Vrije Universiteit Brussel

FACULTY OF ENGINEERING

Language Facilities for the Deployment of Reusable Aspects

Thesis submitted in fulfilment of the requirements for the award of the degree of Doctor in de ingenieurswetenschappen (Doctor in Engineering) by

Bruno De Fraine

June 2009

Advisor(s): Prof. Dr. Viviane Jonckers
Department of Computer Science
Dr. Wim Vanderperren
Department of Computer Science

Language Facilities for the Deployment of Reusable Aspects

Bruno De Fraine

June 2009

Abstract

In the domain of aspect-oriented software development, there is a high level of interest in reusable aspect modules by both researchers and practitioners. A number of approaches target the incorporation of aspect mechanisms in a component-based style of software development. But also in the context of traditional aspect systems the concept of aspect libraries with reusable implementations of crosscutting concerns is becoming increasingly relevant.

At the level of the aspect programming language, considerable attention has been devoted to powerful abstraction mechanisms, to be able to describe aspect behavior independent of a specific context, with flexible extension points. However, a separate deployment step is also needed, to configure and activate reusable aspect behavior for a concrete setting. We claim that advanced *deployment mechanisms* are equally important to enable the intensive use and reuse of aspects. We consider two dimensions of support for aspect deployment:

- *Expressive* deployment mechanisms allow to use existing aspects in more contexts. They improve the general software engineering properties of deployment logic.
- *Safe* deployment mechanisms automatically verify the compatibility between the aspect and the context where it is used. This makes the reuse of aspects more manageable.

We find that the deployment mechanisms of current aspect approaches are lacking in these two respects, and present contributions in each area.

With respect to the expressiveness of deployment mechanisms, we analyze the current mainstream deployment constructs and identify a number of common shortcomings that relate to (i) reuse of the deployment logic, (ii) quantification of aspect behavior, and (iii) activation of new deployments at dynamic program events. We propose a language design to remedy these problems and we realize this design as the EcoSys AOP framework. EcoSys employs general program code for the specification of deployment logic, which provides rich abstraction mechanisms and control structures. It also allows for the deployment to be better integrated with the rest of the program.

For the safety of deployments, we focus on the static typing of deployment code. Compared to a full specification and verification of the program behavior, types offer a practical and lightweight approximation of the semantics. Our contribution is the development of flexible typing principles for the safe application of aspect behavior at concrete program points, based on the subtype relations that provide both an upper and lower bound for types, and the use of type variables to support genericity. We demonstrate how the type system can be integrated with the dynamic deployment mechanisms of EcoSys, thanks to generic typing features of

recent languages such as JAVA 5. The proposed typed ECOSYS approach achieves a combination of both expressive and typed aspect deployments, and demonstrates that these two properties should not be considered at odds with each other. In addition, we present the STRONGASPECTJ language, which integrates the proposed typing principles in a mainstream aspect language. Finally, we provide a formal evaluation of the proposed typing principles with a rigorous proof of the safety properties in the context of the FEATHERWEIGHT JAVA calculus.

Samenvatting

Binnen het veld van aspectgerichte softwareontwikkeling is een grote interesse in herbruikbare aspectmodules merkbaar, zowel vanwege onderzoekers als gebruikers. Een aantal benaderingen beoogt de integratie van aspectmechanismen binnen een componentgebaseerde manier van softwareontwikkeling, maar ook voor traditionele aspectsystemen wordt het concept van aspectbibliotheken met herbruikbare implementaties van veelvoorkomende aspectfunctionaliteit steeds belangrijker.

Op het niveau van de aspectprogrammeertaal werd aanzienlijke aandacht besteed aan krachtige abstractiemechanismen, om aspectgedrag onafhankelijk van een specifieke context en met flexibele aanpassingspunten te programmeren. Een afzonderlijke inzetstap is echter vereist om het herbruikbare aspectgedrag te configureren en activeren binnen een concrete omgeving. We beweren dat geavanceerde *inzetmechanismen* net zo belangrijk zijn om het intensieve gebruik en hergebruik van aspecten mogelijk te maken. We beschouwen twee belangrijke eigenschappen van de faciliteiten voor het inzetten van aspecten:

- *Expressieve* inzetmechanismen laten toe om bestaande aspecten in meer situaties te gebruiken. Ze verbeteren de algemene softwareontwikkelingseigenschappen van de inzetlogica.
- *Veilige* inzetmechanismes controleren automatisch de compatibiliteit tussen het aspect en de context waar het wordt ingezet. Dit maakt het hergebruik van aspecten makkelijker beheerbaar.

We stellen vast dat de inzetmechanismen van huidige aspectbenaderingen tekortschieten en we dragen bij tot de verbetering van elk van deze twee eigenschappen.

Om de expressiviteit van inzetmechanismen te verbeteren, analyseren we de voornaamste huidige inzetmechanismen en duiden we een aantal tekortkomingen aan die betrekking hebben op (i) het hergebruik van de inzetlogica (ii) de kwantificering van het aspectgedrag, en (iii) het inzetten van nieuwe aspecten bij dynamische programmagebeurtenissen. We stellen een taalontwerp voor om deze problemen op te lossen en we werken dit ontwerp concreet uit als het EcoSys AOP framework. EcoSys maakt gebruik van algemene programmacode om inzetlogica uit te drukken en het kan hiervoor dus rijke abstractiemechanismen en controlestructuren bieden. Deze organisatie laat ook toe om de inzetlogica beter te integreren met de rest van het programma.

Om de veiligheid van de inzetmechanismen te verbeteren, concentreren we ons op de statische typering van inzetlogica. In vergelijking met een volledige specificatie en verificatie

van het programmagedrag bieden types een praktische en eenvoudige benadering van de programmasemantiek. Onze bijdrage is de ontwikkeling van flexibele typeregels die garanderen dat aspectgedrag veilig kan toegepast worden op concrete programmapunten. Hiervoor maken we gebruik van subtyperelaties die zowel een boven- als een ondergrens voor types opgeven, en een abstractie met typevariabelen om generiek aspectgedrag mogelijk te maken. We tonen hoe dit typesysteem geïntegreerd kan worden met de dynamische inzetmechanismen van ECOSYS, door gebruik te maken van de generieke typeringsfuncties van recente programmeertalen als JAVA 5. De getypeerde variant van ECOSYS die aldus bekomen wordt, combineert zowel het expressief als veilig inzetten van aspecten, en toont aan dat deze twee eigenschappen verenigbaar zijn. Daarnaast definiëren we de STRONGASPECTJ taal die de voorgestelde typeregels integreert in een courante aspecttaal. Tot slot onderzoeken we de voorgestelde typeregels op een formele wijze in de context van de FEATHERWEIGHT JAVA calculus en leveren we een rigoureus bewijs van de veiligheidsgaranties.

Acknowledgments

The following three factors were of vital importance for the realization of this work:

– When I approached the SSEL lab as a stranger from a different university with the objective of obtaining a doctoral degree, I immediately received the confidence and support from Viviane Jonckers to make this happen. As the main adviser, she displayed tremendous practicality throughout and she was always available for constructive feedback on my work, even as I delved into more technical regions.

– For the larger half of the time, Wim Vanderperren provided me with day-to-day guidance, and I relied on his extensive know-how for more than one important decision. I appreciate that he continued his role of adviser at his new position outside of the university. From the beginning, Wim treated me as one of his peers, and the stereotype of the "evil postdoc" could not have been further from the truth in his case.

– The financial support from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) ensured that I could carry out the work in a largely carefree manner.

The quality of my thesis text improved considerably from the corrections and knowledgeable suggestions of my committee members: Jacques Tiberghien, Rik Pintelon, Theo D'Hondt, Dirk Vermeir, Erik Ernst and Ralf Lämmel. I particularly thank the external examiners for their scrutiny. Some chapters of the text were additionally proofread by courtesy of Mario Südholt and Ragnhild Van Der Straeten. In general, I thank Mario for sharing his expertise during our fruitful collaborations on this (and other) projects.

On countless occasions, I benefited from the scientific, technical and practical experience from the colleagues that precede me at the SSEL lab — not to mention the friendship I received from them. Other than Wim and Ragnhild, this includes Maja D'Hondt, Davy Suvée, Bart Verheecke, María Agustina Cibrán and Dennis Wagelaar. Much the same can be said about those colleagues that joined our group later on: Niels Joncheere, Mathieu Braem, Dirk Deridder, Mario Sánchez, Oscar González, Andrés Yie, Eline Philips and Carlos Noguera. Additionally, I greatly enjoyed the fellowship of many members of the PROG lab, too numerous to list here. Suffice it to say that a bright future lies ahead in the new constellation as the "Software Languages Lab".

On a personal note, I thank Dominique for her unconditional love and encouragement during these formative years of our joint lives. I thank my parents, sister and brother, Véronique

and Hans, friends and family for their interest and all-round support. The persistent inquiries about my progress frequently caused me to reflect on the larger context of my work when it was sorely needed.

Bruno De Fraine
June 2009

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Problem Statement

Aspect-oriented and component-based software development are two paradigms that have been proposed to cope with the complexity of present-day software. Component-based software development (Szyperski, 1998) aims at the construction of software by composing a number of off-the-shelf components. This requires highly reusable components and expressive composition mechanisms that support plug-and-play recomposition. Aspect-oriented software development (Kiczales et al., 1997a) pursues the modular treatment of specific concerns (aspects) that elude the traditional decomposition mechanisms, typically because the functionality is non-hierarchical in nature. One of the best known mechanisms to achieve the modularization of aspects in a software implementation consists in the specification of additional behavior (advice) and a description (the pointcut) of the points during the execution (join points) where the behavior needs to be included. The execution platform co-composes the aspect description with the main program (a technique called weaving).

The usefulness of combining both paradigms has been demonstrated on numerous occasions. In order to develop aspect behavior according to the component paradigm, it is crucial to achieve independent and reusable aspect specifications. This is an important facet in the work by Vanderperren (2004) or by Lieberherr et al. (1999). Outside of the component paradigm, there is also an increasing interest for libraries of reusable aspect implementations as the application of aspect technologies becomes more mature. This includes simple toolboxes with direct solutions for a broad range of typical problems (Isberg, 2006; Colyer, 2005b) as well as highly specialized catalogs of implementations for a specific problem such concurrency (Cunha et al., 2006), object relationships (Pearce and Noble, 2006) or design patterns (Hannemann and Kiczales, 2002).

At the level of the programming language, the reuse of aspects requires abstraction mechanisms, to describe aspect behavior independent of a particular context, and with flexible extension points. However, equally important to enable advanced reuse are the *deployment* mechanisms that instantiate and configure the aspects for a particular use[1]. It is not sufficiently

[1] Note that in context, we use the term 'deployment' for an activity during the software implementation, which is not

recognized that the deployment of an aspect may describe a complex configuration that involves (among other things) data exchange with the main application and interactions with other aspects. Current mainstream approaches confine aspect deployment to various configuration mechanisms with substandard language properties. We find that this second-rate treatment prohibits the intensive usage of aspect mechanisms in the implementation of complex software.

One class of the issues with current deployment mechanisms relates to their lack of expressiveness. It is impossible to abstract the commonalities in deployment specifications, which prevents any manageable reuse or quantification of aspect deployments. In addition, the means to integrate the deployment of aspects with relevant events during the execution of the program are rigid and restricted. This causes a direct failure to express a desired deployment strategy. These expressiveness problems pose a serious obstacle for the advanced use and reuse of aspects.

