraction (see Section 3.1.1).

Second, mutation of JPT fields can yield incorrect or even undefined semantics. In line 15, the code updates the variable `totalValue`. The type checker for JPTs allows those updates also in the case where `totalValue` is a local variable. But this is unsound: any aspect advising `CheckingOut` join points may opt to execute the original join point in another thread, thus causing a data race on the *local* variable `totalValue`, breaking the important guarantee of Java and AspectJ programs that local variables cannot cause data races. Or even worse, the aspect may choose not to `proceed` at all, in which case the value of `totalValue` may be completely undefined. Bodden’s earlier work on closure join points discusses those issues in even further detail [Bodden, 2011].

Third, join point types are unsound because they do not specify return and exception types at join points. This means that both weave time and runtime errors can occur whenever aspects and base code do not coincidentally agree on these types. JPIs make these assumptions explicit and therefore ensure type safety.

```

1 joinpointtype CheckingOut {
2     double price;
3     Customer c;
4 }
5
6 class ShoppingSession exhibits CheckingOut {
7     ShoppingCart cart;
8     double totalValue;
9
10    void checkOut(Item item, double price, int amount, Customer cus) {
11        //assume price==100 && c.hasBirthdayToday()
12        exhibit new CheckingOut(price, c) {
13            cart.add(item, amount); //then here price==95...
14            c.charge(price);
15            totalValue += amount;
16        }; //... and here price==100 again
17    }
18 }
19
20 aspect Discount advises CheckingOut{
21    void around(CheckingOut jp) {
22        double factor = jp.c.hasBirthdayToday() ? 0.95 : 1;
23        jp.price = jp.price * factor;
24        proceed(jp);
25    }
26 }

```

Listing 2.3: Shopping example with join point types (JPTs)

Fourth, JPTs allow for a variant matching semantics in their pointcut expressions. As explained in Section 6.4, this is unsound because although the calling of **proceed** can be performed with “valid” arguments, this potentially can lead to a **ClassCastException** when the base code is executed.

Fifth, JPTs do not handle polymorphism properly. To illustrate this point, let us consider that we introduce two subtypes of **Buying** from our example of Section 3.1: **BuyingBestSeller** and **BuyingEcoFriendly**. With JPTs, whenever a book is bought that is both a bestseller and an eco-friendly print, the same **Discount** advice for **CheckingOut** is executed twice. This implies that any side effects of the advice (e.g. sending a notification email) are duplicated for a single book purchase. The reason is that JPTs do not really support polymorphic join points: instead, in the case above, two separate join point instances are generated, one of each type. Because both instances are subtypes of **CheckingOut**, the advice executes twice. In Chapter 4, Section 4.2.2 we show a case, in our case studies, where the problem of multiple advice execution manifests.

Sixth, Steimann’s proposal is not symmetric in that aspects cannot exhibit join points on their own, to be advised by other aspects. The justification for this choice is to avoid infinite

loops due to aspects advising themselves. As explained by Tanter [2010], this is however not an adequate solution. Avoiding infinite loops requires a dynamic semantic construct, like execution levels. As a matter of fact, infinite loops can happen with JPTs¹. Our proposal of join point interfaces is symmetric: aspects can exhibit join points. Integrating execution levels in AspectJ has already been done [Tanter et al., 2010] and would be helpful for JPIs as well, although this is an orthogonal concern.

Finally, JPTs do not support type variables nor global quantification. Directly motivated by the case study our design includes these two additions making JPIs a much more practical approach to aspect interfaces.

2.2.5 Ptolemy

Ptolemy [Rajan and Leavens, 2008] supports explicit announcement of events (an idea initially proposed by Hoffman and Eugster [Hoffman and Eugster, 2007]). Events are defined with event types. Event types are similar to JPTs in the sense that they are struct-like specifications; they include information about the return type, but not about checked exceptions. Originally, Ptolemy did not support event subtyping. It was recently extended with a form of subtyping, which allows for depth subtyping in event types [Fernando et al., 2012]; an event type that binds a value of type **Point** can be specialized by an event subtype that declared to bind a value of type **ColorPoint**; this is possible only because Ptolemy does not support the call of **proceed** with alternative arguments, otherwise depth subtyping would be unsound, as we will explain in this thesis. Further, because Ptolemy only supports explicit announcement, emitted join points have a single most specific type, simplifying advice dispatch. Ptolemy supports behavioral contracts, called translucent contracts [Bagherzadeh et al., 2011], to specify and verify control effects induced by event handlers. These verification techniques go beyond more lightweight interfaces like Java interfaces and JPIs.

2.2.6 EScala

EScala [Gasiunas et al., 2011] is an approach to modular event-driven programming in Scala, which, like JPTs and JPIs, also combines implicit and explicit events. EScala does not support **around** advice, so event definitions need not declare return types. Also exception types are not provided by design, but this reflects the design philosophy of the Scala language. EScala treats both events and handlers as object members, subject to encapsulation and late binding. Aspects are scoped with respect to event owners rather than event types.

A major contribution of our work is to realize that the above proposals rely on *insufficiently expressive interfaces* to really allow separate development and modular type checking. As mentioned above, the contribution of our work has already been reflected in recent enhancements to the Ptolemy language.

¹Examples available online: <http://pleiad.cl/research/scope/levels/iiia-loops>

2.3 Type soundness and aspects

Soundness issues with the type system of AspectJ were first reported by Wand et al. [2004], although no solution was proposed. Jagadeesan et al. [2006] formulate a sound approach in which an advice type may depend on explicitly-declared type variables. They use the same signature for **proceed** and the corresponding advice. Some flexibility is regained because the type variables from the signatures can be instantiated for each join point. Due to parametricity, it is difficult to express arbitrary replacement advice.

MiniMAO₁ [Clifton and Leavens, 2006] and StrongAspectJ [De Fraine et al., 2008] both consider different signatures for advice and **proceed**, thereby allowing more liberal pointcut/advice bindings while maintaining soundness. StrongAspectJ is more flexible in that it supports signature ranges for pointcuts and type variables for generic advices. As recognized by the authors, however, the more expressive typing constructs of StrongAspectJ result in quite complicated syntax forms.

Aspectual Caml [Masuhara et al., 2005] is an aspect-oriented extension of OCaml. An interesting feature of Aspectual Caml is that it uses type information to influence matching, rather than for reporting type errors. More precisely, they infer the type of pointcuts from the associated advices, and only match join points that are valid according to these inferred types. It is unclear if the inference approach may be ported to an object-oriented language with subtype polymorphism.

All the above approaches are formulated in a traditional pointcut-advice setting, without relying on interfaces to uncouple base and aspect code. Our design of generic JPIs basically follows the proposal of Jagadeesan et al. This choice is pragmatic: while we have clear evidence of the interest of this approach for JPIs in our case study (Section 5.1), we have not faced a single case where the extra flexibility of StrongAspectJ was required. This may change in the future as we experiment with more programs.

2.4 Summary

This chapter has introduced most of the concepts used in this thesis. We have presented the fundamental concepts of Aspect-Oriented Programming. We have shown how AspectJ, an Aspect-Oriented language, implements the Pointcut/Advice mechanism bringing *aspects* to the Java programming language. We finished the chapter presenting a review of the state of the art on modular reasoning and type soundness in presence of aspects.

The next chapter presents Join Point Interfaces (JPIs) and how they can be used. We explain how programmers can use JPIs with either implicit or explicit join point announcements. Also, we show how JPIs properly manage join point polymorphism and how programmers can take advantage of this. Then, we present the core semantics of JPIs, describing the syntax, dynamic and static semantics.

Chapter 3

Join Point Interfaces

In this chapter we describe how Join Point Interfaces (JPIs) work in practice and we present the JPIs core semantics. In the first part of this chapter we show how to use JPIs to decouple the aspect definition from the advised base code through a practical example. Then, we show the different mechanisms supported by JPIs to raise join points. We finish this part describing how our advice-dispatch semantics works properly on polymorphic join points. In the second part of this chapter, we present the syntax and describe the type system and the dynamic semantics of JPIs in full generality.

3.1 A First Tour of Join Point Interfaces

To show how JPIs work in practice, let us reuse the introductory example used in Section 2.1 - Figure 2.2, which we will then improve using join point interfaces. We assume an e-commerce system in which a customer can check out a product by either buying or renting the product. A business rule states that, on his/her birthday, the customer is given a 5% discount when checking out a product. We will be adding further rules later.

Listing 3.1 shows the complete implementation of this example. The around advice in lines 17–21 applies the discount by reducing the item price to 95% of the original price when proceeding on the customer’s birthday. Note how brittle the AspectJ implementation is with respect to changes in the base code. Most changes to the signature of the `checkOut` method, such as renaming the method or modifying its parameter declarations, will cause the `Discount` aspect to lose its effect. The root cause of this problem is that the aspect, through its pointcut definition in lines 13–15, makes explicit references to named entities of the base code—here to the `checkOut` method.

```

1 class ShoppingSession {
2     ShoppingCart cart;
3     double totalValue;
4
5     void checkOut(Item item, double price, int amount, Customer cus){
6         cart.add(item, amount); //fill shopping cart
7         cus.charge(price); //charge customer
8         totalValue += price; //increase total value of session
9     }
10 }
11
12 aspect Discount {
13     pointcut checkingOut(double price, Customer cus):
14         execution(* ShoppingSession.checkOut(..))
15         && args(*, price, *, cus);
16
17     void around(double price, Customer cus):
18         checkingOut(price, cus) {
19         double factor = cus.hasBirthdayToday() ? 0.95 : 1;
20         proceed(price*factor, cus);
21     }
22 }

```

Listing 3.1: Shopping session with discount aspect

3.1.1 Defining Join Point Interfaces

For this example, programmers could use the following JPI definition to successfully decouple the aspect definition from the advised base code:

```

jpi void CheckingOut(double price, Customer cus);

```

In our core language design, join point interfaces are, except for the `jpi` keyword, syntactically equivalent to method signatures. This is for a good reason: methods are designed to be modular units of code that can be type-checked in a modular way, and we wish to inherit this property for join point interfaces. The method signature chosen for a join point interface typically coincides with the signature chosen for the advice that handles the appropriate join point (see Line 17 in Listing 3.1).

It is important to note that join point interface signatures are assumed to be complete with respect to the checked exceptions they define. The above JPI definition declares no exception, which gives both the aspect and the base code the guarantee that the respective other side of the interface cannot throw any checked exceptions at join points of this type. We will present the details of this in Section 3.2. Note that, as we discussed in Section 2.2, our system is the first to correctly handle exceptions in that manner.

3.1.2 Advising JPIs

The following piece of code shows how programmers would advise join points defined by a join point interface:

```
aspect Discount {
    void around CheckingOut(double price, Customer c) {
        double factor = c.hasBirthday()? 0.95 : 1;
        proceed(price*factor, c);
    }
}
```

Crucially, advices in our approach only refer to join point types, not to pointcuts. Concretely, the above advice declaration refers to the **CheckingOut** JPI definition. Because of this, advices are completely decoupled from the code they advise; their dependencies are defined just in terms of a JPI declaration. As shown in the example, similar to AspectJ, an advice can use the formal parameters of a JPI to obtain context information exposed at join points. As we will explain in Section 3.2, we impose strict typing constraints on argument and return types. In plain AspectJ, this is not the case. In particular, an advice can use **Object** in the return-type position as a wildcard that just denotes “any type”. In addition, AspectJ uses unsafe covariant typing through **this**, **target** and **args** pointcuts (more on this in Section 6.4).

3.1.3 Implicit Announcement with Pointcuts

Programmers have two different ways to raise join points declared through join point interfaces, through implicit and through explicit announcement. We first explain implicit announcement, in which join points are raised automatically at certain program events captured by AspectJ pointcuts. In our running example, the class **ShoppingSession** can raise the appropriate **CheckingOut** join points as follows:

```
class ShoppingSession {
    exhibits void CheckingOut(double price, Customer c):
        execution(* checkOut(..))
        && args(*, price, *, c);
    ...
}
```

In this piece of code, the programmer raises join points implicitly through an **exhibits** clause. Programmers will usually use this construct whenever wishing to concisely expose multiple join points from the same class. Within our JPI language, a pointcut attached to an **exhibits** clause only matches join points that originate from a code fragment lexically contained within the declaring class. This is to avoid an overly complex semantics with respect to subclassing, as previously observed by others [Steimann et al., 2010]. As for advice,

```

1 class ShoppingSession {
2     ...
3     void checkOut(final Item item, double price, final int amount,
4         Customer cus) {
5         totalValue = exhibit CheckingOut(double alteredPrice, Customer c)
6             {
7                 cart.add(item, amount);
8                 c.charge(alteredPrice);
9                 return totalAmount + alteredPrice;
10            } (price, cus);
11    }
12 }

```

Listing 3.2: Example from Figure 3.1 with closure join points

our type system checks that the necessary constraints are satisfied such that weave-time and runtime errors are avoided (details in Section 3.2).

3.1.4 Explicit Announcement with Closure Join Points

Pointcuts suffer from the *fragile pointcut problem* [Gybels and Brichau, 2003; Stoerzer and Graf, 2005]: when software evolves, the patterns of a pointcut can accidentally miss join points or match unintended ones, resulting in surprising and unintended behavior. With join point interfaces, even when using implicit announcement, this problem is much less severe because the scope of quantification is restricted: pointcuts can only match join points within the same class, and hence programmers can easily see how a pointcut affects a given class. Nevertheless, we offer programmers a language construct for *explicit announcement* as well, which allows programmers to explicitly mark expressions, statements or sequences to be advised by aspects. Explicitly announced join points are useful whenever pointcuts are either not expressive enough, too awkward, or too concrete to conveniently describe exactly which part of the execution should be advised. For instance, Steimann et al. [2010] recently showed that out of 484 advices in an aspect-oriented version of BerkeleyDB [BerkeleyDB, 2010], 218 applied to some statements in the middle of a method. Single statements may be hard to select without explicit join points, yielding bloated and fragile pointcuts. Sequences of statements need to be extracted into a method so that they can be advised using pointcuts.

We support explicit announcement through a language construct called *closure join points*, a mechanism that allows programmers to explicitly mark any Java expression or sequence of statement as “to be advised”. Closure join points are explicit join points that resemble labeled, instantly called closures, i.e., anonymous inner functions with access to their declaring lexical scope. Closure join points were first proposed independently of JPIs [Bodden, 2011]; this work integrates both approaches.

Listing 3.2 shows the shopping-session example from Listing 3.1 adapted using the the proposed syntax. Instead of using a pointcut in an **exhibits** clause, the programmer exposes

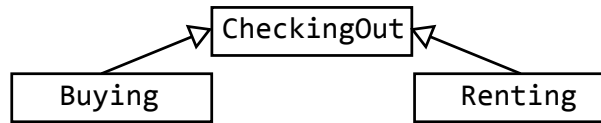


Figure 3.1: Inheritance between Join Point Interfaces

a sequence of statements (lines 5–7) using a closure join point of the **CheckingOut** JPI. Section 3.2 gives the syntax and semantics of closure join points. For a further discussion of our design and implementation of closure join points we refer the interested reader to Bodden [2011]. Finally, note that an advice is oblivious to the fact whether join points are announced implicitly or explicitly; only their types matter.

3.1.5 Join Point Polymorphism

Join points raised explicitly through closure join points can only provide a single join point interface, the one specified in the closure’s header. In the case of implicit invocation, however, JPIs are assigned to join points through pointcuts, and different pointcuts can overlap, i.e., match common join points. As we have seen, a class defines the pointcuts that expose certain join points in its execution, following a given JPI. For instance, in our running example, class **ShoppingSession** defines a pointcut that gives the type **CheckingOut** to all join points that are executions of the **checkOut** method. Because a join point can be matched by several pointcuts, a join point can have multiple types. For instance, an execution of **checkOut** can be seen as a **CheckingOut** join point, and could additionally be seen as a **LoggableEvent** join point (a JPI whose definition is left to the imagination of the reader).

Join point interfaces abstract from join points through types. In the same way that object interfaces in languages like Java support a flexible form of subtype polymorphism, JPIs enable polymorphic join points. A join point can be seen as providing multiple JPIs, and advice dispatch at that join point can take advantage of this polymorphism, increasing the potential for advice reuse. Like interfaces in Java, JPIs support subtyping. Consider two subtypes of **CheckingOut**, **Buying** and **Renting** (Figure 3.1), and the following business rule: the customer gets a 15% discount when *renting* at least five products of the same kind; this promotion is not compatible with the birthday discount.

Listing 3.3 shows an implementation of this additional rule using sub typing on JPIs. First, we declare the JPI **Renting** as extending **CheckingOut**. The semantics of this sub typing relationship implies that any join point of type **Renting** is also a join point of type **CheckingOut**. The **extends** clause defines how arguments are passed to super join points, similar to primary constructors in Scala [Odersky et al., 2008]. In the example, the first and third arguments of **Renting** join points become the first and second argument, respectively when this join point is seen through the **CheckingOut** interface.

This effect can be seen in the aspect **Discount**, which now declares two advices. The first one is the same as in the previous example. In general, it applies to all **CheckingOut** join points, and if this advice was the only advice in the aspect, then it would indeed execute also for join points of type **Renting**. In the example, however, the aspect defines a second

```

1 jpi void CheckingOut(double price, Customer c);
2 jpi void Renting(double price, int amount, Customer c) extends
   CheckingOut(price, c);
3
4 aspect Discount {
5   void around CheckingOut(double price, Customer cus) {
6     /*as before*/
7   }
8
9   void around Renting(double price, int amt, Customer cus) {
10    double factor = (amt > 5) ? 0.85 : 1;
11    proceed(price*factor, amt, cus);
12  }
13 }

```

Listing 3.3: Advice overriding

advice specifically for **Renting**. In this case, an overriding semantics applies: the more specific advice *overrides* the first advice for all join points that are of type **Renting**. As a result, the first advice only executes for join points of other subtypes of **CheckingOut**, i.e., for **CheckingOut** itself, or for **Buying**. Section 3.2.3 describes advice dispatch in full detail.

3.2 Core Semantics of JPIs

We begin this section by describing the syntax of JPIs, designed as an extension to AspectJ (Section 3.2.1). Join point interfaces allow for modular reasoning about aspect-oriented programs by precisely mediating the dependencies between aspects and base code (recall Figure 1.1b). The most fundamental contribution of JPIs therefore lies in the static type system that supports modular checking: we informally describe it in Section 3.2.2. Our proposal of JPIs also innovates over previous work in the way it supports join point polymorphism. The dynamic semantics of JPIs is described in Section 3.2.3.

3.2.1 Syntax

Figure 3.2 presents our syntactic extension to AspectJ to support join point interfaces and closure join points.

Type declarations, which normally include classes, interfaces and aspects, are extended with a new category for JPI declarations. A **jpi** declaration specifies the full signature of a join point interface: the return type at the join points, the name of the join point interface, its arguments, and optionally, the checked exception types that may be thrown. A join point interface declaration can also specify a super interface, using **extends**. In that case, the name of the extended JPI is given, and the arguments of the super interface are bound to

```

TypeDecl ::= ... | JPITypeDecl
JPITypeDecl ::= “jpi” JPISig [JPIExt] “;”
JPISig ::= Type ID “(” [ParamList] “)” [ThrowsList]
JPIExt ::= “extends” “(” [ArgList] “)”
ClassMember ::= ... | ExhibitDecl
AspectMember ::= ... | ExhibitDecl
ExhibitDecl ::= “exhibits” Type Name “(” [ParamList] “)” [ThrowsList] “:” PointcutExpr
AdviceDecl ::= ... | JPIAdviceDecl
JPIAdviceDecl ::= [Modifiers] [Type] AdviceKind ID “(” [ParamList] “)” [ThrowsList] Block
Expr ::= ... | ClosureJoinpoint
StmtExpr ::= ... | ClosureJoinpoint
ClosureJoinpoint ::= “exhibit” ID “(” [ParamList] “)” Block “(” [ArgList] “)” | “exhibit” ID
Block

```

Figure 3.2: Syntactic extension for Join Point Interfaces (AspectJ syntax is shown in gray)

the arguments of the declared JPI.

Classes and aspects can have a new kind of member declaration, for specifying the join point interfaces that are exhibited. An **exhibits** declaration specifies a join point interface signature and the associated pointcut expression that denotes the exhibited join points.

Further, our extension allows advices to be bound to JPIS. Unlike regular advices, instead of directly referring to a pointcut expression, these advices instead refer to a join point interface. The information about return type, argument types and checked exception types that the JPI specifies becomes part of the advice signature.

Closure join points can be used in any place in which an expression can be used. A closure join point comprises the keyword **exhibit**, an identifier (the name of the JPI used to type the exposed join point), and a block, plus optionally a list of formal and actual arguments. If those lists are omitted, this is equivalent to specifying empty lists. The block effectively defines a lambda expression, while the formal parameter list defines its λ -bound variables. Unlike regular lambda expressions, however, a closure join point must be followed by an actual parameter list: the closure is always immediately called; it is *not* a first-class object.

Our language extension is backwards compatible with AspectJ, i.e., any valid AspectJ syntax is still valid in our language. This allows for a gradual migration of AspectJ programs to join point interfaces. It would be simple, though, to include a “strict” mode that instructs the parser to disallow AspectJ’s regular pointcut and advice definitions.

3.2.2 Static Semantics

We next describe the JPI type system, a key contribution of this work. The type system supports modular type checking of both aspects and classes; only knowledge of shared JPI declarations is required to type check either side. This is similar to type checking Java code based on the interfaces it relies on.

We first discuss how JPIs are used to type-check aspects. We then type-check the base code that exhibits certain JPIs. Finally, we discuss type checking JPIs themselves, in particular considering JPI inheritance.

Type checking aspects

An aspect is type checked just like an AspectJ aspect, save for its advices. There are two facets of type checking an advice: checking its signature and checking its body. Type checking the signature of an advice is simple, but requires care. Each advice declares an advised JPI in its signature; the signature must directly correspond to the JPI declaration: both return and argument types must be exactly the same as those of the JPI. The crucial requirement here is that the typing of all types in the signature is *invariant*. To see why this is the case, assume that we allowed for the following advice definition, which declares that it cannot just accept **Customer** objects, but instead, objects of the super type **Person**:

```
void around CheckingOut(float price, Person p) {
    double factor = p.hasBirthdayToday()? 0.95 : 1;
    //p = new Administrator();
    proceed(price*factor, p);
}
```

The contravariant argument definition would not raise any issues with this particular advice definition. The problem is, however, that nothing would prevent the advice from including the commented statement `p = new Administrator()`, assuming that **Administrator** is a subtype of **Person**. If this statement were included, then the advice would invoke the original join point with an **Administrator** object. Assuming that **Administrator** is not a subtype of **Customer**, this code breaks crucial assumptions on the base-code side, usually manifested in the form of a **ClassCastException**. Conversely, if we allowed for covariant arguments then the original join point could pass arguments to the advice that are of a type that the advice is not prepared to handle.

The opposite argument applies to return types. If we allowed for covariant return types, then calls to `proceed` could return objects of a type that the advice is not prepared to handle. If we allowed for contravariant return types, then the advice could return objects of a type that the base code is not prepared to handle.

The essence of the problem observed here is that the advice signature (and our JPI signature) is used to define both the join points intercepted by an advice, and the join point that

an advice can call through **proceed**. Both calls have exactly opposite variance requirements, which means that the only sound typing must be invariant. In Chapter 4, we will see that in practice this can be too rigid; Section 5.1 will then explain how we relax this requirement with parametric polymorphism through generic types, an idea first proposed in the context of the StrongAspectJ language [De Fraine et al., 2008].

Checked exceptions also require some care. The advice must handle or declare to throw all exception types declared by the JPI it advises. We analyze the advice body to determine which types of exceptions the advice does or does not handle. The advice must not declare any additional exception, as this could lead to uncaught checked exceptions on the side of the base code. It can, of course, declare fewer exceptions, or declare additional exceptions (e.g. **EOFException**) provided they are subtypes of expected exceptions (e.g. **IOException**) that are also declared.

The exceptions that are declared in the advice interface are typed invariantly with respect to the JPI. This strict invariance requirement for exceptions is required for the same reason as for return and argument types. Let us denote T_S the checked exception thrown at a join point shadow¹, T_I the exception type declared in the JPI, and T_A the type of checked exception thrown by the advice. If $T_S < T_I$ ², this means that the context of the shadow is not prepared to handle T_A if $T_S < T_A <: T_I$. Conversely, if $T_S > T_I$, the advice is not prepared to handle T_S when invoking **proceed**. Therefore $T_S = T_I$ by necessity.

In our example, the JPI declaration for **CheckingOut** contains no checked exceptions. This imposes the following restrictions and guarantees:

- The base code can rely on the fact that an advice handling **CheckingOut** cannot throw any checked exceptions. The type system ensures that this is indeed the case.
- Likewise, an advice can rely on the fact that a join point of type **CheckingOut** cannot throw checked exceptions when the advice invokes it through a call to **proceed**. The type system forbids programmers from declaring join points that could throw any checked exceptions to be of type **CheckingOut** .

If needed, the programmer can relax these restrictions by declaring checked exceptions in the join point interface:

```
jpi void CheckingOut(float price, Customer cus) throws SQLException;
```

In this example, both the advised join point and the handling advice are allowed to throw **SQLExceptions**, but also must handle (or forward) the exceptions appropriately.

Type checking the advice body is similar to type checking a method body, with the additional constraint of considering calls to **proceed**. As it turns out, a join point interface is identical to a method signature (except for the **extends** clause used for join point subtyping). In fact, a JPI specifies the signature of **proceed** within the advice body, thereby abstracting

¹A join point shadow is the source expression whose evaluation produces a given join point [Masuhara et al., 2003; Hilsdale and Hugunin, 2004].

²We use $<:$ for subtyping, and $<$ for strict (i.e., non-reflexive) subtyping.

away from the specific join points that may be advised. This is a fundamental asset of JPIs, and the key reason why interfaces for AOP ought to be represented as method signatures (including return and exceptions types). JPIs *fully* specify the behavior of advised join points, thereby allowing safe and modular static checking of advice.

Type checking base code

On the other side of the contract is the base code, which can exhibit join points. The base code must also obey the contract specified by join point interfaces. Part of this contract has to be fulfilled by the pointcut associated with the **exhibits** clause used for implicit announcement: the pointcut has to bind all the arguments in the signature, using pointcut designators such as **this**, **target** and **args**. To comply with our invariant semantics, those pointcuts must match invariantly, i.e., a pointcut such as **this(A)** must only match join points with declared type **A**. As we will explain in Chapter 6, this semantics is different from the one of AspectJ.

Because pointcuts do not account for return and exception types, our type system checks these types at each join point shadow matched by the pointcut associated with the **exhibits** declaration. More precisely, the pointcut is matched against all join point shadows in the lexical scope of the declaring class. Whenever the pointcut matches a join point shadow, the type system checks that the return type of this shadow and the JPI coincide. If the shadow has a different return type, our type checker raises an error message stating that the selected join point shadow is incompatible with the JPI in question. Similarly, the type system validates that the declared exceptions of the shadow are the same as those of the JPIs; if they are not, the type checker raises an error. Here again, type compatibility is invariant.

One may wonder why we raise an error message when pointcuts select incompatible join point shadows and not simply restrict the matching process to exclude incompatible shadows instead. However, our chosen design is in line with AspectJ. In the following example, AspectJ raises an error because the **execution** pointcut selects a join point (in line 2), whose return type is incompatible with that of the around advice:

```
1 | void around(): execution(int foo()) { ... }
2 | int foo () { return 0; }
```

Our type checks at the join point shadows are the fundamental contribution of JPIs from the point of view of type checking base code. Most previous approaches, such as [Steimann et al., 2010], are not able to perform these checks modularly, simply because the specification of return and exception types are not part of their interfaces. Those approaches have to defer type checks to weave time, or even worse, to runtime. With JPIs, conformance can be checked statically and modularly, prior to weaving.

Type checking closure join points

Type checking closure join points follows the Java static semantics for methods defined within inner classes. Code within a closure join point has access to its parameters, to fields from the declaring class and to local variables declared as **final**. Access to non-**final** local variables is forbidden because, by exposing a block of code to aspects in the form of a closure join point, aspects may choose to execute the closure in a different thread, or may choose to not execute the closure at all. In this case, write access to local variables may cause data races on those variables or may leave their value undefined if the write is never actually executed [Bodden, 2011].