The other class of issues involves the verification of aspect deployments, in particular of the compatibility between the reusable aspect and the concrete context where it is deployed. We believe that an automated verification of the consistency of an aspect deployment may unburden the developer from the need to have detailed knowledge about the implementation of the employed reusable aspects, allowing him or her to reason about the application in more abstract terms. However, it is an open problem which abstract properties should be part of this verification process in order to preserve the expressiveness yet capture the essential properties such that compatibility may be conclusively determined.

1.2 Goal

The general goal of the research in this dissertation is to enable the advanced reuse of aspect implementations by providing improved programming language facilities for their deployment. This is motivated by the observation that deployment facilities are a crucial requirement for the effective reuse of aspects.

Observing the above problem description, we focus on two properties of deployment facilities for reusable aspects:

- *Expressive* deployment facilities support the abstraction and quantification of deployment behavior, and they allow a general integration of this behavior with the program logic.

- *Safe* deployment mechanisms support an automatic verification of the compatibility between the aspect and the context where it is used.

We elaborate further on these properties below. For now, we specify that we do not consider these two properties in isolation. Generally, it is harder to reason about behavioral program properties such as safety in the presence of more expressive language mechanisms. This given does not justify extremism in one or the other direction (*i.e.*, considering only expressiveness without regard for safety or the other way around). Rather, we believe that the most effective programming language design is obtained when the two properties are properly balanced.

to be confused with the use of the term to denote a separate software life cycle step which installs and activates an entire software system for concrete use.

1.3 Context

The research reported in this dissertation was carried out at the System and Software Engineering Lab (SSEL) at the Vrije Universiteit Brussel. SSEL has developed extensive expertise in software language engineering and has made numerous contributions to the fields of both component-based and aspect-oriented software development. In particular, the integration of these two paradigms was pioneered by Suvée, Vanderperren, and Jonckers (2003) with the proposal of the JAsCo aspect language. Bruno De Fraine contributed to extensions of the pointcut mechanism of JAsCo, in collaboration with Vanderperren et al. (2005a) and Benavides Navarro et al. (2006), in order to factor in temporal relations and distribution context in the activation of advice behavior. These extensions are beyond the scope of this dissertation.

Nevertheless, the experience with the JAsCo approach is the main point of departure for the personal research on language facilities for aspect deployment reported here. JAsCo places a strong emphasis on the reusability of aspect logic and employs dedicated connector entities for the deployment of aspect modules. This explicit treatment of aspect deployment brings to light the second-rate treatment of deployment logic, as the connectors lack a number of expressiveness and safety properties in comparison to the other language entities. We address these shortcomings in this dissertation. However, as we will demonstrate, these are general issues shared by all mainstream aspect approaches. Because of this wider relevance, the research is not tied to the JAsCo language.

1.4 Approach

1.4.1 Expressiveness of Deployment Logic

Current approaches organize aspect deployment by means of inheritance, with explicit connector entities or through XML configuration files. Our analysis of these solutions in this dissertation identifies shortcomings in three respects. First, when multiple aspects need to be deployed in a similar manner, then the deployment of each of them needs to be rewritten from scratch because of lacking abstraction mechanisms. Second, one cannot express quantification of deployments due to the absence of control structure: when 3 aspects each need to be deployed for 5 contexts, 15 deployment entities need to be written, which does not scale. Third, it is often necessary to deploy aspect at specific program events, but since the deployment is separated from the main program this is not always possible or fragile constructions need to be set-up in order to exchange program data.

As a solution for these problems, we propose the design of *first-class deployment procedures*. These deployment constructs can be parameterized with aspect entities such as pointcuts and advice, which allows to reuse their behavior for multiple deployments. Additionally, they offer control structures to express deployment quantification, and their invocation can be tied to run-time events in the main program. We also provide a concrete realization of this design as the EcoSys approach. Starting from the observation that first-class deployment procedures propose many facilities of standard program code, EcoSys takes this one step further and removes any distinction between deployment code and ordinary code. It provides an AOP framework where the aspect behavior, including the aspect deployment, is expressed in a

standard object-oriented programming language, using a number of predefined classes. ECOSYS realizes the benefits of first-class deployment procedures, but also provides comprehensive support for composition of advice behavior in the case where different advice instances advise the same join point. A prototype implementation of ECOSYS has been developed by using ASPECTJ and its tool chain for the atypical goal of bytecode manipulation.

Finally, we analyze a number of other, specific proposals for improvements in the field of aspect languages, where we find that these proposals provide alternative ways to address some of the shortcomings we discuss. However, none of these approaches provides a complete solution for all of the issues tackled by first-class deployment procedures and ECOSYS. In some cases, deployment is not an explicit focus of investigation, but deployment issues are at the root of many of the examples that motivate a new language mechanism or a different organization. This illustrates the breadth of deployment expressiveness issues in aspect languages.

Important parts of this work were reported previously in De Fraine et al. (2005a, 2006a,b) and De Fraine and Braem (2007).

1.4.2 Safety of Deployment Logic

For the verification of aspect deployments, we focus on static typing to determine the compatibility between the advice behavior of a reusable aspect and the pointcut that specifies concrete program points. Compared to a full specification and verification of the program behavior, types offer a lightweight approximation of the semantics: types generally constrain only the data values. Type checking can therefore be carried out automatically, as a mandatory part of the compilation phase. Static typing is a common practice in current mainstream programming and although typed aspect languages exist, we do not consider them adequate since they have important safety problems and/or constrain practical aspect deployments.

The difficulty in developing an adequate type system for the aspect mechanism is due to its expressive power: advice behavior for a join point completely controls its execution, and the advice may manipulate the data transfer between the join point and its client in both directions. Additionally, it is the nature of aspect-orientation to apply advice to many join points at once, where the join points may be different in structure. This requires that the relevant commonalities between the join point are tracked in the typing.

As a solution, this dissertation proposes typing principles based on the notions of *subtype polymorphism* and *parametric polymorphism* (as explored by Cardelli and Wegner, 1985). Subtyping is a relation between a specific type (such as “integer”) and a more general type (such as “number”), where instances of the specific type may be treated as instances of the general type as well (*i.e.*, integers may be treated as numbers). We use this concept to express relations between advice and join point, for example, when advice behavior expects to obtain a number from the join point yet itself provides an integer to the join point client instead, then the advice behavior may be safely applied to both number and integer join points, as in both cases both the assumptions of the advice and the join point are met. Alternatively, parametric polymorphism provides a complementary mechanism to express type relations for advice behavior of a different nature. When advice behavior returns an original value obtained from the join point to the join point client, then this value is always valid, regardless of its type. We track this by representing the join point type with a type parameter T and by verifying that the advice behavior both accepts a T and provides a T, for any type T. We find that both

typing mechanisms are useful for practical advice behavior and we describe how they may be combined.

In this dissertation, we further present an integration of the typing principles with concrete aspect approaches. This includes the development of a typed version of the ECOSYS approach, where the typing principles may be enforced through the generics mechanism in recent object-oriented languages (Naftalin and Wadler, 2006). Using this system, we illustrate that our typing schemes are sufficiently expressive to support a variety of practical examples. While the typed ECOSYS system achieves a combination of both expressive and safe aspect deployments, we also present an extension of the ASPECTJ language, in order to directly apply our results in a mainstream context. Finally, the dissertation contains a formal evaluation of the typing principles inside the FEATHERWEIGHT JAVA calculus, where we rigorously prove that our typing schemes prohibit safety problems.

Important parts of this work were reported previously in De Fraine et al. (2007, 2008b).

1.5 Contributions

The principal contributions of this dissertation are the following:

- An analysis of the deployment mechanisms in current mainstream aspect approaches and the identification of a set of requirements for the expressive deployment of reusable aspects.
- Proposal of the design of first-class deployment procedures to meet the identified requirements, and a concrete realization of this design as the ECOSYS AOP framework, including a prototype implementation of this system.
- An extensive discussion of the other approaches from recent literature with relevance to deployment expressiveness, and an analysis of their capabilities with respect to the identified set of requirements.
- An informal characterization of the capabilities of advice behavior and the corresponding formulation of typing principles for the pointcut/advice mechanism, integrating the mechanisms of subtype and parametric polymorphism and the combination of the two.
- An extension of the ECOSYS approach to integrate the proposed typing principles in the context of a framework-based AOP approach, employing only the facilities of JAVA 5 generics.
- The concrete STRONGASPECTJ proposal for the integration of the proposed typing principles in the traditional aspect language ASPECTJ, with an implementation of this system using the ASPECTBENCH compiler.
- A comprehensive discussion of the safety loopholes and expressiveness restrictions in ASPECTJ and other related typed aspect languages.
- A discussion of the practical applicability of the typing principles, using a variety of real-life examples and some terminology introduced by Rinard et al. (2004).

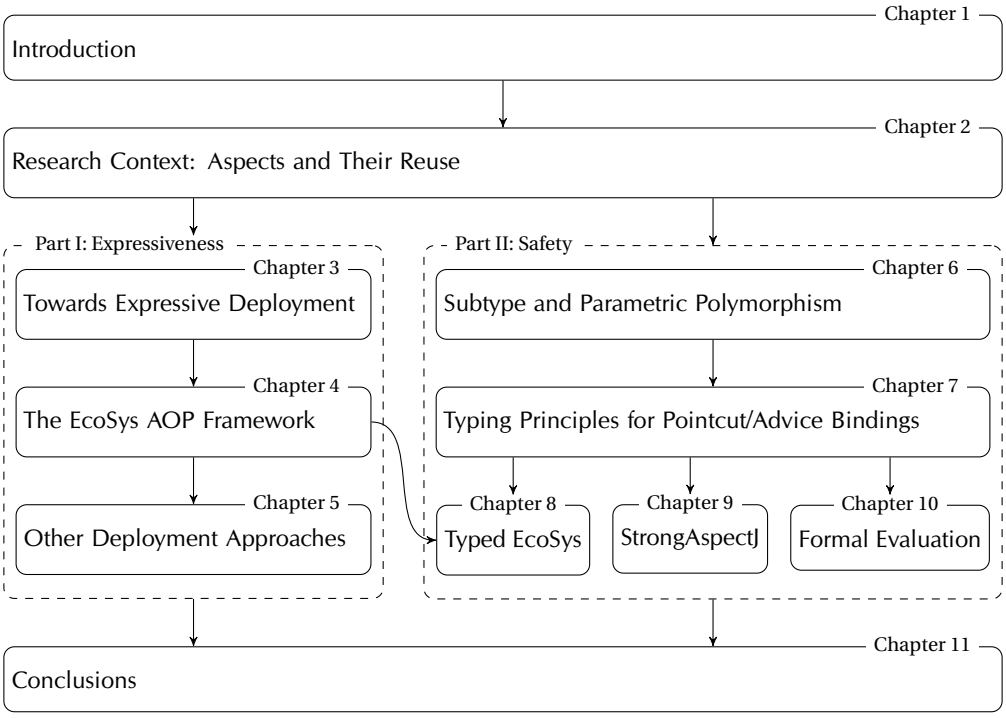


Figure 1.1: Overview of the relation between dissertation chapters. An arrow indicates when one chapter builds on the material of a previous chapter. The width of the boxes is not significant.

- A formalization of advice behavior and the proposed typing principles in the context of the FEATHERWEIGHT JAVA calculus, and a rigorous proof of the safety of this system.

1.6 Outline

Since the work in this dissertation focuses on both the expressiveness and safety facet of the aspect deployment problem, the text is correspondingly structured using two parts that are each devoted to one facet. In general, these two parts may be read independently from each other. Additionally, the second part of the dissertation consists of a general proposal of typing principles that is consecutively integrated in practical aspect approaches and a formal framework. This treatment of the same principles on a number of different levels again provides opportunities to read chapters independently from each other. The relation between the different parts and chapters of the dissertation is outlined in Figure 1.1. The connecting thread between the two parts is the ECOSYS approach, which is developed as an approach for expressive aspect deployment in the first part and extended with a typing mechanism in the second part.

The following is a more detailed description of the contents of each of the chapters:

Chapter 2: Aspects and Their Reuse In this chapter, we introduce the most important con-

cepts from the field of aspect-oriented software development. We present the major aspect-oriented mechanisms and their principal implementations, and we delineate for which of these approaches our contributions are applicable. Additionally, we discuss the major aspect module structures and reuse mechanisms in the current mainstream aspect approaches.

Chapter 3: Towards Expressive Aspect Deployment This chapter, which marks the beginning of Part I, starts with an analysis of the responsibilities of deployment logic for aspects. Next, we describe the requirements for expressive deployment, and state three expressiveness properties that are lacking in current approaches. In order to remedy these shortcomings, we propose the design of first-class deployment procedures.

Chapter 4: The EcoSys AOP Framework In this chapter, we realize the design of first-class deployment procedures in the context of an AOP framework called ECOSYS. We extensively discuss the programming interface of ECOSYS and we demonstrate how it provides improved deployment expressiveness. We also discuss the current prototype implementation of ECOSYS.

Chapter 5: Other Approaches for Expressive Deployment This chapter considers five main other approaches that provide some form of expressive deployment. We discuss the proposal of each of these approaches and evaluate it with respect our criteria for deployment expressiveness from the previous two chapters.

Chapter 6: Subtype and Parametric Polymorphism This chapter introduces the necessary type system concepts; it marks the beginning of Part II. We explain how typing provides a static abstraction of the program semantics, and we define what type soundness means. The main body of the chapter is then devoted to a presentation of the mechanisms of subtype polymorphism and parametric polymorphism in the context of a simplified functional programming language. Finally, we explain how parametric polymorphism is realized in the generics features of recent object-oriented programming languages.

Chapter 7: Typing Principles for Pointcut/Advice Bindings In this chapter, we propose principles for the flexible typing of the pointcut/advice mechanism in the context of the functional language from Chapter 6. We characterize the general capabilities of advice behavior and we informally consider a sufficient condition for the type soundness. From this condition, we derive typing rules using both subtype and parametric polymorphism, which introduces the concepts of ordinary advice and generic advice. We also elaborate on the relation between the two. Because of its particular relevance in later chapters, we specialize the typing rules for the case of advice for function join points.

Chapter 8: Safe Deployment Logic in EcoSys This chapter applies the typing principles from Chapter 7 to the ECOSYS approach from Chapter 4. Since ECOSYS is an AOP framework where the deployment logic is expressed as standard program code and compiled using an ordinary compiler, we encode the typing rules using the generics features of JAVA 5. Using the typed ECOSYS approach, we illustrate that the typing principles are sufficiently powerful to support a variety of practical advice behavior.

Chapter 9: Recovering Mainstream AOP Type Safety In this chapter, we demonstrate how the typing principles from Chapter 7 may also be applied in traditional AOP languages. We present an extension of the ASPECTJ language, named STRONGASPECTJ, and we discuss its implementation. We discuss the relation to standard ASPECTJ typing and illustrate how each difference causes either safety problems or restricted power in the case of ASPECTJ. We also consider other related work in the area of typed aspect languages.

Chapter 10: Formal Evaluation of Pointcut/Advice Bindings This chapter presents a formal evaluation of the typing principles from Chapter 7 in the context of the FEATHERWEIGHT JAVA calculus. We develop a calculus where the most general mechanisms we have discussed are captured in their essential form and we describe how the type soundness is proved in a formal and rigorous manner. We also discuss the related work in the area of formal foundations of AOP.

Chapter 11: Conclusion This chapter concludes the dissertation. An overview of the presented work is given and future work is discussed.

Chapter 2

Research Context: Aspects and Their Reuse

2.1 Aspect-Oriented Software Development

The field of aspect-oriented software development is generally concerned with novel notions of modularity that crosscut traditional abstraction boundaries. Section 2.1.1 explains the modularity problems in current programming paradigms — with typically hierarchical decomposition mechanisms — and introduces common terminology from aspect-oriented approaches. While there is general agreement on the problems and purpose of aspect-orientation, different solutions have been proposed to achieve this goal. A representative set of four main aspect-oriented composition mechanisms is presented and compared in Section 2.1.2. This discussion allows to explain in clear terms for which aspect approaches the contributions in this dissertation are applicable.

2.1.1 Modularizing Crosscutting Concerns

As with the construction of any nontrivial system, the process of software development is concerned with many things. The majority of concerns originate from the complex functionality that the software is required to provide, but requirements may also relate to the general quality of its service (such as the system's robustness or performance), or to its life cycle (e.g. the support for evolution or reuse).

It is a fundamental and long-standing assumption of software engineering that development greatly benefits from the identification of the software parts related to each concern, and the ability to treat these parts in isolation. This property, which was referred to as the *separation of concerns* by Dijkstra (1982), is traditionally achieved by decomposing the software's implementation in modules (also see Parnas, 1972). Each concern is encapsulated in a distinct unit with a well-defined interface, allowing for code that is easier to develop and maintain, and that has a greater potential for reuse.

However, a clean modularization is only possible insofar as the implementation techniques

provide abstraction and composition mechanisms that support the natural units of concern. Most existing programming languages (including object-oriented, procedural and functional languages) all have their key mechanisms rooted in some form of *generalized procedure*: *e.g.*, method, function, service,... While composition mechanisms based on generalized procedure calling are very well suited for a hierarchical breakdown of the system in functional units, Kiczales et al. (1997b) argue that they cannot cleanly encapsulate those properties that *crosscut* the functionality hierarchy (*i.e.*, that fundamentally involve different parts of this hierarchy)[1].

Such crosscutting concerns, which are also called *aspects*, require a composition different from the generalized procedure calling provided in existing languages, yet coordinated with it. Since such a mechanism is not provided by the programming language, the programmer must co-compose the aspects manually, a process which is error-prone and which causes the aspect concerns to end up "scattered" across modules and "tangled" with one another.

Example. A good illustration of the discussed modularity problems was presented by Kiczales et al. (2001b). In a case study, several concerns are investigated in the implementation of the Tomcat web server. In the visualization of Figure 2.1, the different modules (in this case classes) are represented by vertical bars with a length proportional to the size of their code. While some concerns, such as URL pattern matching, are well modularized in dedicated classes (see Figure 2.1a), others concerns, such as logging, appear scattered across multiple implementation modules (see Figure 2.1b). Although the functionality to implement that particular service may be located in a single module, invocations of this code still appear scattered across many modules. The concern fundamentally crosscuts the functional decomposition of the application and is therefore a good example of an aspect.

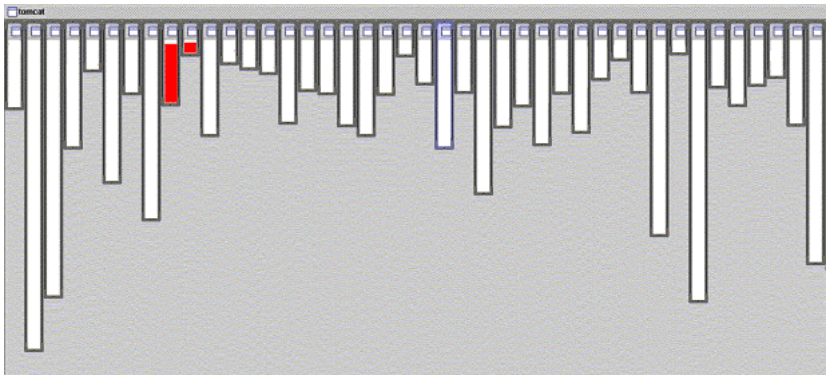
Many other classic aspect examples from literature also affect the performance or semantics of the functional concerns in systemic ways: program monitoring (Bodkin, 2005), loop optimization (Harbulot and Gurd, 2006), synchronization of concurrent objects (Cunha et al., 2006), result caching (Colyer, 2004), failure handling (Laddad, 2003, Sec. 3.2.8), enforcement of security policies (De Win et al., 2001), *etc.*

The goal of *aspect-oriented software development* (AOSD) (Kiczales et al., 1997b) is to support the developer in cleanly separating and encapsulating all concerns, by providing mechanisms that are capable of abstracting the aspects and composing them. (The term *aspect-oriented programming* (AOP) refers to doing so in the software's implementation, using primarily language changes, while AOSD considers the entire software life cycle and all the involved languages, methods and tools.)

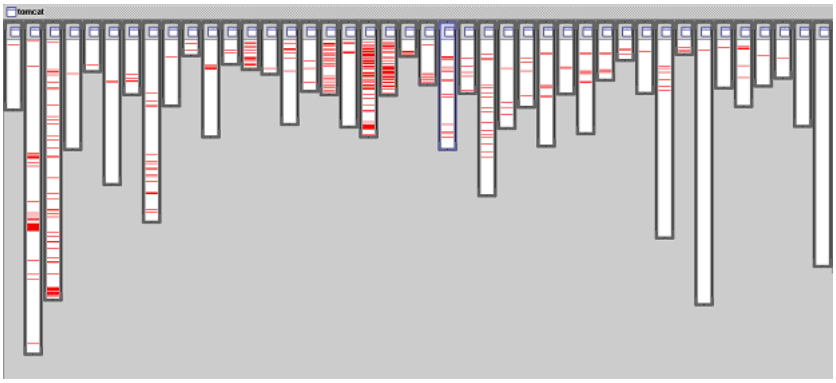
In order to achieve this goal, AOP languages generally complement a traditional *base language* (or component language) with one or more aspect languages to program the aspects. An aspect program is able to coordinate with certain elements of the base language, called *join points*. The join points can be explicit constructs of the base language[2], as well as implicit properties that appear only in the base language's semantics. An *aspect weaver* will process the base and aspect languages, co-composing them properly to produce the total system operation.

[1] This encapsulation problem is generalized by Tarr et al. (1999): they claim that while current approaches might allow multiple decompositions (*e.g.*, the structural decomposition of object-oriented style versus the behavioral decomposition of functional style), only one decomposition can be used at a time.

[2] Filman and Friedman (2000) additionally argue that the ability to coordinate with join points that do not explicitly refer to (or exist solely for the purpose of) the aspects is the feature that discriminates AOP mechanisms from generalized procedures. This property of an aspect language is referred to as *obliviousness*.



(a) URL pattern matching



(b) Logging

Figure 2.1: Modularization of two concerns in the implementation of the Tomcat web server. Figures adapted from Kiczales et al. (2001b).

```
1 after(FigureElement fe):
2     call(void FigureElement+.set*(..)) && target(fe) {
3   Display.update(fe);
4 }
```

Listing 2.1: Advice declaration to trigger display updates after figure element changes

The concrete realization of these concepts differs greatly between different proposals. This is discussed in more detail in the following section.