Other than that, we type check closure join points according to the same rules as implicitly-announced join points. To keep the code as concise as possible, closure join points do not define return and exception types; instead those are inferred from the join point interface declaration they reference. Because of this, those parts of the closure join point's signature are automatically invariant with respect to the referenced join point interface. Arguments are deliberately not inferred. This is because we want to allow users to give arguments within the closure join point their own local name, independent of what was specified in the JPI declaration and independent of the context in which the closure join point is declared. This is in line with method definitions and lexical scoping in Java. The independence from the closure join point's declaring context is important to distinguish the values of such variables that are declared in this context from the potentially altered values that an advice may pass as the closure join point's arguments. (As we explained in Section 2.2, previous approaches did not treat this problem in a semantically sound manner.) For the types of those arguments, we check that the declaration in the closure join point's header obeys an invariant typing semantics. Further, we check that the body of a closure join point only throws checked exceptions of types that the referenced JPI declares, and conversely that the declaring context of the closure join point is prepared to handle (or declares to forward) all checked exceptions of these types.

Type checking JPIs

Primarily to facilitate reuse on the side of the aspect side, join point interfaces support a subtyping relationship. We employ type checking at the level of JPIs to ensure that subtyping relationships do not break type soundness. As we exemplified in Section 3.1, any JPI can declare to extend another JPI. During this process, the sub-JPI can add context parameters; JPIs thus support *breadth subtyping*. On the other hand, all arguments that do coincide have to be of the exact same type as the respective arguments in the JPI super type; JPIs do not support *depth subtyping*. This is due to the same reasons for which we use invariant argument typing in all other situations. For example, the following code would raise a type error, even if **B** were a subtype of **A**:

```
jpi void Base(A a);  
jpi void Sub(B b) extends Base(b);
```

In addition, JPI subtypes must declare the same return type as their super-JPI, and must declare the same exceptions to be thrown.

3.2.3 Dynamic Semantics

The dynamic semantics of JPIs differ slightly from that of a traditional aspect language. Briefly, the traditional model is as follows [Wand et al., 2004]: all aspects (pointcuts and associated advices) are present in a global environment; at each evaluation step, a join point representation is built and passed to all defined aspects; more precisely, the pointcuts of an aspect are given the join point in order to determine if the associated advices should be executed or not.

With JPIs, aspects do not have pointcuts. They advise JPIs, and base code defines the join points that are of these types, either using pointcuts (implicit announcement) or using closure join points (explicit announcement). The global environment contains aspects with their advices. With implicit announcement, conceptually, a join point representation is passed *only* to the pointcuts defined in the current class, at each evaluation step. If a pointcut matches, then the join point is tagged with the corresponding JPIs. Then, the advices that advise one of the tagged JPIs are executed.

In presence of join point polymorphism and inheritance among JPIs, it is interesting to ask which advice is executed. We write A_T to denote an advice of aspect A that advises JPI T ; we write jp^T to denote a join point jp tagged with JPI T . The semantics of advice dispatch closely mimics the semantics of message dispatch in multiple dispatch languages like CLOS [Paepcke, 1993] and MultiJava [Clifton et al., 2000]. Indeed, an aspect with its multiple advices (each declared to advise a specific JPI) can be seen as a generic function with its multiple methods. Once a join point jp is tagged with interfaces T_1, \dots, T_n we select, for each aspect A , all *applicable* advices. An advice A_S is *applicable* to jp^{T_1, \dots, T_n} if there exists an i such that $T_i <: S$. In order to support overriding, among all applicable advices A_{S_1}, \dots, A_{S_k} , we invoke only the *most-specific* ones, defined as the A_{S_j} such that for all i , either $S_j <: S_i$ or $S_i \not<: S_j$.

Aspect-oriented programming, like any publish-subscribe mechanism, inherently supports *multiple reactions* to a single event. This differs from multiple dispatch, which requires exactly one method to execute. The difference manifests in two ways in the semantics. First, if there are no applicable advice, then nothing happens; no advice executes. In contrast, a multiple-dispatch language throws a *message-not-understood* error if no applicable method can be found. Also, message dispatch requires that there is a *unique* most-specific applicable method, otherwise an *ambiguity* error is raised³. In our case, we execute all the most-specific applicable advices, in the precedence order imposed by regular AspectJ when multiple advices of a same aspect match the same join point [AspectJ Team, 2003].

In the above, we have overlooked one specificity of AspectJ and most aspect languages: the fact that advices can be of different *kinds*—before, after, or around. The advice overriding

³In a statically-typed language like MultiJava, both cases can be ruled out by the type system.

scheme we described is kind-specific: an advice may override another advice only if it is of the same kind. (In Chapter 4, we will show cases where this is useful.) For instance, consider an aspect that defines two advices A_{T_1} and A_{T_2} , with $T_2 <: T_1$. If one is a before advice and the other is an after advice, both are executed upon occurrence of a join point jp^{T_2} . Conversely, if both are around advices, only the most specific (A_{T_2}) executes, as explained above.

In practice, we found that advice overriding is not always desirable (see Section 4.2.2). We support the possibility to declare an advice as **final**, meaning it will always execute if applicable, regardless of whether there exists a more specific applicable advice; in such a case, both execute following the standard AspectJ composition rules.

A fundamental asset of the dispatch semantics presented here is that it gives the guarantee that a given advice executes *at most once* for any given join point. This is in stark contrast with the semantics of join point types [Steimann et al., 2010], where the same advice can surprisingly be executed several times for the same join point (as we explained in Section 2.2).

3.3 Summary

In this chapter, we have exposed how Join Point Interfaces (JPIs) can be used in practice. We first describe how programmers can decouple aspects by defining and advising JPIs. Then, we show how the advice-dispatch semantics works over polymorphic join points. Finally, we concluded this chapter giving a detailed description of the syntax, static semantics and dynamic semantics of the language developed so far. The following chapter presents the impact of JPIs in real software projects and also exposes some limitations detected in this first evaluation.

Chapter 4

A First Evaluation of JPIs

We first discuss the benefits of Join point interfaces based on previous studies. Then we report on a case study where we converted several AspectJ applications to use JPIs. This study brings a number of interesting insights related to join point polymorphism. Finally, we discuss the limitations revealed by this study, which motivated the extensions to JPIs that we will describe in the following chapters.

4.1 Benefits of Joint Point Interfaces

Join point interfaces establish a clear contract between base code and aspects, such that separate development can be supported. This is in essence similar to crosscutting programming interfaces (XPIs) [Sullivan et al., 2010] and join point types (JPTs) [Steimann et al., 2010], see Chapter 2. The benefits of XPIs and JPTs on modularity have been empirically demonstrated in previous editions of TOSEM [Sullivan et al., 2010; Steimann et al., 2010].

Recently, Dyer et al. 2012 report on an exploratory study of the design impact of different approaches to aspect interfaces. They consider the evolution of aspect-oriented software using different approaches, namely plain AspectJ, annotation-style, and quantified, typed events [Rajan and Leavens, 2008]. While they do not consider JPIs in their study, their key results support our design of join point interfaces:

- The use of inter-type declarations is prevalent. By integrating JPIs in AspectJ, we inherit this mechanism for free.
- Quantification failure due to the need to advise join points that cannot be denoted using the pointcut language occurred on 5% of advised join point shadows. Explicit announcement addresses these cases nicely.
- The lack of quantification support with quantified typed events was problematic because it required keeping track of design rules that affect all members of a given class (e.g. make all methods **synchronized**). Implicit announcement with **exhibits** clauses supports these cases.

- Almost a fifth of changes to pointcuts were due to the fragile pointcut problem. Any abstraction mechanism for denoting join points is immune to this issue.
- There were several instances where the fact that context information is restricted to join point specific attributes (**this**, **target**, **args**) was problematic, yielding additional complexity. Explicit-announcement approaches make it easy to expose arbitrary context information.¹

To evaluate the importance of a sound treatment of checked exceptions in aspect interfaces, it is instructive to look at the lessons learned by Robillard and Murphy in an effort to design robust programs with exceptions [Robillard and Murphy, 2000]. They report that focusing on specifying and designing the exceptions from the very early stages of development of a system is not enough; exception handling is a global phenomenon that is difficult and costly to fully anticipate in the design phase. Thus, inevitably, the exceptions that can be thrown from modules are bound to evolve over time as development progresses and this global phenomenon is better understood. The support that JPIs provide to report exception conformance mismatch between aspects and advised code in a modular fashion is therefore particularly necessary: as modules change their exception interface, immediate and local feedback is crucial for deciding if these changes must be promoted to the actual contract between aspects and advised code. This avoids errors before system integration time.

4.2 Join Point Polymorphism

To evaluate JPIs in practice, we have converted two existing AspectJ applications from the corpus of [Khatchadourian et al., 2009]: AJHotDraw and LawOfDemeter. These rewritten projects are available online at our project’s web page². Then, to assess our semantics for join point polymorphism, we have closely inspected a set of interesting subjects from this corpus: AJHotDraw, an aspect-oriented version of JHotDraw, a drawing application; Glassbox, a profiler for Java applications; SpaceWar, a space war game that uses aspects to extend the game in various respects; and LawOfDemeter, a small set of aspects that check for the compliance with the Law of Demeter programming rules [Lieberherr et al., 1988].

The first three projects were selected because of their comparatively large size and number of aspects. LawOfDemeter is a rather small project that showcases an interesting use of pointcuts, as discussed further below.

We inspected the programs using both the AspectJ Development Tools [AJDT, 2012] and AspectMaps [Fabry et al., 2011]. These tools allowed us to easily identify which advices advise which join point shadows. In particular, we focused on the shadows that are advised by more than one advice, as this hints at potential for subtyping. We also systematically investigated all pointcut expressions used in these projects and looked for potential type hierarchies. Our investigation revealed several interesting example hierarchies and clearly supports the usefulness of our semantics of join point subtyping. We now discuss a few rep-

¹Context-aware aspects [Tanter et al., 2006] are a general mechanism that supports arbitrary context information with implicit announcement; a similar flexibility is found in AspectScript [Toledo et al., 2010].

²<http://www.bodden.de/jpi/>

representative examples.

4.2.1 Subtyping Patterns

We identify two patterns that programmers use to “emulate” subtyping with pointcuts.

LawOfDemeter contains the following pointcuts:

```
pointcut MethodCallSite():
    scope()
    && call(* *(..));

pointcut MethodCall(Object thiz, Object target):
    MethodCallSite()
    && this(thiz)
    && target(target);

pointcut SelfCall(Object thiz, Object target):
    MethodCall(thiz, target)
    && if(thiz == target);
```

These pointcuts form an instance of a pattern that we call *subtype by restriction*. **MethodCall** restricts the join points exposed by **MethodCallSite** to instance methods, through additional **this** and **target** pointcuts. **SelfCall** restricts this set further by identifying self calls using an additional **if** pointcut. A programmer could model this join point type hierarchy with JPIs as follows:

```
jpi Object MethodCallSite();

jpi Object MethodCall(Object thiz, Object target) extends
    MethodCallSite();

jpi Object SelfCall (Object thiz, Object target) extends
    MethodCall(thiz, target);
```

The example shows that it is useful to allow subtypes to expose more arguments than their super types (a.k.a. breadth subtyping): **MethodCall** exposes **this** and **target**, while **MethodCallSite** exposes nothing at all.

SpaceWars includes various instances of the *subtype by restriction* pattern, but also features instances of the dual pattern, *super type by union*. In the example that follows, the pointcut **unRegister**, is defined which matches a subset of the join points matched by **syncPoint** because **syncPoints** includes additional join points by disjunction (set union). Here also, the subtype induced by **unRegister** exposes an additional argument.

```

pointcut syncPoint():
    call(void Registry.register(..))
    || call(void Registry.unregister(..))
    || call(SpaceObject[] Registry.getObjects(..))
    || call(Ship[] Registry.getShips(..));

pointcut unRegister(Registry registry):
    target(registry)
    &&
    (call(void register(..)) || call(void unregister(..)));

```

4.2.2 Depth of Subtyping Hierarchies

Glassbox proved to be a very interesting case study in that it provides over 80 aspects, and more than 200 pointcut definitions, with potential for non-trivial join point type hierarchies. Here, we only show one of the most interesting examples in Figure 4.1. The figure shows a hierarchy formed by 11 pointcuts within the aspect **ResponseTracer**. We have added **Stats** as a root type that the aspect does not contain explicitly, but which could be introduced to abstract the common parts of the **StartStats** and **EndStats** pointcuts.

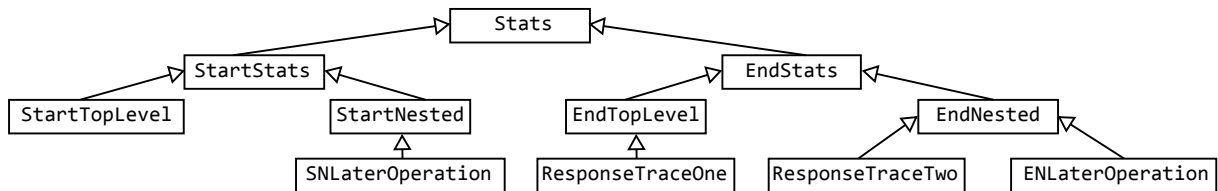


Figure 4.1: A Potential Hierarchy of Join Point Interfaces in Glassbox.

Advice Overriding

Glassbox showcases the interest of being able to declare some advice as **final** to avoid overriding.

An aspect in charge of system initialization advises the execution of **TestCase** object constructors. Some test classes implement the **InitializedTestCase** interface (**ITC** for short), and some implement the **ExplicitlyInitializedTestCase** interface (**EITC** for short); some classes implement both (these interfaces are added via inter-type declarations). More precisely, five classes implement only **ITC**, five classes **EITC** only, and one class implements both. The aspect defines four before advices on these constructors, discriminating between different categories using pointcuts. These pointcuts correspond to four join point types $T_1 \dots T_4$, where T_1 is a super type of the three others. Due to overriding, the advice for T_1 never executes since it is always overridden. The solution would then be to refactor the program to move the advice for T_1 in a separate aspect. A simpler solution is to declare the advice for T_1 as **final**, which guarantees that it will not be overridden and therefore always execute.

4.2.3 Per-Kind Advice Overriding

AJHotDraw contains the following definitions:

```
pointcut commandExecuteCheck(AbstractCommand acommand) :
    this(acommand)
    && execution(void AbstractCommand+.execute()) ..
    && !within(*..JavaDrawApp.*);

before(AbstractCommand acommand):
    commandExecuteCheck(acommand) {...}

pointcut commandExecuteNotify(AbstractCommand acommand) :
    commandExecuteCheck(acommand)
    && !within(org.jhotdraw.util.UndoCommand) ..
    && !within(org.jhotdraw.contrib.zoom.ZoomCommand);

after(AbstractCommand acommand):
    commandExecuteNotify(acommand) {...}
```

This is another instance of the *subtype by restriction* pattern, with **commandExecuteNotify** refining the **pointcut commandExecuteCheck**. This example validates our semantics to consider advice kinds separately when resolving advice overriding (Section 3.2.3). In the example, the first **pointcut** is advised with a **before** advice, while the second is advised by an **after** advice. Assume now that we had abstracted from those **pointcuts** using JPIs as in the following code:

```
before CheckingView(AbstractCommand acommand){..}
after NotifyingView(AbstractCommand acommand){..}
```

In this example, **NotifyingView** is a subtype of **CheckingView**. If we did *not* separate advices by advice kind when determining advice overriding then only the **NotifyingView** would execute at a **NotifyingView** join point, leading to an altered semantics compared to the original AspectJ program. Conversely, because we *do* separate advices by kind, when encountering a **NotifyingView** join point, the **CheckingView** advice is executed before the join point and the **NotifyingView** afterward.

4.3 Threats to Validity

As it was described in the previous sections, part of our methodology to assess the impact of JPIs in the development of real-world software projects consists of porting two concrete projects from the corpus of Khatchadourian [Khatchadourian et al., 2009]: AJHotDraw and LawOfDemeter. We have identified two main assumptions that could pose a threat to the validity of our results.

The first assumption is that the selected projects actually represent the two opposites in the spectrum of how crosscutting aspects can be in a project: AJHotDraw implements only specific aspects, while LawOfDemeter implements only highly-crosscutting aspects. As a consequence of this assumption, the first threat to the validity of our methodology is that it could be argued that the number of migrated projects is too low, and that is necessary to migrate a major number of projects to assess the real impact of JPIs in aspect-oriented software projects.

Regarding this issue, we argue that since we have shown how to use JPIs in the two extremes of aspect categorization, a combination of both approaches can be applied to projects with aspects that are neither too specific nor too highly-crosscutting. Moreover, it is important to observe that the study of these projects leads us to a more flexible implementation of JPIs, as we explain in the following section. This increases our confidence on the assumption regarding the selected projects. In a worst case scenario, a case study involving several projects should involve a methodology similar to the one already used in this work.

The second assumption is about the validation of the migrated versions against the original ones. Our criteria to decide whether a project was successfully migrated to its JPI version was only to verify that the compilation process finishes without reporting any compilation errors. As an additional verification, we perform manual checks to compare the basic functionality of the projects. In these checks, the original version and the JPI version of each project seemed to have equivalent behavior. We did the validation this way because unfortunately the selected projects did not provide, at that time, a test suite.

The threat associated with this assumption is that the behavioral equivalence of the JPI version of the projects cannot be asserted with strong confidence. However, to mitigate this situation, we relied on the help of the AspectJ Development Tools [AJDT, 2012] and AspectMaps [Fabry et al., 2011] in the migration process. These tools helped us identify the classes where the **exhibits** clauses should be defined in order to exhibit exactly the same join points of the original version. In case of any misbehavior detected on the JPI versions, it would be necessary to perform this process again.

4.4 Perspectives on the Flexibility of JPIs

The typing rules for JPIs currently enforce invariance on both return and exception types of join point interfaces (Section 3.2.2). This is the simplest way to ensure soundness, but can be too rigid in practice. If a pointcut matches a large number of shadows, invariance forces us to define multiple JPIs (and advices) for the different types of shadows. In an application like AJHotDraw, in which most pointcuts are very specific anyway, this rigidity is less problematic than in applications that rely on wide-matching pointcuts, such as LawOfDemeter. In AJHotDraw, we found that only 3 advices were concerned, and as a result the total number of advices in this project increased from 49 to 77, almost twice as much. In LawOfDemeter the number of advices was increased more than 10 times, going from 6 to 68! This observation strongly motivates the need for introducing more flexibility in the type system. In Section 5.1, we introduce *generic* interfaces to address this issue.

On a related front, aspects that use wide-matching pointcuts, like in LawOfDemeter, or dynamic analysis aspects like data race detectors and profilers, suffer from the class-local quantification approach followed by JPIs and others [Steimann et al., 2010]. These aspects require modifying many (if not all) classes to add the corresponding **exhibits** clauses. Since aspects are also extremely helpful in these application scenarios, it seems necessary to make some pragmatic decision with respect to the scope of quantification. We address this issue in Section 5.2 with a novel mechanism for controlled global quantification.

Chapter 5

A More Flexible Language

In this chapter, we present two new language features which allow us to overcome the limitations detected in the first evaluation of JPIs in real software projects. The first part of this chapter presents *Generic join point interfaces* which ease the implementation of aspects which have pointcuts that match a large number of shadows that typically have unrelated types. Generic interfaces aim to avoid code duplication in the aspect side. The second part of this chapter presents *Controlled global quantification* which is a mechanism to ease the implementation of highly-crosscutting aspects; it prevents the modifying of several modules when aspect programmers implement them.

5.1 Generic Join Point Interfaces

As revealed by the case study, JPIs easily cause code duplication in aspect definitions when programmers implement highly-crosscutting aspects, such as monitoring, dynamic analysis, access control, exception handling, etc. The problem is that highly-crosscutting aspects have pointcuts that match a large number of shadows that typically have unrelated types. AspectJ deals with this issue by abandoning soundness. JPIs retain soundness, but at a cost: in the language design presented so far, programmers have to define different JPIs and advices for all the different shadow types. To mitigate this problem, we introduce *generic JPIs*. A generic JPI is a JPI that includes type parameters in its signature. Type parameters allow for the sound use of a polymorphic advice. We first dive into a concrete example from the case study, then show how a generic JPI solves the problem. We then briefly discuss the semantics and implementation of generic JPIs, before reporting more precisely on their impact on our case study.

5.1.1 Motivation

To illustrate the situation, let us consider the **Check** aspect defined in the plain AspectJ version of LawOfDemeter (LoD) project. This aspect checks that an object only sends messages

```

1 aspect Check {
2     private IdentityHashMap objectViolations = new IdentityHashMap();
3
4     public pointcut scope() :
5         !within(lawOfDemeter..*)
6         && !cflow(withincode(* lawOfDemeter..*(..)));
7
8     pointcut methodCalls(Object thiz, Object target) :
9         scope()
10        && call(* *(..))
11        && this(thiz)
12        && target(target);
13
14    after(Object thiz, Object target): methodCalls(thiz, target) {
15        if (!ignoredTargets.containsKey(target) &&
16            !Pertarget.aspectOf(thiz).contains(target)) {
17            objectViolations.put(thisJoinPointStaticPart,
18                thisJoinPointStaticPart
19            );
20        }
21    }
22 }

```

Listing 5.1: LoD Check aspect (excerpt) in plain AspectJ

to a certain set of closely related objects according to the Law of Demeter [Lieberherr et al., 1988]. Listing 5.1 shows an advice that registers the LoD violations within an aspect called **Check**. The important thing to note in this aspect definition is the extreme quantification used (`call(* *(..))`), which is typical of dynamic analysis aspects.

To migrate the **Check** aspect to use JPIs, conversely to what we suggested in Section 4.2.1, we *cannot* use a single JPI like the following:

```

1 |jpi Object MethodCall(Object thiz, Object target);

```

The reason is that, in order to ensure type soundness, such a JPI could only match join points where the actual types at the shadows are **Object**. As explained earlier, covariant matching of shadows is unsound. This means that we need to define one JPI per possible combination of types for this, target and the return type! In the LoD project, just handling the **Check** aspect for the examples included in the project required us to write 21 JPIs:

```

1 |jpi void    MethodCall_1(TemporalQueue thiz, TemporalQueue target);
2 |jpi int     MethodCall_2(TemporalQueue thiz, TQ_N target);
3 |jpi PQ_Node MethodCall_3(TemporalQueue thiz, TemporalQueue target);
4 |//etc.

```

```

1 aspect Check {
2     void registerViolation(Object this, Object target,
3         JoinPoint.StaticPart jp) {
4         if (!ignoredTargets.containsKey(target) &&
5             !Pertarget.aspectOf(this).contains(target)) {
6             objectViolations.put(jp, jp);
7         }
8     }
9
10    // one advice per type combination...
11    after MethodCall_1(TemporalQueue this, TemporalQueue target) {
12        this.registerViolation(this, target, thisJoinPointStaticPart);
13    }
14
15    after MethodCall_2(TemporalQueue this, TQ_N target) {
16        this.registerViolation(this, target, thisJoinPointStaticPart);
17    }
18
19    after MethodCall_3(TemporalQueue this, TemporalQueue target) {
20        this.registerViolation(this, target, thisJoinPointStaticPart);
21    }
22
23    // etc.
24 }

```

Listing 5.2: LoD Check aspect with (non-generic) JPIs

In addition to this tedious and fragile list of JPI definitions, the **Check** aspect itself has to contain one advice for each such JPI. As Listing 5.2 shows, all advice bodies are exact copies (note that for **after** advice, the return type of the JPI is simply ignored). While this approach is “correct” in that all expected join points are matched, and type soundness is preserved, it is highly cumbersome for programmers and, most importantly, it does not scale. As soon as one wants to apply the LoD aspects to other projects, more JPIs and advices have to be defined.

5.1.2 Defining and Using a Generic JPI

We therefore extended our syntax and type system to allow programmers to express a whole range of possible type combination through a single *generic* JPI:

```

1 |<R, A, B> jpi R MethodCall(A this, B target);

```

This generic interface abstracts away the specific types involved. Now the programmer can define a generic version of the **Check** aspect using a polymorphic advice (Listing 5.3). Note that the specific type variables used in the JPI and the corresponding advice do not need to match; they are independent abstract parameters.

```

1 aspect Check {
2     // ... registerViolation ...
3     <T, U> after MethodCall(T this, U target){
4         this.registerViolation(this, target, thisJoinPointStaticPart);
5     }
6 }

```

Listing 5.3: LoD Check aspect with a generic JPI

Our support for generics includes type *bounds* as in Java. For instance, the following JPI characterizes join points where an argument is of a subtype of **Number**:

```

1 <T extends Number> jpi void JP(T arg);

```

In such a case, an **exhibits** clause on the base code side must also be generic, and it must use the same type bounds (if present). For instance:

```

1 class A {
2     <N extends Number> exhibits void JP(N n) :
3         call(void *(..)) && args(n);
4     // ...
5 }

```

Note that this is much more flexible than the same definition using **Number** directly as the type of **n**. The invariant matching semantics means that only methods where the argument is specifically of type **Number** will match. Using the type parameter **N** makes it possible to match all join points uniformly where the argument is a subtype of **Number**.

5.1.3 Static Semantics

Generic JPIs are a simple extension of JPIs with basic support for bounded polymorphism. We only support upper bounds for type variables (declared with **extends**), since they are enough to handle the needs of polymorphism that we have observed in practice through the case study. We leave to future work the exploration of the other features found in object-oriented languages with parametric polymorphism (lower bounds, wild-cards, type constraints as in Scala, etc.).

The semantics of type checking in presence of parametric polymorphism is exactly the same as that of Java with generics, more precisely as defined in Featherweight Generic Java [Igarashi et al., 2001].

A generic JPI is like a generic method signature. A generic advice is type-checked just like a generic method, where **proceed** is given the type of the JPI (modulo renaming of

type variables). When checking base code, a type at a shadow (join point representation) matches a type variable in the JPI if it is a subtype of the upper bound of that variable. As explained above, this is where flexibility is gained, allowing generic JPIs to be practical in the face of wide crosscutting. Note that soundness is not compromised thanks to parametricity [Reynolds, 1983]: only the bound of the type variable can be directly relied upon.

5.1.4 Evaluation

	AspectJ	Non-generic JPI	Generic JPI
LoD	6	68	6
AJHotDraw	49	77	49

Table 5.1: Number of advices defined in each version

To assess the impact of generic JPI, we re-implemented the AJHotDraw-JPI and LoD-JPI projects. As Table 5.1 shows, generic JPIs completely eliminate the problem of repeated advice declarations in both projects. While the number of advice declarations in the (non-generic) JPI version was considerably increased (more than 10x for LoD), the version using generic JPIs presents the exact same number of advices. This clearly validates the practical positive impact of extending JPIs to support parametric polymorphism.

5.2 Controlled Global Quantification

As revealed by the case study from Chapter 4, if JPIs are used in combination with highly-crosscutting aspects, developers may be forced to modify several (if not all) existing classes to introduce **exhibit** clauses. The problem is that such aspects use wide-matching pointcuts to capture several join points within different classes. To overcome this problem, we introduce *global pointcuts*, which can be attached directly to JPI definitions. Aspect programmers can use global pointcuts for *controlled global quantification*. The quantification is global, as it is evaluated over all classes in the system. At the same time the quantification is controlled because classes and aspects have the option to restrict or widen quantifications within themselves, or to seal themselves off against any kind of global quantification. We first revisit the motivating example of Section 5.1, then show how a global pointcut solves the problem. We then discuss the semantics and the implementation of global pointcuts. Finally, we report the impact of global pointcuts on our case study.