2.1.2 Aspect-Oriented Mechanisms

We describe four representative techniques that allow the modular implementation of aspects, and we discuss the prototypical aspect languages that implement these mechanisms. The identification of these four mechanisms is originally by Masuhara and Kiczales (2003). This discussion provides a reasonably complete overview of the spectrum of *general-purpose* aspect approaches, *i.e.*, languages that are not designed for one particular concern (as opposed to *domain-specific* aspect approaches, which are not discussed here).

Pointcuts and Advice

The *pointcut/advice mechanism* is probably the AOP technique that is most widely used today. It is the first of two mechanisms provided in the well-known ASPECTJ language[3] by Kiczales et al. (2001a); Colyer (2005a). This particular mechanism allows to intervene with the dynamic control flow of the base program using two new constructs, called *pointcut* and *advice*. The pointcut selects a set of join points, which are nodes in the execution tree of the program, while the advice specifies preceding, succeeding or replacing behavior to be executed at these coordination points. As such the behavior of aspects can be co-composed with the base program.

Example. Listing 2.1 shows an implementation of an event detector using the pointcut and advice constructs in the ASPECTJ language. The purpose of this code is to trigger display updates when figure elements are changed. The invocation of the update behavior is specified as the body (line 3) of an *advice method* (lines 1–4), which differs from a method in the base language (JAVA in this case) primarily in the characteristic that it is invoked implicitly: instead of a method name, the declaration specifies a *pointcut expression* (line 2) and indicates the relative position of the new behavior using a keyword (**after** on line 1). The advice weaver will arrange for the execution of the method body at the join points selected by the pointcut expression.

In most approaches, the pointcuts are specified in a declarative fashion. In ASPECTJ, pointcut expressions are logic propositions built up by connecting a number of primitives (such as **call** to select method calls) which are often parameterized with method, type or identifier patterns, or in some cases even with other pointcut expressions. This declarative notation allows to select join points using very succinct expressions. For example, the first part of the pointcut expression in Listing 2.1 selects the invocation of methods on objects of type `FigureElement`

[3] The pointcut/advice mechanism is also referred to as "dynamic crosscutting" in ASPECTJ documentation.

or its subtypes, where the method name starts with "set" and there are no return values and any number of argument values.

The example also illustrates a feature of pointcuts which we have not discussed yet: they may additionally indicate certain parameters from the join point's context to be used as arguments for the advice method (a technique called *context exposure*). In Listing 2.1, this is the case for the receiver of the method call (as indicated by the usage of the primitive **`target`**); this particular figure element is used to bind the method argument with name `fe`.

Beside ASPECTJ, a large number of AOP approaches have adopted the pointcut/advice mechanism. For example, the vast majority of the languages discussed in the survey by Brichau et al. (2005) employ a pointcut/advice mechanism for dynamic join points, either exclusively or in combination with other mechanisms. Additionally, we observe that the *composition filters* model by Bergmans and Akşit (2001) is technically a pointcut/advice mechanism for one important kind of join points: the model allows to select incoming and outgoing message sends, and advise them with filters that can inspect and modify information about the message (such as the target, selector, arguments and sender) in some reified form. In their study of aspect-oriented mechanisms, Masuhara and Kiczales (2003) also note that composition filters appear to be a variant of the pointcut/advice mechanism.

Traversals

The *Demeter method* and related systems (Lieberherr, 1996) provide a mechanism that enables programmers to implement functionality as traversals through object graphs in a structure-shy fashion. This is an aspect-oriented mechanism since it allows to modularize behavior that crosscuts the structural decomposition of an object-oriented application.

In contrast to standard behavioral decompositions, traversals avoid scattering the structural information across different functions by confining the knowledge of the graph structure to specific, minimal statements. They can thus obey the *Law of Demeter* (Lieberherr and Holland, 1989), which states that each unit should have minimal knowledge about the program structure (this can be regarded as a special case of the low-coupling principle introduced by Stevens et al., 1974). Functionality implemented as a traversal is therefore immune to most effects of changing the object relationships. The technique is sometimes called *adaptive programming* for this reason.

Example. In Listing 2.2, we show an example of a traversal specification using DJ, a pure-JAVA package for adaptive programming by Orleans and Lieberherr (2001). The code fragment implements the behavior of counting all direct and indirect figure elements in a figure using a *traversal specification* `s`, which is a high-level descriptions of all objects to be visited, and an *adaptive visitor* `v`, which defines the behavior at each traversed object (or `host`). These two specification elements, together with the starting object, are sufficient input for the `traverse` method from the DJ library to execute the traversal: the abstract traversal specification is completed with the reachability information from the class graph (discovered through program reflection) to determine the actual range of the traversal.

Since this operation combines data from many objects in the object graph, its implementation using ordinary object methods would be scattered across the different participating classes. And while it is possible to employ a more functional style and centralize the operation

```
1  class Figure {
2      static ClassGraph cg = new ClassGraph();
3
4      int countFigureElements() {
5          String s = "from Figure to FigureElement";
6          Visitor v = new Visitor() {
7              int count;
8              void start() { count = 0; }
9              void before(FigureElement host) { count++; }
10             Object getReturnValue { return count; }
11         }
12         return (Integer) cg.traverse(this,s,v);
13     }
14 }
```

Listing 2.2: Counting figure elements using a traversal

in bigger methods that retrieve data from (multiple levels of) associated objects, this design implies a violation of the encapsulation principle, with detrimental effect on the adaptability of the application: as the methods encode information about the structure of the application, any changes to the object relationships will likely break the implementation of these methods. Traversals alleviate this issue of behavioral decompositions by encoding only a minimum of structural information in the operations[4].

Using aspect-oriented terminology, we conclude that traversals offer an aspect language that allows to modularize concerns that crosscut the structural decomposition of an object-oriented program. The aspects are composed with this base program during the process of an object graph traversal, where the arrival at objects constitutes a join point where aspect behavior is considered. While the nature of this join point is very different from the dynamic execution join point of the pointcut/advice mechanism, the two mechanisms are generally considered complementary. For example, the JAsCo language was extended by Vanderperren et al. (2005b) to support a traversal mechanism in which its advice construct corresponds in function to an adaptive visitor.

Alternatively, Lämmel et al. (2003) present a method of realizing structure-shy traversals using the concepts of *strategic programming*, an approach which originates from the domain of program transformation and analysis. Strategic programming organizes traversals by separating the problem-specific actions from the reusable traversal schemes, which are constructed using an algebra of predefined strategies and strategy combinators. Lämmel et al. demonstrate how the traversal specifications of the Demeter method may be encoded as traversal schemes, and thus gain reusability and flexible variation points. In addition, the different incarnations of strategic programming demonstrate that the traversal concept is not tied to object graphs only.

[4] Traversals are thus similar to the *Visitor design pattern* described by Gamma et al. (1995, p. 331–344), but without the overhead of manual creation and maintenance of "accept" methods in the object structure.

```
1  class Observable {
2      Display display;
3      void moved() {
4          display.update(this);
5      }
6  }
7
8  package figures : Kernel;
9  package display : Display;
10
11 hypermodule UpdatingDisplay
12     hyperslices: Kernel, Display;
13     relationships:
14         mergeByName;
15         equate class Kernel.FigureElement, Display.Observable;
16         bracket "{Point,Line}"."set*"
17             after Display.Observable.moved();
18 end hypermodule;
```

Listing 2.3: Class composition to trigger display updates after figure element changes

Program Composition

The HYPER/J system by Tarr and Ossher (2001) provides mechanisms to compose programs from a number of self-contained (partial) programs. The approach is motivated by the observation that the modular implementation of crosscutting concerns requires multiple decompositions to be used *simultaneously, i.e.,* without one decomposition dominating the other(s) (Tarr et al., 1999). Different overlapping partial versions of program parts (called *hyperslices*) are therefore constructed to enable multiple decompositions, while a *composition rule* specifies how to automatically combine them in a resulting program or program part. The combination of a set of hyperslices and their composition rule is also called a *hypermodule*. The authors refer to this approach as *multi-dimensional separation of concerns* and consider HYPER/J the concrete incarnation of these ideas for JAVA.

Example. Listing 2.3 presents an implementation of the event detector from Listing 2.1 using class composition in HYPER/J. In lines 1–6, the display update behavior is implemented as a small independent JAVA program. Two hyperslices are then defined using the concern mapping declarations in lines 8–9: the hyperslice `Kernel` contains the main figure classes (from package `figures`), and the hyperslice `Display` contains the new class regarding the display (from package `display`). Finally, the hypermodule declaration in lines 11–18 will effectuate the composition of the classes. The relationships section specifies that units with identical names are to be merged (**`mergeByName`** strategy), but additionally configures an **`equate`** relationship between the classes `FigureElement` and `Observable`, causing them to be merged as well. Lastly, the **`bracket`** relationship specifies that the execution of "setter" methods from `Point` and `Line` should be followed by an invocation of the method `moved`. This realizes the behavior

of triggering display updates after figure element changes in the composed classes.

While the program composition mechanism from HYPER/J realizes the example behavior similar to the pointcut/advice mechanism, it differs in two important respects:

1. The base classes and aspects are expressed in the same language, and a separate entity, expressed in a metalanguage for matching and merging, is used to specify their composition. In fact, there is no distinction between base classes and aspects[5], and HYPER/J is generally considered a *symmetrical AOP approach* for this reason.

2. The join points for the composition process are static elements of the program structure: the matching and merging composition rules refer to declarations in the composed program.

Open Classes

The *open class* mechanism enables declaration of class features (methods, fields, supertypes) outside of the textual body of the class declaration[6]. This is the second mechanism offered by the ASPECTJ language[7]. Similar to traversals and visitors, it allows to modularize behavior that crosscuts the structural decomposition of the class hierarchy (although not in a structure-shy fashion: it does not prevent the scattering of structural information in the external definitions). Additionally, the open class mechanism supports the addition of state to support the newly introduced behavior.

Example. Listing 2.4 demonstrates these properties by presenting another implementation of figure element counting behavior using the open class mechanism in ASPECTJ. The mechanism simulates the effect of adding the counting functionality to all `FigureElement` subclasses, while keeping the code of the concern textually localized.

First, the interface with which to invoke the counting behavior is declared in line 1. This interface is then added as a supertype of the abstract class `FigureElement` (line 2). As with an ordinary "implements" relation, each concrete subclass of `FigureElement` is required to have an implementation for the method declared in the interface. The declaration of this method is shown for class `Point` in line 4 and for class `Group` in lines 9–17. Being a composite figure element, the implementation for a `Group` returns the sum of the count of each of its children, and a cache is used to remember the result of this recursive calculation.

We added this memoization functionality to demonstrate that it is also possible to introduce additional state in classes: the cached data value (and a flag regarding its validity) are stored for each `Group` object by declaring new fields for this class in lines 6–7. It would have been much harder to add this functionality to the traversal implementation of this counting behavior (*cf.*

[5] Although HYPER/J programs typically identify one core (or kernel) hyperslice that can be considered the base program (other hyperslices will typically overlap with units from this kernel).

[6] This design already existed unrelated to the goals of AOP: it is supported in the "fragment language" for modularization of BETA program (Kristensen et al., 1983), and some LISP object systems, such as FLAVORS (Moon, 1986) and CLOS (DeMichiel and Gabriel, 1987), always have methods declarations external to the class declaration. It also arises naturally when adopting multiple dispatch, since methods can no longer be logically tied to a single class. The term "open class" was introduced in the context of a multi-dispatch system by Clifton et al. (2000).

[7] The open class mechanism is referred to as "static crosscutting", "inter-type declaration" and "feature introduction" in the ASPECTJ documentation.

```
interface Countable { int getCount(); }
declare parents: FigureElement implements Countable;

int Point.getCount() { return 1; }

boolean Group.hasValidCache = false;
int Group.cache;

int Group.getCount() {
    if (!hasValidCache) {
        cache = 0;
        for(FigureElement fe: elements)
            cache += fe.getCount();
        hasValidCache = true;
    }
    return cache;
}
```

Listing 2.4: Open class extension to count figure elements

Listing 2.2), since that mechanism does not directly support the addition of state (and an explicit map would need to be used to associate new data values with the objects). The downside of the open class mechanism is that external declarations spread structural information about the classes over multiple places. For example, if the association between a group and its element is changed, this counting functionality would need to be updated as well.

Like traversals, open classes are considered complementary to the pointcut/advice mechanism. An aspect may combine both mechanisms, and can for example have an advice that use class features that were themselves introduced by the aspect. In addition to ASPECTJ, both JASCO and JBOSS/AOP also offer an open class mechanism. In both cases this mechanism is formulated in terms of *mixin composition* (Bracha and Cook, 1990): the new features for a class are jointly specified as a single mixin class (also called an *abstract subclass*, *i.e.*, a class whose parent is given as a formal parameter); one or more classes can be designated to receive the features of the mixin class (*i.e.*, they are each implicitly specialized by the mixin class).

Scope of the Dissertation

The remainder of this dissertation focuses on the pointcut/advice mechanism exclusively. In general, our contributions are applicable for any approach that implements this mechanism. Because of the widespread usage of pointcuts and advice for run-time join points, this includes the majority of aspect approaches. In addition, because of the highlighted similarities with some of the other aspect-oriented mechanisms (for example, traversals), our conclusions may be valid in those contexts as well. However, this topic was not further researched.

```
1  aspect NoNullPrint {
2      pointcut appScope():
3          within(com.company.demo..*) || within(com.company.core..*);
4
5      before(Object o): call(void PrintStream.print*(Object))
6        && args(o) && appScope() {
7          if(o == null)
8              throw new IllegalArgumentException();
9      }
10 }
```

Listing 2.5: An aspect to check the argument of `print` invocations for **`null`** values

2.2 Reuse in Representative Pointcut/Advice Approaches

Aspect modules So far, the description of the pointcut/advice mechanism has focused on the ability to automatically compose functionality with the base program. By untangling the implementation of aspects from the code of base concerns, the modularity of the base concerns is preserved. However, for the modularization of the aspect concern, a simple advice method is insufficient for any but the most trivial aspects. Most pointcut/advice approaches therefore allow to group aspect functionality in an *aspect module* or simply an *aspect* (note that the latter term is used both for an aspect concern and its principal implementation entity).

For almost all pointcut/advice approaches that target an object-oriented base language, the aspect module construct is modeled heavily after the concept of objects and classes. Typically, aspect modules may define multiple advice methods and ordinary methods, as well as instance variables whose value is maintained between invocations of these methods. At run-time, multiple *aspect instances* may exist for one aspect, each with its own state (*i.e.*, its own set of values for the declared instance variables). The advice body is run in the context of a particular aspect instance, in a similar sense that object-oriented languages run a method body in the context of an object.

In the following sections, we will discuss the possibilities for reuse of aspect modules as a whole, or the sharing of pointcut and advice entities within or between aspect modules.

2.2.1 AspectJ

The aspect module of ASPECTJ (Kiczales et al., 2001a; Colyer, 2005a) is simply called an *aspect*. It is largely analogous to a class in standard JAVA, but it may contain advice methods. The aspect instances are automatically created and associated with advised join points, in order to have an aspect instance available when an advice method is invoked. The default *instantiation strategy* is simply to have one global instance of the aspect. ASPECTJ allows the reuse of both pointcut and advice declarations inside aspect definitions.

```
1  abstract aspect Counter {
2      int count = 0;
3      abstract pointcut toCount();
4      before(): toCount() { count++; }
5  }
6
7  aspect MyCounter extends Counter {
8      pointcut toCount():
9          execution(void MyClass.expensive*(..));
10 }
```

Listing 2.6: Aspect behavior to count the number of join point executions (abstract aspect and concrete subaspect)

Named pointcuts Pointcut expressions can be *named* and referenced from other pointcut expressions through this name. This is demonstrated in Listing 2.5: the `appScope` pointcut defines the extent of the in-house developed code in an application by selecting all join points in the types of certain packages. The advice in lines 5–9 combines this pointcut with other criteria to check a precondition for the invocation of library methods in this scope. The `appScope` pointcut is a complex expression that might evolve in later revisions of the software. By defining it separately it may be shared between several advice methods. The pointcut may be refined by combining it with other pointcut expressions through the logical operators `&&`, `||` and `!`.

The usage of pointcut `appScope` is not confined to aspect `NoNullPrint`. Much like static members, named pointcuts may be referenced from other aspects by prefixing them with the aspect name (*i.e.*, by referencing the pointcut as `NoNullPrint.appScope` in the case of our example). Similar to other members, access modifiers may be used to control the accessibility of pointcuts from other types. The technique of referencing pointcuts from other aspects is employed in a common idiom that is termed the *Border Control design pattern* by Miles (2004, Sec. 23.3). The idiom consists in defining an aspect with only a number of named pointcuts. Each of these pointcuts defines an application scope which can be shared between different aspect definitions with effective advice methods. As such, the definition of these program scopes can be maintained in one place.

Abstract aspects, inheritance and refinement Contrary to pointcuts, the reuse of advice methods is not done through simple referencing. As explained, advice methods are anonymous entities that specify themselves where to be invoked through a pointcut expression. The definition of an advice method is already the last step in the specification of the advice behavior: any regular aspect is implicitly instantiated and all of the advice methods are woven (the aspect is *deployed*). To prevent this, an aspect can be declared *abstract*, which will inhibit its instantiation and the application of its advice methods. However, other aspects may inherit functionality from an abstract aspect: any aspect may designate a class or an abstract aspect (but not a concrete aspect) as its parent. When multiple concrete subaspects inherit from the same abstract aspect, each of the subaspects is implicitly instantiated, and the advice methods

```
1  class Counter {
2      int count = 0;
3      hook CountHook {
4          CountHook(method(..args)) { execution(method(args)); }
5          before() { global.count++; }
6      }
7  }
8
9  static connector CountDeployer {
10     Counter.CountHook c =
11         new Counter.CountHook(void Component.expensive*(*));
12 }
```

Listing 2.7: Aspect behavior to count the number of join point executions (aspect bean and connector)

from the abstract aspect will be executed for each of the different subaspect instances. As such, the advice declaration may be shared between multiple aspects.

In general though, it is not very useful to deploy multiple aspect instances with exactly the same advice definitions. It is much more common to use aspect instances in different contexts with an adapted definition (or at least a different pointcut expression). This is supported in ASPECTJ through the overriding of aspect members in subaspects. Specifically, named pointcuts can be overridden, and the reference to named pointcuts from advice methods is late bound: in subaspects, inherited advice methods will employ the overriding definition of named pointcuts. In fact, the definition of the named pointcut may be omitted altogether from the parent by declaring it as an *abstract pointcut*. In that case, the compiler will enforce that each concrete subaspect provides a definition of the pointcut.

The standard manner to share an advice method between different aspect modules is therefore to bind the advice to an abstract pointcut in an abstract aspect, and have the different aspect modules inherit from this abstract aspect. This is illustrated for a very simple advice method in Listing 2.6. The abstract pointcut must be refined in each of the subaspects (*e.g.*, in lines 8–9 for aspect `MyCounter`) to determine where the advice should be applied.

2.2.2 JAsCo

The JASCO language is proposed by Suvée et al. (2003) as a general aspect-oriented approach tailored for component-based software development (in the sense of Szyperski, 1998). One of the main objectives of this work, is to enable common component properties (such as a high degree of reusability, plug-and-play composition, *etc.*) for aspect modules. JASCO therefore also contains mechanisms to reuse aspect behavior in different contexts.

Aspect beans and hooks The aspect module construct of JASCO is the *aspect bean*, which is modeled after a JAVABEAN component. In addition to the regular component interactions, an

aspect bean is capable of describing aspect interactions through a pointcut/advice mechanism. To this end, the aspect bean may contain one or more *hooks*, which provide functionality similar to advice definitions in ASPECTJ.

An example of an aspect bean with a hook is shown in lines 1–7 of Listing 2.7. The hook declaration (lines 3–6) has a syntactic form similar to the declaration of an inner class and consists of two parts. The hook constructor (line 4) specifies a pointcut expression in terms of an abstract method parameter (the syntax of pointcut expressions is otherwise similar to ASPECTJ). The hook methods (line 5) specify advice behavior for the join points selected by this pointcut.

Connectors Component approaches provide expressive composition mechanisms that are available in separate *connector* entities. Taking inspiration from the *aspectual components* proposed by Lieberherr et al. (1999), JASCO employs connector entities to deploy aspect beans, and to connect them to components. The corresponding connector is shown in lines 9–12 of Listing 2.7. In the body of the connector, hook instances can be created and configured. The hook instantiation will provide a method pattern as the actual argument for the hook constructor. Together with the pointcut expression from the declaration of the hook constructor, this will determine where the advice behavior of the hook is applied. Naturally, several instances of the same hook can be defined, and these instances will share the specification of advice behavior from the hook.

Compared to mechanism of advice code reuse in ASPECTJ, we encounter largely the same capabilities in JASCO. An abstract aspect is similar to an aspect bean, and its advice methods with abstract pointcuts are comparable to hooks. The concrete subaspect fulfills the role of the connector that deploys the advice behavior in a concrete context. The remaining differences between the two approaches are minor. Aspect beans are always independent of a concrete context, while aspect needs to be defined specifically with an abstract pointcut. And while abstract pointcuts are specialized with a pointcut expression, a hook is instantiated with a pointcut designator argument (the method pattern). Typically, a hook will predefine a pointcut that is a tiny bit more specific than a corresponding advice method with an abstract pointcut. For example, when comparing Listing 2.6 and Listing 2.7, the `Counter` aspect bean will predefine the matching of method execution join points, while the specification of the join point kind is delayed until the subaspect for the abstract `Counter` aspect.