5.2.1 Motivation

Let us consider the situation in which a programmer wishes to apply the generic JPI version of the **Check** aspect (Section 5.1, Listing 5.3) to the following two arbitrarily chosen classes **TemporalQueue** and **Repository**. First, the programmer would introduce the **exhibits**

clauses, as shown in Listing 5.4. We note that this approach is fragile and does not scale: programmers must modify every single class to introduce the **exhibits** clauses. Moreover, this redundancy is unnecessary, as in most cases all classes will bind the same JPI to the same pointcut expression. In the migration of LoD to its generic JPI version, this problem forced us to modify 21 of 23 classes to include the corresponding **exhibits** clauses.

```

1 import jpis.*;
2
3 public class TemporalQueue extends PriorityQueue {
4     <R, T, I> exhibits R MethodCall(T this, I target) :
5         call(R *(..))
6         && This(this)
7         && Target(target)
8         && !within(lawOfDemeter..*)
9         && !cflow(withincode(* lawOfDemeter..*(..)));
10    ...
11 }
12
13 public class Repository extends Entity {
14     <R, T, I> exhibits R MethodCall(T this, I target) :
15         call(R *(..))
16         && This(this)
17         && Target(target)
18         && !within(lawOfDemeter..*)
19         && !cflow(withincode(* lawOfDemeter..*(..)));
20    ...
21 }

```

Listing 5.4: Base code advised by the generic JPI version of Check aspect

5.2.2 Defining and Using a Global JPI

We therefore extended our syntax and type system to allow programmers to introduce globally defined templates for **exhibits** clauses, defined in the form of a *global pointcut*, attached to a JPI:

```

<T, U, V> jpi T MethodCall(U this, V target) :
    call(T *(..))
    && This(this)
    && Target(target)
    && !within(lawOfDemeter..*)
    && !cflow(withincode(* lawOfDemeter..*(..)));

```

Those templates affect all existing classes. Global pointcuts use an inlining semantics: by default, i.e., if a class (or aspect) states nothing about the JPI **MethodCall**, then the class

(or aspect) will automatically inherit an appropriate **exhibits** clause based on the global pointcut definition. Such implicitly introduced **exhibits** clauses are then treated exactly as if programmers had introduced them by hand. As we will next show, though, those implicitly defined **exhibits** clauses are just templates that every class (or aspect) can choose to refine.

5.2.3 Refining global pointcuts

The JPI **MethodCall** exposes every single method invocation to the **Check** aspect, which in turn detects whether such calls violate the Law of Demeter or not. The Law of Demeter states that classes should only access the public interface of other classes. In particular, they should not invoke methods on objects referenced through another class' public fields. But now consider a case where we use the class **Output** to print some messages to the standard output stream.

```
1 class Output {
2     public static void print(String message){
3         System.out.println("Message: " +message);
4     }
5
6     public static void print(float value){
7         System.out.println("Message: " +value);
8     }
9     ...
10 }
```

As can be seen, every call to **System.out.println** causes the **Check** aspect to register a violation of the law. But such violations are not of interest to us. A naïve attempt to address this issue is to modify the global pointcut in the JPI, but this is fragile.

Instead, we propose a refinement mechanism that allows classes and aspects to control the way in which they are advised through global pointcuts: Whenever a class (or aspect) *does* define an **exhibits** clause for a JPI that also comprises a global pointcut, this **exhibits** clause *overrides* the one that would normally be implicitly inherited. At this point, the programmer can use a special primitive pointcut **global** to refer to the global pointcut, if needed.

```
1 class Output {
2     <T, U, V> exhibits T MethodCall(U this, V target):
3         global(this, target)
4         && !call(T java..*.*(..));
5     ...
6 }
```

Listing 5.5: **global** pointcut designator

As an example, Listing 5.5 shows how the class `Output` can refine the global pointcut to restrict matching within itself.

5.2.4 Protecting from Global Quantification

When refining a global pointcut with an `exhibits` clause, programmers may provide no pointcut at all. In that case, the class or aspect is *sealed* against the global pointcut of the respective JPI. This approach to controlled global quantification is useful in that it allows programmers to refine pointcuts for JPIs that they are aware of. But in a code-evolution scenario, programmers may wish to protect classes and aspects from being advised altogether, even as new global pointcuts are introduced. To this end, we introduce a new modifier that allows programmers to mark those classes and aspects in the form of `sealed class Output {...}`. Sealing a class or aspect completely disables the effects of global pointcuts within this scope. Note, however, that even a sealed class can still opt to expose join points selectively by using `exhibits` clauses or closure join points.

5.2.5 Static Semantics

Global pointcuts are a simple extension of JPIs, also in terms of their semantics. Our type checks enforce that the `global` pointcut designator is only used in `exhibits` clauses that bind to a JPI declaring a global pointcut. The signature of the `global` pointcut is defined through the signature of the JPI that defines this pointcut. Our type-checking rules for classes, aspects and their exhibit clauses remain unchanged.

5.2.6 Evaluation

	Generic JPI version	Global JPI version
LoD	130	0
AJHotDraw	46	46

Table 5.2: Number of exhibits clauses defined in each version

To assess the impact of global pointcuts, we re-implemented the generic-JPI versions of both AJHotDraw and LoD with global pointcuts. As Table 5.2 shows, global pointcuts significantly decrease the amount of scattered `exhibits` clauses in projects such as LoD, which implement highly-crosscutting aspects. While the number of `exhibits` clauses in the (generic) JPI version of LoD was 130, the version using global JPIs does not have any `exhibits` clauses left. However, for applications like AJHotDraw, which include highly specific pointcuts, global quantification cannot help, as different classes must expose different join point shadows.

5.2.7 Discussion

There is a clear tension between global quantification and class-level **exhibits** clauses. On the one hand, **exhibits** declarations at the class level demand more code annotations but have the positive effect of allowing for truly local reasoning at the class level. On the other hand, global pointcuts allow programmers to eliminate many code annotations at the cost of losing the ability to reason about advising locally.

We believe—and our case study shows—that none of the two mechanisms is ideal for every aspect; highly generic aspects do benefit from global quantification. Many of them, like LoD, or more generally dynamic analyses, cause no harm, because they are only observing the base code execution. For other aspects that are meant to directly affect the base execution, global quantification may be the wrong choice. Our design reflects our intent to be pragmatic and support both families of aspects. With our approach, programmers can make the choice of which mechanism to use, in a sound and well-typed setting.

5.3 Summary

In this chapter, we have presented generic interfaces and controlled global quantification. Generic interfaces introduce type variables which allow the sound use of polymorphic advice. On the other hand, controlled global quantification is a mechanism which supports two kinds of quantification: global and controlled. The quantification is global because it is evaluated over all classes and aspects in the system. At the same time the quantification is controlled because classes and aspects have the option to restrict or widen the quantification within themselves.

In the next chapter, we provide a description of how each feature of our language design was implemented. We implement JPIs as an extension of the AspectBench compiler for the aspect-oriented language AspectJ.

Chapter 6

Join Point Interfaces in AspectJ

With this thesis we provide a full implementation of join point interfaces as an extension to the AspectBench Compiler (abc) [Avgustinov et al., 2005]. We begin this chapter by presenting an introduction to abc, its architecture and how each of its components: frontend, matcher, weaver and backend works. Then, we present a detailed explanation of how each feature of JPIs was implemented. Since this implementation is maintained within abc’s own code base¹, we point out only the most relevant parts of our compiler extension.

6.1 AspectBench Compiler

The AspectBench Compiler (abc) is an extensible compiler for AspectJ. abc is a framework for implementing AspectJ extensions, which truly allows easy experimentation with new language features. This is the main reason for using abc to develop an AspectJ extension which fully implements the ideas proposed in this thesis work.

The four main components that abc implements are: frontend, matcher, weaver, and backend. The frontend, implemented by using JastAdd [Ekman and Hedin, 2007], is in charge of both lexer and parser processes, and, also performs semantics checks such as type-checking, name analysis, among others. The matcher component tests every pointcut definition, that belong to an advice declaration on all join points located in the base code. The weaver component introduces in each join point a call to those advice definitions where the associated pointcut definition matches. The backend component, which is partly implemented by using Soot [Raja Vallée-Rai and Co, 1999], is in charge of performing certain code optimizations and finally generates the Java bytecode.

Figure 6.1 shows, in a high-level overview, how the above described components interact with each other. A further detailed explanation can be found in [Avgustinov et al., 2005]. As shown in this diagram, the frontend and the backend have been split into two decoupled components.

¹abc can be downloaded at: <http://aspectbench.org/>

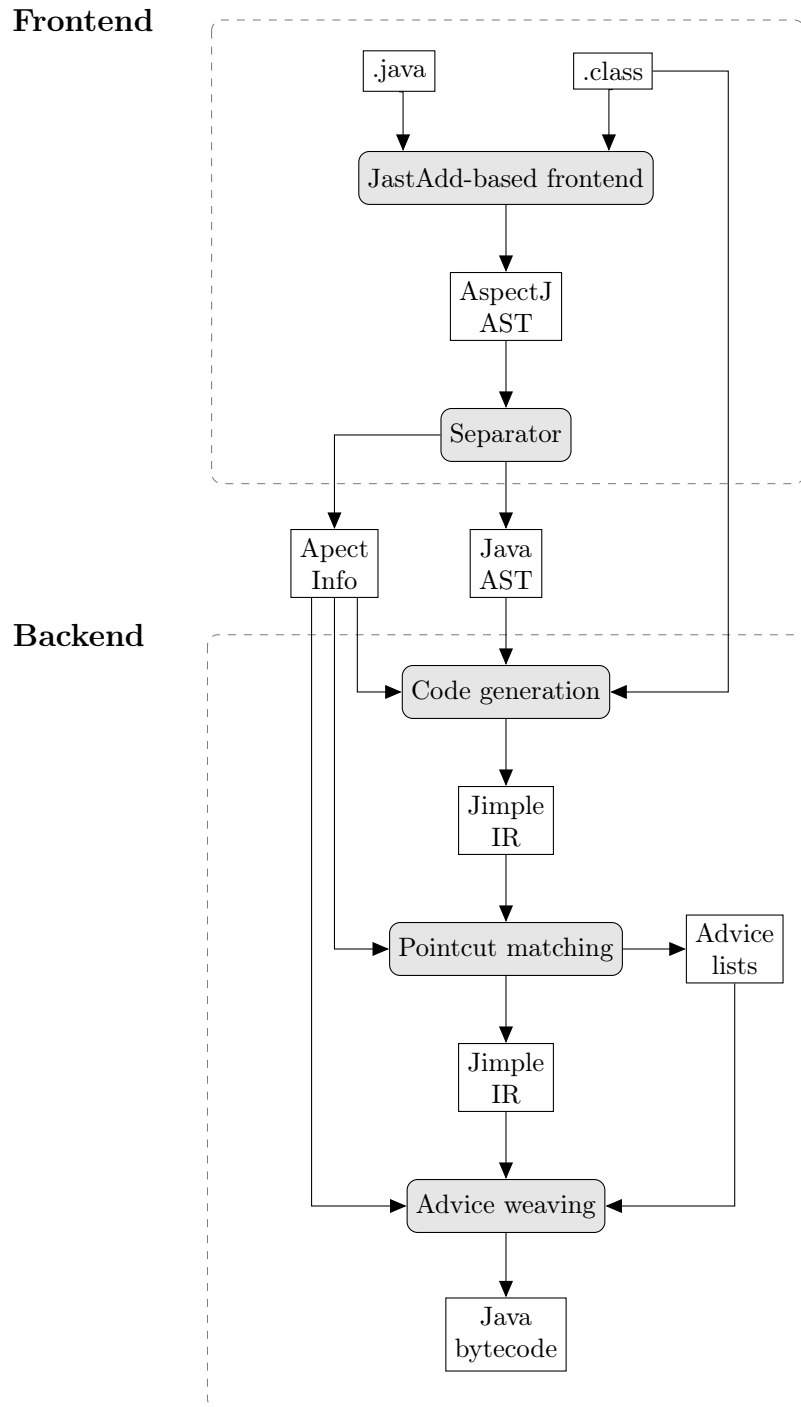


Figure 6.1: High-level overview of the components of the abc compiler (adapted from Av-
gustinov et al. [2005])

The frontend has been split into two sub-components *JastAdd-based frontend* and *Separator*. First, the JastAdd-based frontend takes as input both .class and .java files and is where the AspectJ semantic checks are performed in addition to the ones required by pure Java. A big difference between a Java compiler and an AspectJ compiler is that some semantic checks have to be deferred until after the weaving has occurred. Some of these checks are related to exception checking and return type-checking (in the case of around advice) which must be delayed until the backend takes place. Second, the Separator receives the AspectJ AST and splits it into a standard Java AST and an *AspectInfo* structure. The AspectInfo structure records the aspect-specific information such as advice definitions, pointcut expressions, among others.

As can be seen in Figure 6.1 the Java AST and AspectInfo structure are the bridge that communicates the frontend with the backend components. This means that the backend is completely decoupled from the JastAdd AST.

The backend has been split into three sub-components *code generation*, *pointcut matching*, and *advice weaving*. First, the code generation takes both the Java AST and AspectInfo structure translating them into an internal representation (IR). The IR used is the corresponding Jimple representation [Vallee-Rai and Hendren, 1998]. Jimple is an IR of a Java program designed as an alternative to the stack-based Java bytecode. Second, the pointcut matching determines the static location where each pointcut matches. This information is stored in the *advice lists* structure. Finally, the advice weaving takes as input this advice list and also the Jimple IR. This process weaves the advices that appear in the advice list at the corresponding static location. The output of this process is a woven Java bytecode.

Since our extension mainly performs extensions in the frontend component, in the next section we explain in more details how JastAddJ works. Conversely, since we do not perform any bytecode optimization, a detailed explanation of how Soot works is left out of this work.

6.1.1 JastAddJ as frontend

JastAddJ is an extensible compiler for Java, implemented using JastAdd. JastAddJ is used in the abc compiler to perform tasks such as parsing, name analysis, type analysis, compile-time error checking, among others.

In traditional compilers important data structures like symbol tables, flow graphs, among others, are typically separate from the AST. In JastAdd, these data structures are instead embedded in the AST, using attributes, resulting in an object-oriented model of the program. The resulting AST is represented using Java classes (Figure 6.2b) and the defined attributes form a method API to those classes. The value of an attribute is defined by a directed equation $attr = e(b_1, \dots, b_n)$, where the left-hand side is the attribute and the right-hand side is an expression e over zero or more attributes b_k in the AST. JastAdd supports different kinds of attributes such as *synthesized*, *inherited*, *reference*, *parameterized*, *collection*, and *circular*.