2.2.3 AOP Approaches for Enterprise Middleware Frameworks

A number of aspect-oriented approaches based on the pointcut/advice mechanism have been proposed in the context of enterprise middleware frameworks. The textbook on the subject by Pawlak, Retaillé, and Seinturier (2005) discusses the approaches JAC (Pawlak et al., 2001), JBOSS/AOP (Burke et al., 2004) and SPRING/AOP (Johnson et al., 2004). Additionally, the ASPECTWERKZ approach by Bonér and Vasseur (2004) was incorporated with the ASPECTJ tools and became available as framework variant of this language with the release of ASPECTJ 5 (Colyer et al., 2005, Chap. 9). All of these approaches provide comparable framework-based mechanisms for the definition of reusable aspects.

```
1  import org.jboss.aop.joinpoint.Invocation;
2  import org.jboss.aop.advice.Interceptor;
3
4  class Counter implements Interceptor {
5      int count = 0;
6      Object invoke(Invocation inv) {
7          count++;
8          return inv.invokeNext();
9      }
10 }
```

Listing 2.8: Interceptor to count the number of join point executions

Enterprise frameworks such as JAVA EE offer a managed, server-side component architecture for the development of software applications. In this architecture, a piece of middleware (an *application server*) will host the software components and offer a number of services that typically include a transparent distribution, persistent storage of data, enforcement of security policies, and data integrity through transactions. The aforementioned AOP approaches integrate aspect modules in a manner that is similar to the definition of components in these enterprise frameworks. The advice code is specified as standard JAVA classes that must either implement a predefined interface, or that must be decorated with predefined annotations. The pointcuts and other weaving information is extracted from XML configuration files, or from parameters in the annotations. The weaving is typically carried out at load-time by means of a dedicated class loader.

An example advice definition for JBOSS/AOP is shown in Listing 2.8. In this approach, advice behavior is specified as a class that implements the predefined interface `Interceptor`. This interface contains only the `invoke` method (implemented in lines 6–9). It receives a representation of the join point as an argument of type `Invocation` and implements the actual advice behavior. The accompanying XML configuration is shown in Listing 2.9. This configuration file references the class with the advice behavior and binds it to a pointcut expression. It may also configure other properties of the deployment, for example, how the interceptor class should be instantiated.

Through the separation of the advice definition and the advice application, a reuse of the aspect specification similar to ASPECTJ and JASCO can be obtained. Although not fundamentally different, the approach of specifying aspect behavior as plain JAVA code with configuration data is perceived as a large benefit, because the standard compiler tool chain does not need to be altered, and because the aspects can be reconfigured without recompilation.

```
<aop>
    <bind pointcut="execution(void MyClass->expensive*(..))">
        <interceptor class="Counter" />
    </bind>
</aop>
```

Listing 2.9: XML configuration file to apply the `Counter` interceptor

Part I

Expressiveness of Deployment Logic

Chapter 3

Towards Expressive Aspect Deployment

The usage of reusable aspects necessarily involves a separate deployment step during which the aspects are configured for a concrete application at hand. As explained in Section 2.2, the deployment logic specified to this end can take various forms, e.g. subaspects, explicit connector entities, or XML configuration files. We claim that the current means for the specification of deployment logic are insufficiently expressive to (re)use aspects intensively when building a software application. In this chapter, we analyze the different responsibilities of deployment logic, and discuss the requirements that we identified to create more opportunities for reuse and thus improve maintainability. We then propose to meet these requirements by specifying deployment logic as procedures that borrow a number of properties from ordinary code.

Representative case Throughout the discussion, we will use a small but representative example of a reusable aspect. The aspect is presented in the AspectJ language in Listing 3.1; it implements functionality to print an execution trace of the program on a given output handle. More concretely, the aspect declares an abstract pointcut (line 8) to select join points that expose an object as a context value. An advice (lines 10–12) specifies that, before entering these join points, a description of the exposed object has to be printed. The other members of the aspect manage the output stream to which the tracing messages can be sent.

Additionally, the aspect in Listing 3.2 specifies a number of important pointcuts of the application. We place them in a separate aspect since we have the intention of sharing them between different aspects. The pointcut `OrderManip` selects the execution of *setter*-methods that belong to the class `Order`. As the parameter, it exposes the `Order` object on which the method is executed.

3.1 Deployment Responsibilities

We will now define deployment code by outlining its typical responsibilities. These responsibilities are illustrated in Listing 3.3, which provides an example of deployment logic for the reusable

```
1  abstract aspect AbstractTrace {
2      PrintStream output;
3
4      void setOutput(PrintStream p) {
5          this.output = p;
6      }
7
8      abstract pointcut TracePoint(Object o);
9
10     before(Object o): TracePoint(o) {
11         output.println("Entering " + o.toString());
12     }
13 }
```

Listing 3.1: AspectJ implementation of a reusable tracing aspect

```
1  aspect ProgramPoints {
2      pointcut OrderManip(Order o): execution(* Order.set*(..)) && this(o);
3
4      ...
5  }
```

Listing 3.2: Independent specification of program pointcuts

```
1  aspect TraceDeploy issingleton() extends AbstractTrace {
2      pointcut TracePoint(Object o): ProgramPoints.OrderManip(o);
3
4      TraceDeploy() {
5          setOutput(Application.getLog());
6      }
7
8      declare precedence: TraceDeploy, ProfilingDeploy;
9  }
```

Listing 3.3: Elements of aspect deployment logic: 1. Concrete program points, 2. Instantiation, 3. Configuration, 4. Interaction resolution

`AbstractTrace` aspect. As explained in Section 2.2.1, deployment occurs in the ASPECTJ language by creating a concrete subaspect: concrete aspects are automatically instantiated and their advices are woven. The other elements of the aspect `TraceDeploy` are the following:

Concrete program points Foremost, the advices of a reusable aspect must be connected to program points in the concrete application where the aspect is deployed. This occurs by overriding named pointcut definitions in ASPECTJ (so the unit for these program points is a join point). Instead of defining a new pointcut expression, we employ the predefined pointcut `OrderManip` in Listing 3.3. The pointcut is used as such, without further modification. Note that the argument value of this pointcut is of a more specific type (`Order`) than what is expected by the abstract aspect (`Object`).

Instantiation Several instances of the same aspect may be required in the deployment context. In that case, the deployment code should specify an instantiation strategy. ASPECTJ offers a number of predefined strategies that are selected through an aspect modifier. In Listing 3.3, we use the **`issingleton`** strategy to create a single, global instance; this option is used by default, when no strategy is specified. Other strategies create one aspect instance per calling or receiving object associated with the join point (**`perthis`** and **`pertarget`**), or per execution of a join point (**`percflow`**).

Configuration Related to instantiation, the aspect instances may need to be configured with program parameters as well. Such parameters are typically stored as instance data for the aspect. In the case of our tracing behavior, the aspect needs the handle of a log file to write tracing messages to. The constructor of the aspect is used to configure this handle in Listing 3.3.

Similar to JAVA classes, ASPECTJ aspects always execute a constructor upon instantiation; because aspects are implicitly instantiated by the run-time environment, this is always the *nullary constructor*, *i.e.*, the constructor without arguments. This constructor can therefore be employed to place initialization code. Alternatively, code outside of the `TraceDeploy` aspect may access its sole aspect instance through the predefined static **`aspectOf`** method.

Interaction resolution Multiple aspects deployed in the same application may produce results that are unintended or undesirable. Most notably, when several aspects advise the same join point, their weaving order is not defined (this is analyzed by Douence et al., 2002, among others). In cases where the correct operation of the application depends on a particular order or combination of the advice code, deployment logic may specify some form of *interaction resolution* for aspects. For example, in Listing 3.3, we specify that tracing advice should have higher precedence than profiling advice (which is deployed by a similar `ProfilingDeploy` aspect), to avoid distortion of the performance measurements. Note that while before advices are executed in order of decreasing precedence (highest precedence first), after advices are run in the opposite order (lowest precedence first). As such, the precedence declaration will ensure that the before/after tracing advices are executed 'around' before/after profiling advices, which are in turn executed around the join point.

> In ASPECTJ, only ordering of advice code is supported: precedences may be declared between aspect types, while the declaration order is employed for advice methods of the same aspect. Other approaches offer more extensive means for the specification of an interaction resolution, for example, dependencies, exclusions, implicit activations or even arbitrary compositions between the advice behavior may be needed[1].

Not all of the above elements are necessarily specified in a deployment. In some cases the defaults may be sufficient, for example, when only a single aspect instance is needed and all the aspects have strictly orthogonal functionality. The important point, however, is that these properties normally depend on the concrete context where an aspect is deployed, and deployment code will therefore typically control them.

3.2 Requirements for Expressive Deployment

We now present a number of problems that we have identified in the deployment facilities of current aspect approaches. The problems concern the sharing of deployment logic between different aspects, the growth of the deployment specification when quantifying deployments, and the integration of deployment code with run-time events in the main program.

3.2.1 Reuse of Deployment Logic

As indicated by the previous discussion, the deployment entities for aspects may become a complex and substantial piece of code. When multiple aspects are deployed with equal or similar deployment logic, one may desire to share parts of their deployment code. However, deployment logic is treated as 'throwaway' code in current aspect approaches: deployments are intended to be used only once, and must be written from scratch for each new deployment.

Concretely, there is no sharing of deployment code between connectors in JASCO. In ASPECTJ, one can set-up an inheritance hierarchy to create multiple deployments of one reusable aspect; these deployments may share elements of their deployment code. However, the same mechanism cannot be used to share deployment elements between deployments of different aspects, due to the single inheritance restriction. For example, `TraceDeploy` cannot inherit from both the `AbstractTrace` aspect and an aspect `StandardDeploy` with deployment elements. `StandardDeploy` could inherit from `AbstractTrace` instead, but then it can only be used to deploy tracing behavior. Finally, XML configuration files do not predefine general inclusion or name reference mechanisms, and to our knowledge none of the aspect approaches discussed in Section 2.2.3 provide custom mechanisms to share parts of a deployment configuration.

When more complex deployments are taken into account, a mechanism for the sharing of deployment code between different aspects is required to avoid undesirable code duplication and ensure maintainability of the software application.

[1] The subject of possible interaction resolutions is treated more extensively in Section 4.1.3, including a discussion of literature. In the current context, we simply point out that the resolution specification is a non-trivial part of the deployment logic.

3.2.2 Deployment Quantification

Current deployment means offer essentially no control structures: the deployment code is executed only once, during weaving or initialization of the application. Consider an application with n reusable aspects (defined as abstract aspects such as `AbstractTrace`) and m program regions (defined by pointcuts such as the ones in `ProgramPoints`). To deploy an instance of each aspect for each of these program regions, regardless of the approach, one has to write $n \times m$ deployment entities, one for each combination of an aspect and a program region. For example, in AspectJ one has to define a concrete subaspect for each combination, and in JAsCo one has to write a hook instantiation line inside a connector for each combination.

Such a "specification complexity" of $n \times m$ is much larger than it needs to be, since an unambiguous specification may be given consisting of only $n + m$ pieces of information (n aspect names and m pointcut names), plus some constant information to specify the quantification. When the quantification cannot be expressed in the specification, the programmer needs to unwind the repetition manually and the specification may grow to an unwieldy large size for larger values of n or m.

One might think that this limitation can be worked around by defining a new pointcut that selects the union of the regions selected by the m pointcuts (*i.e.*, that is the disjunction of the pointcut predicates and thus a specification of size m), followed by a deployment of each aspect with this new pointcut (these are n specifications of constant size). While this avoids the growth of the specification, it does not achieve the same effect, since one instance of each aspect is used for all program regions. The aspect uses one set of data for all regions, cannot be (dis)activated per region separately, *etc.*

Proper means of deployment quantification are needed to further avoid undesirable code duplication. They are required to avoid scalability problems that may — at least theoretically — occur when a large number of aspects are deployed in an application.

3.2.3 Dynamic and Integrated Deployment

Most approaches only support the static deployment of aspects. In the AspectJ language for example, the application of a reusable aspect requires the introduction of a static entity such as the `TraceDeploy` aspect, even if an existing aspect and pointcut are being reused without modification. This design choice is related to the implementation technology: if the advice application can be determined at the time of weaving, the performance overhead of a conditional dispatch at every candidate join point can be avoided. AspectJ supports only compile-time or load-time weaving of advice, and can produce faster code if the pair of pointcut and advice are known statically.

Dynamic deployment JAsCo connectors are also static entities, but can be enabled and disabled at run-time through predefined API methods, by polling a deployment directory for changes, or through an administrative console called the *introspect* tool. This *dynamic deployment* is realized by preparing all possible join points with a dispatch mechanism or, more recently, by employing a run-time weaver to avoid the performance overhead (De Fraine et al., 2005b). Alternatively, aspect-aware virtual machines (Bockisch et al., 2004; Bonér et al., 2005)

have been proposed for this purpose. JBOSS/AOP offers a dynamic deployment mechanism very similar to the one of JASCO, called *hot deployment*.

Integrated deployment Although the above provisions for dynamic deployment remedy some problems, the dynamic deployment is still separated from the main application, and intended to be used by the application administrator rather than the developer. However, it is easy to imagine situations where it is useful and desirable to deploy new aspect instances upon run-time events. For example, one might want to deploy a new instance of the tracing aspect when a user logs into the system (and subsequently remove this instance from the system on logout). Since the developer has no explicit control over the instantiation of aspects in most approaches, such *integrated deployment* is only supported through predefined aspect instantiation strategies such as **`perthis`**, **`pertarget`** and **`percflow`**.

However, these predefined strategies cannot handle any but the most simple variations. For example, only if the login event involves the creation of a session object, one is able deploy one aspect instance for each session (by associating it to a session). If there do not happen to be such session objects in the organization of the application, another instantiation strategy would be needed or, if this is not supported, the design of the application would need to be altered. Also, if it were required to create two aspect instances at this event, yet another strategy would be needed to avoid writing two deployment aspects and duplicating the effort.

Information transfer A further problem is that the aspect instances often need to receive configuration parameters from the main application and/or that results from the aspect instance need to be transferred back to the main application. In case of our example tracing aspect, the aspect instance needs to be configured with a log handle. Since the deployment is separated from the main application, such information transfer requires a form of static lookup: either the application makes the desired objects available through static fields or methods (as is done with the `getLog` method in Listing 3.3), or the aspect instances are retrieved by the base application through the static **`aspectOf`** construct, after which the base application *injects* (or retrieves) the information in the aspects.

These solutions impose a restriction as the information must be addressable with a unique key. In case of more complex instantiations, this becomes increasingly difficult to employ: suppose two aspect instances could be implicitly created after a login, by what key would one address each of them to pass the relevant log output? A workaround for this problem could be that the aspect implements additional advice methods to capture (or expose) the required information. This is not straightforward and error-prone: *e.g.*, in case of a required parameter, one has to verify that in all cases this capture has succeeded before the main advice of the aspect is triggered.

In summary, we identify shortcomings with the static nature of deployment constructs in current approaches. Useful cases require deployments that are not only dynamic, but which can be integrated with program events of the main application. This necessitates the flexible creation of aspect instances at these events, and the configuration of the aspect instances with program parameters. Support for these cases is required to allow developers to reuse existing aspects on more occasions.

Requirement	Solution
Reuse of deployment logic	Shared deployment procedures parameterized with abstract aspects, advice, pointcuts, resolution strategies, *etc.*
Deployment quantification	Control structures for deployment code, nested invocations of deployment procedures
Dynamic/integrated deployment	Deployment procedure invocation from base program, or attached to base program events, with a possible transfer of program values

Table 3.1: Organizing deployment procedures to allow expressive deployment

3.3 First-Class Deployment Procedures

To meet the requirements from the previous sections, we propose to organize deployment code as procedures that borrow a number of properties from ordinary code, and that could therefore be called *first-class deployment procedures*. Concretely, we argue that it should be possible to parameterize deployment procedures with aspect entities such as pointcuts and advice, that control structures should be available in deployment procedures, and that it should be possible to invoke deployment procedures from the base program (or otherwise connect them to program events). Below, we discuss these properties in relation to the identified requirements; the discussion is summarized in Table 3.1.

Reuse of deployment logic The ability to parametrize deployment procedures with aspect entities such as pointcuts and advice enables to build abstractions that allow the reuse of the deployment logic. For example, when it is required to deploy both tracing and profiling advice in the same manner, we can define this common logic as a shared deployment procedure that receives the advice to deploy as a parameter. As such, the specification can be shared between the deployment of multiple concerns and implementation details of this shared behavior can be hidden from its clients.

Additionally, other kinds of parameters may allow to build more variation points into deployment procedures, such that similar but different deployments can share a common specification. For example, a boolean parameter may be used to (dis)activate an optional part of the deployment specification, or certain parts of the deployment may be left abstract such that the client can fill in specific functionality by providing a function or object parameter.

Deployment quantification The ability to use control structures in the deployment code allows to quantify deployments without the effects of growth that occur when all deployments need to be manually enumerated. Control structures such as loops allow for a very direct expression of the quantification: in case of the example from Section 3.2.2, one can specify a loop that iterates over the m pointcuts that specify the main program regions and the n reusable aspects, and deploys an aspect instance for each of the $n \times m$ combinations.

Additionally, we observe that the mechanism of parameterization and invocation of deployment procedures already serves as a simple control structure. Since a deployment procedure may create multiple aspect instances per invocation, one can define a deployment procedure that receives an aspect as a parameter and that deploys m instances of this aspect, one for each of the m pointcuts (this amounts to m statements). The deployment procedure may then be invoked with each of the n reusable aspects that needs to be deployed (using n invocation statements). In this way, we also achieve the desired creation of $n \times m$ aspect instances with a specification that has size $n + m$, without resorting to explicit control structures. Nevertheless, we feel that recognizability of control structures may still provide an additional level of convenience for the developer, despite this redundancy.

Dynamic and integrated deployment Through a linking of the deployment procedures to points in the execution of the main program, aspects can be deployed with a tight integration to the run-time events of this program. The effect of the link is that the deploy procedure is invoked each time the execution of the application reaches one of these points, which makes it easier to accommodate for different variations in the instantiation of aspects, since aspect instances can now be created explicitly by the developer, at the relevant events. This also greatly facilitates the configuration of the aspect instances, because the relevant parameters for the transfer of information to the base program are often readily available at these points (e.g. a reference to the user and his log file is normally available in the code section that handles a login event). For example, if we consider the requirement to deploy a new aspect instance at each login that writes to the user's personal log file, then the implementation becomes trivial when one can attach the instantiation logic to that point in the base application where a user's login is processed. The occurrence of this event is the appropriate time to instantiate the aspect, and the parameters to configure it (e.g. the session and the user) are readily available.

More concretely, we envisage two levels at which deployment logic can be integrated with program events in the base application. The first, straightforward solution is to include invocations of deployment logic directly in the code of the main application. This may seem to imply that the base program is no longer *oblivious* to the presence of the aspects[2]. Note however that the program points where the advice behavior is applied can still be unaware of the aspects — the implicit invocation of advice methods is retained. Nevertheless, if the deployment is required in different parts of the application, this has the possible disadvantage of scattering the deployment logic and tangling it with the implementation of the main application. At that point, the deployment logic of the aspects has itself become a crosscutting concern, and its modularization could be achieved by specifying it as another aspect. The second solution is therefore to allow the invocation of deployment procedures from advice code inside of such deployment aspects.

[2] Recall from page 10 that such obliviousness is considered a discriminating feature of AOP by Filman and Friedman (2000). This is a source of much debate in the AOSD community, for example in relation to the use of annotations to indicate program points where advice behavior needs to be applied.

3.4 Discussion: Intensive Usage of Aspects

This chapter discusses a number of problems in the deployment facilities of current aspect-oriented approaches. These issues are not minor problems that 'slipped through the net' of the unwary designers of one particular approach or the other, and that can be straightforwardly patched or added by cherry-picking and combining from the different approaches available today. The issues are more fundamental in the sense that all of the current approaches appear to have been designed with a mindset of aspect-oriented technology where only a handful of aspects are deployed inside of one application (*e.g.*, one aspect for security, one aspect for profiling,...). In a way, most or all of the discussed problems relate to the fact that current deployment mechanisms do not scale up to a higher number of aspects or aspect instances.

This work paves the way for applications where aspects may be used much more intensively. Similar to the number of objects and classes in object-oriented applications today, we envision applications with hundreds or thousands of aspect instances. To enable applications with such a large number of aspect instances it should be as easy and flexible to deploy an aspect as it is to create an object instance, spawn a thread, *etc.* It should therefore not come as a surprise that the solution proposed in this chapter — first-class deployment procedures — makes a number of features from ordinary code available to deployment code.

Chapter 4

The EcoSys AOP Framework

In the previous chapter, we have made the case of increasing the expressiveness of aspect deployment code by adding a number of rich mechanisms that are commonly available for general program code (such as parameters, control structures, run-time execution,...) and by providing a better integration with the code and program events in the base program. Deployment entities with these properties were called first-class deployment procedures. In this chapter, we take this idea one step further and propose an AOP approach named ECOSYS which does not organize any distinction between deployment code and ordinary code. ECOSYS provides an AOP framework where the aspect behavior — including the aspect deployment — is expressed in a standard object-oriented programming language, using a number of predefined classes. This object-oriented programming language is called the *host language*.

Recall from the discussion of existing pointcut/advice approaches in Chapter 2 that most aspect languages already express the advice behavior in the host programming language. For example, ASPECTJ advice method bodies are ordinary JAVA code blocks[1]. Other elements of the aspect specification, such as the pointcut expressions and the deployment information, are specified in a dedicated language. Current AOP frameworks, as discussed in Section 2.2.3, enable the compilation of the advice methods using a base language compiler, but otherwise the situation remains unchanged: the pointcuts and the aspect deployment is still expressed using a dedicated formalism, although this specification is placed in XML configuration files or expressed using annotations. ECOSYS differs fundamentally from these approaches in the sense that at least the deployment specification is also specified in the host language.

The presentation of ECOSYS proceeds as follows. Section 4.1 introduces the classes and interfaces that constitute the ECOSYS programming model and Section 4.2 demonstrates how deployment logic specified in the host language using these types provides the features of first-class deployment functions. The ECOSYS programming model is defined independent of a particular implementation technology. Section 4.3 discusses a number of common implementation platforms for aspect-oriented approaches and evaluates how they may be employed for an ECOSYS implementation. A prototype implementation that employs the ASPECTJ implementation to instrument the bytecode of possible join points is then presented.

[1] With the exception of the reserved identifier **proceed** in case of around advice. This is discussed extensively in Section 9.1.1 and Section 9.2.1.