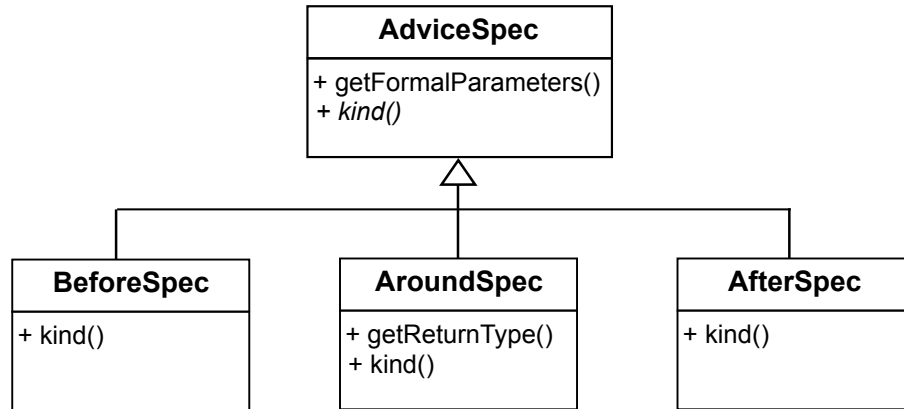
In the implementation of JPIs, only synthesized and inherited attributes were required.

```

1 AdviceDecl ::= AdviceSpec PointcutExpr Block;
2 abstract AdviceSpec ::= ParamaterDeclaration*;
3 BeforeSpec : AdviceSpec;
4 AfterSpec : AdviceSpec;
5 AroundSpec : AdviceSpec ::= ReturnType : Access
6 Proceed ::= Arguments : Expr*

```

(a) AST node for advice declarations in AspectJ (simplified version)



(b) AST classes

Figure 6.2: JastAdd AST representation

A synthesized attribute allows to transfer information up in the AST, whereas an inherited attribute allows to transport information down in the AST. To illustrate how these attributes work in practice, consider the Figure 6.2a that describes the **AdviceDecl** AST node. An **AdviceDecl** AST node is composed of the following AST nodes: an advice specification (**AdviceSpec**), which is related to the kind of advice; a pointcut expression (**PointcutExpr**); and the body of the advice (**Block**). As can be seen, the **AdviceSpec** is an abstract AST node, thus one of its children must be used (line 3 - 5) instead. The **Proceed** AST node, defined in line 6, is composed of zero or more expressions (**Expr***). This makes it possible to call **proceed** with arguments that are either expressions or values. With that said, consider the two following tasks:

1. Define a method called **name()** that returns the name of an **AdviceDecl** AST node. Since programmers cannot give a specific name to an advice declaration, the name is given by its kind—after, before or around—concatenated with a # symbol plus a random alphanumeric string, i.e.: `'after#m4f4ld4'`.
2. Define a method called **getSignature()** that returns the signature of the **Proceed** AST node. The **proceed** signature is given by the return type and the formal parameters of the advice declaration. For an advice declaration with **int** as return type and **Customer** as formal parameter, the **proceed** signature is `int proceed(Customer)`.

The first step to solving the first problem is to define, in the **AdviceDecl** AST node, a method called **name()**. From this method, we can access all the AST nodes defined within **AdviceDeclaration**. We begin the implementation by defining the following method tem-

plate.

```
1 public String name() {
2     ...this.getAdviceSpec()...//retrieves the AdviceSpec AST node
3     ...this.getPointcutExpr()...
4     ...this.getBlock()...
5 }
```

As described earlier, to calculate the **AdviceDecl** name, we need the current “kind advice” of the **AdviceSpec**. As can be seen in the method template, we have access to the advice specification, but not to its children; this means that we need to propagate the information from the child up to the parent. To this end, a synthesized attribute must be defined in the **AdviceSpec** AST node; its children are in charge of providing a concrete implementation of this attribute. The following code shows how **BeforeAdviceSpec**, **AroundAdviceSpec**, and **AfterAdviceSpec** implements this attribute accordingly.

```
1 syn String AdviceSpec.kind();
2 eq BeforeSpec.kind() = "before";
3 eq AroundSpec.kind() = "around";
4 eq AfterSpec.kind() = "after";
```

The synthesized attribute **kind()** is defined, by using **syn**, in line 1. The concrete implementations are defined, by using **eq**, in line 2 - 4. Having this implemented, we can provide a concrete implementation of the **name()** method.

```
1 public String name() {
2     String kind = this.getAdviceSpec().kind();
3     String randomString = Utils.getRandomString();
4     return kind + '#' + randomString;
5 }
```

Notice that through the call of **this.getAdviceSpec()**, we can access the concrete advice specification defined in the advice declaration. The implementation of **Utils.getRandomString()** is left to the imagination of the reader.

To solve the second problem, we first need to define the method **getSignature()** in the **Proceed** AST node. The method template is as follows:

```
1 public String name() {
2     ...this.getArguments()...//returns a list of Expr AST nodes
3 }
```

As can be seen, there is no direct access to the advice declaration, which is the AST node that contains the information to determine the signature of proceed. In this case, we need

to propagate the information down to the AST. We can do that because **Proceed** can be defined as part of the **Block** AST node that belongs to the **AdviceDecl**. To this end, an inherited attribute must be defined in **Proceed**. Also, an implementation of this attribute must be defined in some parent AST node of **Proceed**; the **AdviceDecl** AST node in this case. The following piece of code shows how this can be implemented.

```

1 | inh String Proceed.getReturnType();
2 | inh List<FormalParameter> Proceed.getFormalParameters();
3 |
4 | eq AdviceDecl.getBlock().getReturnType() =
   |   getAdviceSpec().aroundReturnType();
5 | eq AdviceDecl.getBlock().getFormalParameters() =
   |   getAdviceSpec().aroundFormalParameters();

```

The synthesized attributes are defined by using **inh** in line 1-2. The equations (line 4-5), which provide a concrete implementation of these attributes, are defined in the **AdviceDecl** AST node. As can be seen, an equation can access the context of the AST node in which this is defined. For instance, consider the equation, defined in line 4, that retrieves the return type of proceed. This first obtains the advice specification through the call to **getAdviceSpec()** method, and then gets the return type by calling the **aroundReturnType()** method. Having these attributes implemented, we can provide a concrete implementation of the **getSignature()** method.

```

1 | public String name() {
2 |   String returnType = getReturnType();
3 |   List<FormalParameter> parameters = getFormalParameters();
4 |   String params = "";
5 |   for (FormalParameter parameter : parameters) {
6 |     //join the parameters and store into params
7 |   }
8 |   return returnType + " proceed" + "(" + params + ")";
9 | }

```

JastAdd encapsulates the behavior added to the AST by means of aspects. JastAdd aspects can be categorized in imperative or declarative. Imperative aspects are those where programmers add ordinary fields and methods to the AST classes. On the other hand, declarative aspects are those where programmers add attributes, equations, and other operations over the AST classes. JastAdd aspects support inter-type declarations: a declaration that appears in an aspect file, but that actually belongs to an AST class. The JastAdd system weaves the intertype declarations defined in the aspect into the appropriate AST classes. The kinds of intertype declarations that can occur in an aspect include ordinary Java declarations like methods and fields, and attribute grammar declarations like attributes, equations, among others. The aspect syntax is similar to AspectJ. But in contrast to AspectJ, the aspects are not real language constructs. The JastAdd system simply reads the aspect files and inserts the intertype declarations into the appropriate AST classes.

```

public void initLexerKeywords(AbcLexer lexer) {
    lexer.addGlobalKeyword(
        "jpi", // "exhibits"
        new LexerAction_c(
            new Integer(Terminals.JPI), // Terminals.EXHIBITS
            new Integer(lexer.pointcut_state())
        )
    );
}

```

Listing 6.1: New keywords: `jpi` and `exhibits`.

6.2 Syntax extension

The first step of our implementation is to extend the syntax of AspectJ in order to support three new constructs: join point interfaces, exhibits clauses, and advices referring to join point interfaces definitions. To this end, we extend the lexer, parser and the abstract grammar. A join point interface is represented by the `jpi` keyword and it can appear everywhere a type declaration like aspects or classes could. The exhibits clause is represented by the `exhibits` keyword and it can appear in the body declaration of either aspects or classes. Advice declarations can be defined as in normal AspectJ, but now they can refer either to a join point interface definition or to a pointcut expression.

Extending the lexer Implementing the JPIs extension requires adding two new keywords: `jpi` and `exhibits`. As shown in Listing 6.1 - Line 2, we use the `addGlobalKeyword` to add these keywords. Since adding both `jpi` and `exhibits` is quite similar, the addition of `exhibits` is commented out. By using the `addGlobalKeyword` we mean that our keywords will be added to all lexer states: `JAVA`, `ASPECTJ`, `POINTCUT`, and `POINTCUTIFEXPR`. The abc lexer needs to be stateful because it must recognize different tokens in different contexts. This is because the lexical analysis of AspectJ needs to be aware of the fact that different languages are present: Java code, aspect definitions, and pointcut definitions. As an example of that, notice that in Line 6, we change the lexer status to the `POINTCUT` state. This happens after both `jpi` and `exhibits` are consumed by the lexer, and the reason to change to that specific state is because both `exhibits` and `jpi`, potentially, could have attached a pointcut expression.

Extending the parser Once we have introduced the new keywords, we need to extend the parser to add the new productions to the grammar. We begin by extending the already existing grammar rules (Listing 6.2). First, we extend `type_declaration` with a new production called `jpi_declaration` (Line 1) which allows a `jpi` definition to appear anywhere a normal type declaration like classes or aspects could. As can be noted, it is possible to assign identifiers –names after colon– to the matching symbols in order to perform certain operations within Java. Then, we extend `class_member_declaration` with the `exhibit_declaration` production (Line 3) which allows the `exhibits` clause to be defined as a normal method defi-

```

JPITypeDecl : TypeDecl ::=
    ReturnType:Access Parameters:ParameterDeclaration* Exception:Access*;
    SuperTypeName:Access SuperArgumentName:Access*;

ExhibitBodyDecl : BodyDecl ::=
    ReturnType:Access JPIName:Access
    Parameter:ParameterDeclaration* Pointcut:PointcutExpr;

CJPAdviceDecl : AdviceDecl;

CJPPointcutExpr: PointcutExpr;

CJPBeforeSpec : BeforeSpec ::= JPIName:Access;
/*the rest of the advice specifications are omitted from this listing*/

```

Listing 6.3: New AST nodes: `JPITypeDecl`, `ExhibitBodyDecl`, and `CJPAdviceDecl`

tion in either classes or aspects. Finally, we extend the `aspect_body_declaration` with the `cjp_advice_declaration` production (Line 5) which allows advices referring to a `jpi` definition to be considered part of the advice definitions. This means that `cjp_advice_declaration` defines the new syntax representation of before, around, after, after throwing, and after returning advices. The `cjp_advice_declaration` is an existing production defined in the implementation of Closure Join Points [Bodden, 2011].

```

TypeDecl type_declaration = jpi_declaration.d {:return d; :};

BodyDecl class_member_declaration = exhibit_declaration.d {: return d; :};

BodyDecl aspect_body_declaration = cjp_advice_declaration.d {: return d; :};

```

Listing 6.2: New productions for `jpi`, `exhibits`, and `cjpadvice` declaration.

Adding new AST nodes The last step to complete the extension is to add new elements in the abstract grammar. As Listing 6.3 shows, we represent a join point interface through a new AST node called `JPITypeDecl`. An exhibits clause is represented by the new AST node called `ExhibitBodyDecl`. An advice that refers to a join point interface is represented by an `CJPAdviceDecl` AST node.

As it can be seen, the `JPITypeDecl` defines new additional attributes such as `ReturnType`, `Parameters`, among others. Also, this new AST node inherits from the abc’s abstract `TypeDecl` node. In this case, AST nodes referring to JPIs through nodes of the type `TypeAccess`, automatically inherit all the functionality needed to correctly take into account `import` statements and the full system layout when determining the type declaration that a JPI reference refers to: the frontend simply needs to invoke the method `decls()` on the `TypeAccess`, and `JastAdd` automatically computes the matching declarations.

It is important to note that nodes which represent the advice declaration, `CJPAdviceDecl` and `CJPPointcutExpr`, only inherit from the original AspectJ AST node without defining any

new attributes. We will then add new behavior to these AST nodes in order to implement the advice-dispatch semantics. These kinds of AST nodes are introduced to maintain the original syntax and semantics of AspectJ unchanged. We also introduce (Line 13) a child AST node for each different advice specification which contains an attribute, `JPIName`, specifying the access to a referenced JPI definition.

6.3 Advice-dispatch Semantics

One of the most interesting aspects of our implementation is how we assure the correct dispatch semantics for advices referring to JPIs. Remember that our advices, syntactically, do not contain references to pointcut definitions. Instead, each one contains a reference to a given JPI definition which can be associated with one or more `exhibits` clauses. From those exhibits clauses the corresponding pointcuts are extracted and associated with the corresponding advice definition. In presence of polymorphic join points, this turns into a more intricate situation: We must ensure that the most specific advice of a certain JPI hierarchy gets executed. Let a be the advice to compute the pointcut for, as the set of other advices in the same aspect and es , the set of all `exhibits` clauses in the program. Then we compute the pointcut for a as follows:

$$\begin{aligned}
 pc(a, as, es) &= pc^+(a.jpi, es) \wedge \neg pc^-(a, as, es) \\
 pc^+(jpi, es) &= \bigvee_{e \in es, e.jpi <: jpi} e.pc \\
 pc^-(a, as, es) &= \bigvee_{a' \in as, a' \sqsubset a} pc^+(a'.jpi, es)
 \end{aligned}$$

The equation² for pc^+ implements polymorphism: if a refers to $a.jpi$ then a will match not only on join points for $a.jpi$ itself, but also for all subtypes. The equation for pc^- implements advice overriding within the same aspect: if an advice a' has the same kind as a but refers to a more specific JPI type, then a' overrides a , which means that a will not execute for the join points of this JPI. For advice that has been declared `final`, pc^- is simply skipped, so as to avoid overriding.

As shown in Section 6.2, we represent an advice referring to a JPI definition by means of a `CJPAdviceDecl` AST node. The parser creates this AST node containing a special pointcut expression which is represented by the `CJPPointcutExpr` AST node. In order to return the synthesized pointcut, we need to redefine the `pointcut` attribute defined in the `CJPPointcutExpr` AST node to implement the equation described above. This attribute is invoked by the Separator component (see Section 6.1) to store the pointcut expression in the `AspectInfo` structure.

² \sqsubset denotes kind-specific subtyping for advices: $a' \sqsubset a$ means that a' and a are of the same kind and $a'.jpi < a.jpi$.

To illustrate how we calculate the synthesized pointcut expression corresponding to each `CJPAdviceDecl` AST node, suppose that we have a `Discount` aspect defining two *around* advices, one referring to the `Buying` JPI definition and another referring to the `BuyingBestSeller` JPI definition. Suppose also that the following sub-typing relation states for these two JPI definitions: `BuyingBestSeller <: Buying`. The synthesized pointcut expression for the advice referring to the `Buying` JPI definition is calculated as follows: First, we calculate the value of pc^+ by collecting the pointcut expressions attached to the `exhibits` clauses referring to `Buying` or a subtype of it. In this case, we collect the pointcuts attached to `exhibits` clauses referring to either `Buying` or `BuyingBestSeller`, denoted by pc_{Buying} and $pc_{BestSeller}$ correspondingly. Then, to calculate the value of pc^- , we collect the subtypes of the given JPI which have defined an advice declaration in the current aspect. Considering these subtypes we collect the corresponding pointcut expressions. In this case, we collect the pointcut expressions attached to `exhibits` clauses referring to `BuyingBestSeller`. Finally, the synthesized pointcut expression is $(pc_{Buying} \vee pc_{BestSeller}) \wedge \neg pc_{BestSeller}$

To implement the above equation, our implementation has to overcome a few technical obstacles. Both advices and `exhibits` can opt for renaming certain parameter names of its bound JPI. This can lead to problems once we calculate a synthesized pointcut associated with an advice declaration: A pointcut could be defined binding different variable names. We undo this renaming by inlining those pointcuts with the corresponding parameter names defined in the advice declaration. Further, the pointcut $pc^-(a, as, es)$ is used under negation. This raises an issue with argument-binding pointcuts, like `this(a)`, because they cannot be negated: if a pointcut does not match, there is no value that `a` could be bound to. Fortunately, abc supports a way to close such pointcuts so that the variables do not appear free any longer. This is done by rewriting a pointcut such as `this(a)` to `($\lambda a.$ this(a))`; such a pointcut can be negated, and if `a` is of type `A`, the negation is equivalent to `!this(A)`, which yields the semantics we need.

6.4 Invariant Pointcut Designators

As noted in Section 3.2, we must ensure that join point interfaces are invariantly typed. As it turns out, in AspectJ, it is not so straightforward to ensure invariant typing for arguments. The problem is that the standard pointcuts `this`, `target` and `args` come with a variant semantics. In the following AspectJ example, the pointcut matches the call to `foo`, although the declared type of `foo`'s argument is `Integer`, not `Number`:

```
aspect A{
    public static void main(String[] args){ foo(new Integer(2)); }

    public static void foo(Integer a){}

    void around(Number n): call(void *(..)) && args(n) {
        proceed(new Float(3)); // will raise a ClassCastException
    }
}
```

As this example shows, the variant matching semantics of AspectJ is problematic: while the advice is well-typed in AspectJ, the call to `proceed` will cause a `ClassCastException`, as it calls `foo` with a `Float` argument.

One way to address this problem is to re-define the matching semantics of the `this`, `target` and `args` pointcuts such that they match a join point only if the declared type at the join point is exactly the same as the declared type used within the pointcut. This design, however, would break backward-compatibility with AspectJ. Since our overall language design can be integrated as a fully backward-compatible extension, we opted for another design, such that existing AspectJ applications can be easily migrated to our language.

As explained in Section 6.1, the matching process takes place in the backend. For that reason, we need to implement the following strategy to introduce the invariant matching semantics. First, we create the AST nodes which represent the new pointcuts designators. Then, we store in the `AspectInfo` structure a representation of our new pointcut designators. Finally, we need to implement the invariant semantics in these new `AspectInfo` representation objects. With this, we can communicate the new semantics from the frontend to the matcher component at the backend.

At the frontend, we introduce three new pointcut designators `This`, `Target`, and `Args` which are represented by the new AST nodes `ThisInvPointcutExpr`, `TargetInvPointcutExpr`, and `ArgsInvPointcutExpr`. These new nodes inherit respectively from the original AspectJ's AST nodes `ThisPointcutExpr`, `TargetPointcutExpr`, and `ArgsPointcutExpr`. Since the matcher component takes the information about pointcuts from the `AspectInfo` structure, we need to create a proper `AspectInfo` representation for our new AST nodes. To this end, we override the `pointcut` method in order to return the corresponding `AspectInfo` representation: `ThisVarInv`, `ThisTypeInv`, `TargetVarInv`, `TargetTypeInv`, and `ArgsInv`. As an example, suppose that we have the following pointcut definition, `pointcut methodCall(A a): This(a)`. The parser will create the `TargetInvPointcutExpr(a)` AST node, which will return the `ThisVarInv(a)` `AspectInfo` representation when the Separator component invokes the `pointcut` method over this AST node.

At the backend, we need to implement the invariant matching semantics. To implement this, we override the `matchesAt` method used by the matcher component to test if the current pointcut matches at the given join point representation. Concretely, we check that the type defined in the pointcut definition is the same as the join point representation. If this is the case, the `matchesAt` method returns `true`. As a consequence, the weaver introduces the corresponding advice definition at that join point location.

As a result of our implementation, the pointcut `Args(n)` would not match in the example above, since `n` has type `Number`, but the argument value at the join point shadow has declared type `Integer`. Due to the introduction of these novel pointcuts, the existing semantics of `this`, `target` and `args` remains unchanged. When programmers use one of those pointcuts within an `exhibits` clause, our compiler issues a warning, notifying the programmer that `This`, `Target` or `Args` should be used instead, as otherwise type soundness is not guaranteed.

6.5 Static Overloading

We introduce static overloading in JPIs to allow programmers to define JPIs which are inherently related. This means that programmers can define JPIs with the same name, but they can differ in the formal parameter list. For instance, programmers can write the following definitions:

```
jpi void CheckingOut(float price, Customer cus) throws SQLException;

jpi void CheckingOut(float price, int amt, Customer cus) throws
    SQLException;
```

Similar to overloaded methods in Java, each JPI definition is associated with a different advice declaration that implements the particular behavior. Further, JPIs static overloading is completely resolved at compile time.

At the implementation level, we need to modify the type-checker and the mechanism used to perform the type lookup in order to support properly overloaded JPI definitions. To illustrate this, consider that we extend the above program definition with the following:

```
aspect Discount {
    void around CheckingOut(float price, int amt, Customer cus) throws
        SQLException {
        /*omitted code*/
    }
}
```

To type-check this around advice definition, we need to obtain the JPI type declaration associated with the type access name *CheckingOut*. To do that, we call the `decls` method which performs a type lookup related with the given type access name. In this particular case, the `decls` method will return a list containing two elements: `CheckingOut` defined in Line 1 and `CheckingOut` defined in Line 2. This is problematic because the type-checker has no way to know which JPI type declaration is correct.

The type lookup algorithm obtains the corresponding type declaration associated with a type access by considering only the name of the type which is being accessed, in this case *CheckingOut*. In the case of overloaded JPI definitions, we need to provide additional information to the type lookup process. The additional required information are the types associated with each argument defined in the formal parameter list of the JPI type access, in this case `float`, `int`, and `Customer`. In order to keep the current type lookup mechanism unchanged, we decide to introduce a new process in charge of the JPI type lookup which takes as input the name and the type arguments of the current JPI type access.

The JPI type lookups resemble the original type lookup mechanism. This means that the steps followed to find a type declaration are completely the same. First, the frontend

invokes the `decls` method belonging to the `TypeAccess` AST node to check whether there is a type associated with certain type access name or not. In the case of a JPI type access, we introduce a new AST node called `JPITypeAccess` which overloads the `decls` method adding one additional parameter representing the types of the JPI parameters. Then, the `decls` method delegates the search responsibility by calling the `lookupType` method which checks if the type is defined in the current `CompilationUnit`. To perform the JPI type lookup, we introduce a new attribute called `lookupJPIType` in the `CompilationUnit` AST node. If the type is not found in the current `CompilationUnit`, the lookup process is performed in the whole program definition. For this, the method `lookupType` belonging to the `Program` AST node is called. Here again, we introduce a new attribute called `lookupJPIType` in the `Program` AST node in order to perform the JPI type lookup.

6.6 Modular Typechecking

Another interesting aspect of our implementation is how we perform modular type-checking of both aspect and base code. As explained in Section 3.2.2, JPIs are crucial in this process because the type-checking relies only on the type information defined in these interfaces. With this, we ensure that base code is type-checked without considering any type information from the aspect. The situation remains the same for type-checking the aspect.

On the aspect side, we type-check aspects like in normal AspectJ, save for its advice definitions. First, we type-check the advice signature against the referenced JPI definition. We enforce that the advice signature must directly resemble the JPI definition. Then, we type-check the advice body definition. The advice body definition is type-checked as a normal method body, but considering the special case of calling `proceed`. Since a JPI definition represents the signature of the `proceed` within the advice body, we use the referenced JPI to type-check the calls to `proceed`. At this point, we ensure that `proceed` is invoked with the proper argument values like a normal method invocation. An interesting point here is exception handling. All exceptions thrown by `proceed` must be handled properly by the advice definition either by declaring to throw or by handling the exceptions via a try/catch block. As it turns out, in plain AspectJ, such verifications do not exist. Consider the following example in plain AspectJ:

```
public static void checkingOut() throws Exception {
    throw new Exception();
}

void around(): call(* checkingOut(..) ) {
    proceed();
    //throw new Exception();
}
```

The above program is accepted by the static type system. Notice if the commented Line 7 were included, the type-checker would complain because the advice signature says

nothing about throwing such an exception. In our implementation, we have equipped the `Proceed` AST node with three new attributes to perform such verifications. The first attribute `exceptionCollection` collects the exceptions that `proceed` could raise. These exceptions are those defined in the referenced JPI definition. The second attribute `reachedException` checks if a try/catch block handles certain exceptions thrown by `proceed`. The last attribute `exceptionHandling` checks that the advice signature declares to throw the exceptions thrown by `proceed` and that are not handled by a try/catch block.

On the base code side, we need to ensure that join points, emitted via `exhibits` clauses, obey the contract imposed by its JPI definition. The type-checking is split in two. First, the pointcut associated with an `exhibits` clause has to bind all the arguments in the signature, using the pointcut designators `this`, `target`, and `args`. Then, both return and exception types have to be checked. This is performed whenever the associated pointcut matches certain join points. Here, the type system verifies that the matched join point representation has the same return and exception types of the referenced JPI definition. This guarantees that the advice always receives join points of the expected type.

As explained in Section 6.1, some checks related to the return and exception types are postponed until the advice weaving process. At implementation level, this implies that we need to have advice declarations and to define some new nodes into the `AspectInfo` structure to perform such verifications. Because of abc’s architecture, we need the advice definitions to perform type-checking over join points. Without advice definitions, it is impossible to perform this task. This is in clear tension with one of our main objectives to perform modular type-checking. In the case of the base code we do not want to use any information of the “real” aspect side. To address this situation we create a synthesized aspect called `dummyAspect`. This aspect is composed of around advice definitions that bind every JPI declaration that has at least one `exhibits` clause defined in the base code. The pointcut expression associated with these advices are calculated as explained in Section 6.3.

Further, we need to introduce new nodes in the `AspectInfo` structure in order to implement the semantics of the type-checking at join point level. Concretely, we introduce `CJPAdviceDecl` which represents, in the backend, an advice declaration referring to a JPI declaration. `CJPAdviceDecl` has an attribute `postResidue` to check the exception handling. The around advices, those introduced in the `dummyAspect`, are represented by `CJPAroundAdvice` which has a `matchesAt` attribute where the return type is checked. If some join point representation does not meet the contract imposed by the referenced JPI an exception is raised, causing abc to skip the whole compilation process.

6.7 Generic Interfaces

We introduce generic JPIs in our AspectJ extension by reusing the capabilities of JastAdd to deal with Java 5 generics. This allows us to reuse all of the functionality needed to use type variables in our language and only focus on specific changes in the parser and in the static type system. Syntactically, to support generic advice, we allow type variables to be declared and used in JPIs, advice signature, `exhibits` clauses and pointcuts. Closure join points do

```

/*class definition omitted*/
<R extends Number> exhibits R MethodCall() : call(R *(..));

<R extends Number> exhibits R MethodExec() : execution(R *(..));

public static Integer messageInt() {...}

public static Object message(){...}

public void main(String[] args){
    Integer result = messageInt();
    System.out.println(message());
}

```

not need type parameters, as they are actual instances of join points, binding to the type parameters of the corresponding join point interface.

Although the introduction of generics was quite straightforward at the syntax level, we performed several changes to get this working properly. First, we changed static overloading (Section 6.5) to consider the type variables when performing the JPI type lookup. Second, since type variables are part of JPI definitions, **exhibits** clauses can use them. This implies that type variables can appear in the exhibits pointcut expressions. A type variable in a pointcut expression is treated as a normal type, but it has a special treatment when the pointcut expression is translated to the **AspectInfo** structure. As an example, the pointcut expression defined in the following program, Line 2 will be converted to **call(Number+ * (..))**.

Regarding changes in the type system, we introduce a new node in the **AspectInfo** called **GenericAroundSpec**. With this, we override the **checkTypes** method in order to redefine some type checks. In particular, we check that the return type of the join point representation must be a subtype of the type defined in the referenced JPI. As an example, the execution of the method **message** (Line 8) will be rejected because **Object** is not a subtype of **Number**. Instead, the execution of the method **messageInt** (Line 6) will be accepted because **Integer** is a subtype of **Number**.

6.8 Global pointcuts

We introduce global pointcuts to the JPIs implementation by modifying the parser, the static type system and by introducing some AST rewrites.

Syntactically, we introduce a new AST node called **GlobalJPITypeDecl** which allows attaching a pointcut expression to a JPI declaration. Also, we allow **exhibits** clauses to be declared without any pointcut expression at all. In this case, these **exhibits** clauses are created by the parser with an empty pointcut expression represented by **EmptyPointcutExpr**. We further introduce a new pointcut designator called **global** represented by the **GlobalExpr**

```

<R> jpi R MethodCall() : call(R *(..));

class Message {
    /*contains no refinement for JPI MethodCall*/
}

class Output {
    <R> exhibits R MethodCall() :
        global() && !call(String internalOuput(..));
    ...
}

```

AST node.

The type system verifies that the new built-in pointcut designator **global** is only used in the scope of an **exhibits** clause which refers to a global JPI definition.

We then implement the correct runtime semantics for global pointcuts by inlining them into every aspect and class, replacing occurrences of the **global**() pointcut by the corresponding global pointcut definition. In the case that certain class or aspect contains no refinement for a global JPI definition, like **Message** class, we treat it as if it has an **exhibits** clause with **global**() as pointcut expression. In the case of **Message** class we introduce the following **exhibits** clause: <R> **exhibits** R MethodCall(): **global**(). We can only do that with classes or aspects that have not been sealed (Section 5.2.4).

As Listing 6.4 shows, the global pointcut has been inlined in both **Message** and **Output** classes. In the former, the **exhibits** clause contains exactly the same pointcut defined in the global JPI declaration. Instead, in the latter, the **exhibits** clause contains a refined pointcut expression.

```

<R> jpi R MethodCall() : call(R *(..));

class Message {
    <R> exhibits R MethodCall() :
        call(R *(..));
}

class Output {
    <R> exhibits R MethodCall() :
        call(R *(..)) && !call(String internalOuput(..));
    ...
}

```

Listing 6.4: Inlined exhibits clause which use **global**

6.9 Reuse of implementation for closure join points

Interestingly, our abc extension for join point interfaces extends and completely reuses the original implementation of closure join points (CJPs) [Bodden, 2011]. Our JPI extension to abc can modularly define how the extension for closure join points needs to be refined to match the correct syntax and semantics that they require when used in combination with join point interfaces.

Surprisingly the integration between JPIs and closure join points revealed some corner cases that were not considered in the original design of CJPs. The most remarkable one is the relation between generic JPIs and CJPs. The main problem was that in the original type-checker implementation the return type of the CJP is inferred from the JPI definition. This is correct when considering only concrete types, however it can lead to confusing type errors in the presence of type variables. For instance, consider the following example:

```
<R> jpi R JP();  
  
class Foo {  
    void bar () {  
        Integer i = new exhibit JP() {  
            return new Float();  
        }  
    }  
}
```

We believe that the type-checker should give an error that says that the assignment from `Float` to `Integer` is incompatible. The idea is to warn the developer about uses of CJPs in incompatible contexts. However, due to implementation details, this is not the case if we infer the return type of the CJP from the JPI definition. The proper solution then is to infer the return type of the CJP from the context in which the closure join point is declared or "called".

6.10 Summary

In this chapter we have described, in full generality, how we implement Join Point Interfaces (JPIs) as an extension of the AspectBench compiler (abc) for AspectJ. We have first presented a brief description of the frontend and backend components of abc. Then, we explained step by step how each component of the abc compiler was extended in order to implement the new language's features introduced by JPIs. The next chapter presents the major conclusions of this work and the open perspectives that we foresee for future work.

Chapter 7

Conclusion

Join point interfaces enable fully modular type checking of aspect-oriented programs by establishing a clear contract between aspects and advised code. Like interfaces in statically-typed object-oriented languages, JPIs support independent development in a robust and sound manner. Key to this support is the specification of JPIs as method-like signatures with return types and checked exception types. JPIs can be organized in hierarchies to structure the space of join points in a flexible manner, enabling join point polymorphism and dynamic advice dispatch. The use of type variables in JPI definitions, along with support for controlled global quantification, allow for concise definitions of JPIs with minimal programmer effort. We have implemented JPIs as a publicly available AspectJ extension¹, and have rewritten several existing AspectJ programs to take advantage of JPIs². This study supports our major design choices.

Table 7.1 summarizes the main features of our language extension and the design decisions that justify this language design. When defining the JPI mechanism, we always opted for type safety over flexibility. As we explained earlier, in a first approach, we opted for totally invariant typing, which is safe but sometimes too inflexible. Join point polymorphism, parametric polymorphism and global pointcuts restore the necessary flexibility without jeopardizing type soundness. As a result, we have designed a language that has strong typing guarantees, allows for modular reasoning, minimizes the amount of extra code required, and can accommodate a smooth transition path from plain AspectJ.

7.1 Perspectives on quantification

While AspectJ allows global quantification (i.e., pointcuts can match join points that occur anywhere), Open Modules and JPTs have very different takes on quantification: Should classes be aware of the join points they expose? With Open Modules, classes themselves do not declare their exposed join points; it is the task of the module. The argument is that the

¹<https://svn.sable.mcgill.ca/abc/trunk/aop/abc-ja-exts/src-jpi/abc/ja/jpi/>

²<http://www.bodden.de/jpi/>

Feature	Purpose
Join point interfaces	Decoupling aspects from base code
JPIs as method signatures	Preserve procedural abstraction
Class-local pointcut matching	Limit pointcut fragility
Closure join points	Support explicit announcement
Invariant typing of arguments, return type and checked exceptions Invariant pointcuts	Preserve type soundness
Join point polymorphism	Support expressive modeling and handling of events
Parametric polymorphism	Enhance the flexibility of the type system
Controlled global pointcuts	Support wide quantification

Table 7.1: Summary of features and their purpose

maintainer of the module is in charge of all the classes inside the module, and therefore, has sufficient knowledge to maintain classes in sync with the pointcuts in the module interface. Steimann and colleagues, on the other hand, argue for class-local exhibit clauses: each class is responsible for what it exhibits. In JPTs, even nested classes are not affected by the exhibited pointcuts of their enclosing classes. Our current proposal is actually half-way between both standpoints. We do not extend Java with a new notion of modules (this is left for future work), but we do support nested classes in the sense that the exhibited pointcut of a class match join points in nested classes as well. This means that we can use class nesting as a structuring module mechanism, and obtain module-local quantification. Of course, this approach to modules is certainly not as well supported in Java as it would be in other languages, such as Newspeak [Bracha et al., 2010], where modules are objects, supported by a very flexible virtual class system.

In addition, in order to make it possible for AspectJ programmers that adopt JPIs to balance the quantification design trade-off, we have introduced global pointcuts (Section 5.2). This combination of local and global quantification, which preserves the possibility for a given class to be sealed from unwanted advising, is in itself a novel answer to the quantification design question.

Bibliography

- AJDT 2012. Eclipse AspectJ Development Tools. <http://www.eclipse.org/ajdt/> visited 2012-12-12.
- AJHotDraw 2003. The AJHotDraw home page. <http://ajhotdraw.sourceforge.net/> visited 2012-12-12.
- ALDRICH, J. 2005. Open modules: Modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, A. P. Black, Ed. Number 3586 in LNCS. Springer-Verlag, Glasgow, UK, 144–168.
- ASPECTJ TEAM. 2003. The AspectJ programming guide. <http://www.eclipse.org/aspectj/doc/released/progguide/>.
- AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, J., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. abc: An extensible AspectJ compiler. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development (AOSD 2005)*. ACM, Chicago, Illinois, USA, 87–98.
- BAGHERZADEH, M., RAJAN, H., LEAVENS, G. T., AND MOONEY, S. 2011. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. ACM, Porto de Galinhas, Brazil, 141–152.
- BerkelyDB 2010. Oracle Berkeley DB. <http://www.oracle.com/technetwork/database/berkeleydb/> visited 2012-12-12.
- BODDEN, E. 2011. Closure joinpoints: block joinpoints without surprises. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. ACM, Porto de Galinhas, Brazil, 117–128.
- BODDEN, E., TANTER, É., AND INOSTROZA, M. 2012. Safe and practical decoupling of aspects with join point interfaces. Tech. Rep. TUD-CS-2012-0106, Technical University of Darmstadt. May.
- BRACHA, G., AHE, P., BYKOV, V., KASHAI, Y., MADDUX, W., AND MIRANDA, E. 2010. Modules as objects in newspeak. In *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP 2010)*, T. D’Hondt, Ed. Number 6183 in LNCS.

Springer-Verlag, Maribor, Slovenia, 405–428.

- CLIFTON, C. AND LEAVENS, G. T. 2006. MiniMAO₁: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming* 63, 312–374.
- CLIFTON, C., LEAVENS, G. T., CHAMBERS, C., AND MILLSTEIN, T. 2000. MultiJava: Modular open classes and symmetric multiple dispatch in java. In *Proceedings of the 15th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*. ACM, Minneapolis, Minnesota, USA, 130–145.
- DE FRAINE, B., SÜDHOLT, M., AND JONCKERS, V. 2008. StrongAspectJ: flexible and safe pointcut/advice bindings. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*. ACM, Brussels, Belgium, 60–71.
- DYER, R., RAJAN, H., AND CAI, Y. 2012. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD 2012)*, É. Tanter and K. J. Sullivan, Eds. ACM, Potsdam, Germany, 143–154.
- EKMAN, T. AND HEDIN, G. 2007. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. OOPSLA '07. ACM, New York, NY, USA, 1–18.
- FABRY, J., KELLENS, A., AND DUCASSE, S. 2011. Aspectmaps: A scalable visualization of join point shadows. In *Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC2011)*. IEEE Computer Society Press, 121–130.
- FERNANDO, R. D., DYER, R., AND RAJAN, H. 2012. Event type polymorphism. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*. ACM, Potsdam, Germany, 33–38.
- FILMAN, R. E., ELRAD, T., CLARKE, S., AND AKŞIT, M., Eds. 2005. *Aspect-Oriented Software Development*. Addison-Wesley, Boston.
- GASIUNAS, V., SATABIN, L., MEZINI, M., NÚÑEZ, A., AND NOYÉ, J. 2011. EScala: modular event-driven object interactions in Scala. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. ACM, 227–240.
- GUDMUNDSON, S. AND KICZALES, G. 2001. Addressing practical software development issues in AspectJ with a pointcut interface. In *Proceedings of the Workshop on Advanced Separation of Concerns*.
- GYBELS, K. AND BRICHAU, J. 2003. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD 2003)*, M. Akşit, Ed. ACM Press, Boston, MA, USA, 60–69.
- HILSDALE, E. AND HUGUNIN, J. 2004. Advice weaving in AspectJ. In *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*,

- K. Lieberherr, Ed. ACM, Lancaster, UK, 26–35.
- HOFFMAN, K. AND EUGSTER, P. 2007. Bridging Java and AspectJ through explicit join points. In *Proceedings of the 9th International Symposium on Principles and Practice of Programming in Java (PPPJ 2007)*. ACM, 63–72.
- IGARASHI, A., PIERCE, B. C., AND WADLER, P. 2001. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems* 23, 3, 396–450.
- INOSTROZA, M., TANTER, É., AND BODDEN, E. 2011. Join point interfaces for modular reasoning in aspect-oriented programs. In *ESEC/FSE '11: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. New Ideas Track.
- JAGADEESAN, R., JEFFREY, A., AND RIELY, J. 2006. Typed parametric polymorphism for aspects. *Science of Computer Programming* 63, 267–296.
- KHATCHADOURIAN, R., GREENWOOD, P., RASHID, A., AND XU, G. 2009. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. In *International Conference on Automated Software Engineering (ASE 2009)*. IEEE/ACM, 575–579.
- KICZALES, G. AND MEZINI, M. 2005. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering (ICSE 2005)*. ACM, St. Louis, MO, USA, 49–58.
- LADDAD, R. 2003. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA.
- LIEBERHERR, K., HOLLAND, I., AND RIEL, A. 1988. Object-oriented programming: An objective sense of style. In *Proceedings of the 3rd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 88)*, N. Meyrowitz, Ed. ACM, San Diego, California, USA, 323–334. ACM SIGPLAN Notices, 23(11).
- MASUHARA, H. AND KICZALES, G. 2003. Modeling crosscutting in aspect-oriented mechanisms. Springer-Verlag, 2–28.
- MASUHARA, H., KICZALES, G., AND DUTCHYN, C. 2003. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003)*, G. Hedin, Ed. LNCS Series, vol. 2622. Springer-Verlag, 46–60.
- MASUHARA, H., TATSUZAWA, H., AND YONEZAWA, A. 2005. Aspectual Caml: an aspect-oriented functional language. In *Proceedings of the 10th ACM SIGPLAN Conference on Functional Programming (ICFP 2005)*. ACM, Tallin, Estonia, 320–330.
- ODERSKY, M., SPOON, L., AND VENNERS, B. 2008. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA.
- ONGKINGCO, N., AVGUSTINOV, P., TIBBLE, J., HENDREN, L., DE MOOR, O., AND SIT-

- TAMPALAM, G. 2006. Adding open modules to aspectj. In *Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD 2006)*. ACM, Bonn, Germany, 39–50.
- PAEPCKE, A., Ed. 1993. *Object-Oriented Programming: The CLOS Perspective*. MIT Press.
- PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12, 1053–1058.
- RAJA VALLÉE-RAI, LAURIE HENDREN, V. S. P. L. E. G. AND CO, P. 1999. Soot - a java optimization framework. In *Proceedings of CASCON 1999*. 125–135.
- RAJAN, H. AND LEAVENS, G. T. 2008. Ptolemy: A language with quantified, typed events. In *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, J. Vitek, Ed. Number 5142 in LNCS. Springer-Verlag, Paphos, Cyprus, 155–179.
- REYNOLDS, J. C. 1983. Types, abstraction, and parametric polymorphism. In *Information Processing 83*, R. E. A. Mason, Ed. Elsevier, 513–523.
- ROBILLARD, M. AND MURPHY, G. 2000. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '00/FSE-8)*. 2–10.
- STEIMANN, F. 2006. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*. ACM, Portland, Oregon, USA, 481–497.
- STEIMANN, F., PAWLITZKI, T., APEL, S., AND KÄSTNER, C. 2010. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology* 20, 1, 1–43.
- STOERZER, M. AND GRAF, J. 2005. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 653–656.
- SULLIVAN, K., GRISWOLD, W. G., RAJAN, H., SONG, Y., CAI, Y., SHONLE, M., AND TEWARI, N. 2010. Modular aspect-oriented design with XPIs. *ACM Transactions on Software Engineering and Methodology* 20, 2.
- TANTER, É. 2010. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*. ACM, Rennes and Saint Malo, France, 37–48.
- TANTER, É., GYBELS, K., DENKER, M., AND BERGEL, A. 2006. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, W. Löwe and M. Südholt, Eds. LNCS Series, vol. 4089. Springer-Verlag, Vienna, Austria, 227–242.
- TANTER, É., MORET, P., BINDER, W., AND ANSALONI, D. 2010. Composition of dynamic

- analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*. ACM, Eindhoven, The Netherlands, 113–122.
- TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, JR., S. M. 1999. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*. ICSE '99. ACM, New York, NY, USA, 107–119.
- TOLEDO, R., LEGER, P., AND TANTER, É. 2010. Aspectscript: Expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD 2010)*. ACM, 13–24.
- VALLEE-RAI, R. AND HENDREN, L. J. 1998. Jimple: Simplifying java bytecode for analyses and transformations.
- WAND, M., KICZALES, G., AND DUTCHYN, C. 2004. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems* 26, 5, 890–910.