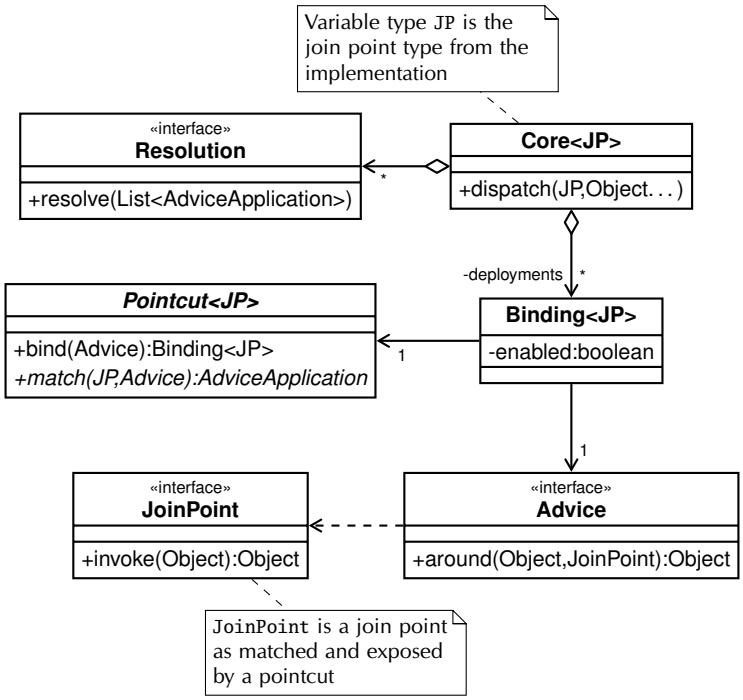


Figure 4.1: UML class diagram of the central classes of the ECOSYS programming interface

4.1 Programming Interface

At its core, the ECOSYS framework consists of a number of interfaces and classes that function as a programming interface that can be used to implement crosscutting behavior modularly as one or more aspects. An overview of this interface is presented in Figure 4.1. The structure of the aspect behavior corresponds to what is known from existing aspect approaches: (i) advice methods specify preceding, succeeding or replacing for intercepted join points, (ii) advice is bound to pointcuts, which select join points and expose context arguments, (iii) bindings of pointcut and advice pairs may be deployed in the system, together with resolutions that handle combinations of multiple advice applications. The elements of this model are discussed in more detail over the course of the following subsections.

4.1.1 Join Point and Advice

Similar to other AOP frameworks such as JBOSS/AOP, the developer specifies advice behavior by constructing a class that implements the predefined `Advice` interface. The interface prescribes a very general advice method. It is executed in place of the intercepted join point and receives the context arguments from the join point, as well as a closure representation of the intercepted join point as an object of type `JoinPoint`. This closure object may be used to invoke the

```
1  package ecosys;
2
3  public class BeforeAfterAdvice implements Advice {
4      public void before(Object arg) {}
5      public void after(Object arg) {}
6
7      public final Object around(Object arg, JoinPoint proceed) {
8          before(arg);
9          try { return proceed.invoke(arg); }
10         finally { after(arg); }
11     }
12 }
```

Listing 4.1: A predefined template advice method which includes configurable behavior before and after the execution of the intercepted join point

intercepted join point at will (*i.e.*, zero or more times, with the original or different arguments). The only constraint is that the advice method should produce a result (or throw an exception) to return to the join point client. An advice method corresponds to around advice in ASPECTJ, with the difference that the join point closure is not an explicit argument in that language, but a special method available through the reserved `proceed` identifier.

The binding of context arguments is the responsibility of the pointcut entity. There is therefore no direct relation between the arguments of the `invoke` method in the `JoinPoint` type and *e.g.*, the actual method arguments in case the join point is some method invocation. Different pointcuts may bind arguments differently, and one pointcut may even have different bindings per join point. For example, a pointcut may expose the receiver of the invocation for some of the join points, while it exposes the first argument for other join points, and an unrelated object for yet other join points. The arguments of the `invoke` method in type `JoinPoint`, as well as all but the last argument of the `around` method in type `Advice` always refer to the join point arguments exposed by the pointcut. Join point arguments not bound by the pointcut are simply "tunneled" from the join point client to any execution of the join point. Note that the interfaces in Figure 4.1 only display versions of the interfaces for the case of one exposed argument. Other cases, such as zero or two exposed arguments, are similar and have been omitted from the presentation for reasons of simplicity.

Although the general `Advice` type is very powerful, it is not very convenient to specify simple advice methods that only add preceding or succeeding behavior to the join point. ECOSYS therefore provides a predefined `BeforeAfterAdvice` class, presented in Listing 4.1, which implements the advice method as a template method which invokes the join point with the original argument and returns it result, and additionally invokes the (currently empty) `before` and `after` methods (this is an instance of the *template method design pattern* described by Gamma et al., 1995, pp. 325–330). Subclasses of this class may override these two methods to specify advice behavior similar to before and after advice methods in ASPECTJ. Since the `around` method has been declared final, it is guaranteed that any instance of `BeforeAfterAdvice`

```
1  class CollaboratingAspect {
2      int data;
3      class Capture extends BeforeAfterAdvice {
4          void before(Object o) {
5              CollaboratingAspect.this.data = ...;
6          }
7      }
8      class Use implements Advice {
9          Object around(Object arg, JoinPoint proceed) {
10             return CollaboratingAspect.this.data;
11         }
12     }
13 }
```

Listing 4.2: Example of an aspect module with different advice methods that operate on the same set of data

follows this template and only adds behavior to the intercepted join point. This guarantee may be exploited by an EcoSys implementation.

Since the `Advice` interface defines only a single advice method, it may seem that a class can define only one advice method, and instance data can therefore not be shared between different advice methods. However, in Java, collaborating classes may be defined as *inner classes* in an enclosing class. As explained by Gosling et al. (2005, §8.1.3), an instance of an inner class is implicitly associated with an instance of the immediately enclosing class. Instances of the inner classes that have the same enclosing class instance may share data in this instance. The mechanism of inner classes may be used to define aspect modules where, as explained in Section 2.2, different advice methods maintain one set of instance variables between invocations of the advice methods. An example of such a pair of collaborating advice methods is shown in Listing 4.2. This mechanism is very similar to the organization of aspect modules in JAsCo: different hooks in the same aspect bean may also share instance data through the enclosing aspect bean instance.

4.1.2 Pointcut and Binding

While EcoSys employs a standard object-oriented programming language for the specification of advice methods and aspect deployment code, this is not necessarily the case for pointcut expressions.

The choice of pointcut language is an important design decision for aspect languages, and a wealth of pointcut approaches have been proposed in literature. While AspectJ employ logic combinations of a fixed number of primitive pointcut designators that match individual program points according to type and identifiers patterns, Douence et al. (2001), Walker and Viggers (2004), Vanderperren et al. (2005a) and Allan et al. (2005) propose event patterns over traces of the program execution as a more powerful model. Gybels and Brichau (2003), Rho and Kniesel (2004) and Ostermann et al. (2005) employ yet another formalism, and select relevant

program points through logic queries over structural or behavioral facts of the program. The cited approaches also frequently make additional program properties available as selection criteria. In addition, approaches by Nishizawa et al. (2004), Benavides Navarro et al. (2006), Harbulot and Gurd (2006) and Tanter (2008) have independently extended the expressiveness of pointcut languages with new properties, while Kiczales and Mezini (2005), Aldrich (2005), Griswold et al. (2006), Kellens et al. (2006) and Sakurai and Masuhara (2008) propose measures to make pointcut expressions more robust with respect to the evolution of the base program.

This is clearly a very active research domain, and while the targeted concerns are very relevant, we note that they seem largely orthogonal to the problems we describe in Chapter 3 regarding the expressiveness of aspect deployments. Moreover, the underlying implementation of an aspect approach generally determines the program properties that can be matched efficiently. For these reasons, ECOSYS is designed independent of a particular pointcut approach. In Figure 4.1, the `Pointcut` type, and the types `Binding` and `Core` that depend on this type, have been parameterized with a type parameter `JP` which represents the type of a primitive join point exposed by the underlying implementation. The `match` method of class `Pointcut` is employed to check whether a pointcut matches such a join point. If there is indeed such a match, then the method will return an `AdviceApplication` which is derived from the `Advice` given as the second method argument. This application takes into account the bindings of context arguments by the pointcut: it provides a translation between a raw join point where all context is available and a high-level join point as provided to the advice (type `JoinPoint`). We elaborate on the precise definition of `AdviceApplication` in the following section.

Based on the underlying implementation, a concrete pointcut language may be offered. We identify at least two principal ways in which this language can be integrated with the framework:

1. The developer writes the pointcut in a concrete syntax that is separated from the host language. For example, the pointcut expressions could be written in annotations or XML files that are transformed to a subclass of `Pointcut` by a preprocessor, or the pointcut expression could be provided as a string to some interpreter that returns a `Pointcut` object. This approach has the advantage that a concrete syntax is typically very convenient.

2. Abstract syntax tree nodes are provided as classes in the host language, and the developer writes code that instantiates these classes and combines instances to construct `Pointcut` objects. While it may be expected to be tedious to write the pointcut expressions directly in the abstract syntax, the host language typically offers very rich abstraction mechanisms and control-flow structures. These can be used to build pointcut libraries that offer more powerful pointcuts that are reusable and high-level since the complexity can be hidden in the library implementation. Ostermann et al. (2005, Sec. 3.2) have also noted such advantages when employing a language with rich abstraction mechanisms (in their case PROLOG) to specify pointcut expressions.

Finally, the `Pointcut` class also offers a `bind` method to construct a binding to an `Advice` instance. (Both pointcuts and advice instances may be bound any number of times.) A pointcut/advice binding is represented by an instance of the `Binding` class, which is returned by this method. This is the principal unit of deployment; a binding may be enabled or disabled individually.

```
1  class Core<JP extends RawJoinPoint> {
2    Collection<Binding<JP>> deployments;
3    Collection<Resolution> resolutions;
4
5    Object dispatch(JP jp, Object thiz, Object target, Object... args) {
6      List<AdviceApplication> advs = new LinkedList<AdviceApplication>();
7      for(Binding<JP> b: deployments)
8          if(b.isEnabled()) {
9              AdviceApplication aa = b.pc.match(jp,b.adv);
10             if(aa != null)
11                 advs.add(aa);
12         }
13
14     for(Resolution r: resolutions)
15         r.resolve(advs);
16
17     RawJoinPoint rjp = jp;
18     ListIterator<AdviceApplication> it = advs.listIterator(advs.size());
19     while(it.hasPrevious())
20         rjp = it.previous().apply(rjp);
21
22     return rjp.invoke(thiz,target,args);
23   }
24 }
```

Listing 4.3: ECOSYS advice dispatch procedure

4.1.3 Join Point Dispatch and Interaction Resolution

As shown in Figure 4.1, the `Core` class acts as a central registry where pointcut/advice bindings may be deployed. In addition, *resolutions* that manipulate advice combinations may be registered. The entities in the central registry are consulted during a dispatch procedure that is triggered when the implementation encounters a new join point. A relevant portion of the `Core` class related to the dispatch procedure is shown in Listing 4.3. Figure 4.2 presents a class diagram with the central classes related to advice application and resolution, as well as a class hierarchy of predefined resolutions. We describe these elements in the remainder of this section.

Join Point Dispatch The `dispatch` method (lines 5–23) is invoked with a primitive join point representation from the underlying implementation (type `JP`), together with the arguments from the join point context: the caller object (typically called `this` but since this name is reserved in many object-oriented languages, we employ `thiz`), the receiver object (called `target`) and the remaining arguments (called `args`). Depending on the kind of join point, some of these parameters will not be provided. For example, the invocation of a static method will not

have a receiver object and a field get operation will not involve any arguments. On line 1, the `JP` type parameter is declared a subtype of the type `RawJoinPoint`, which is shown in Figure 4.2. While the join point type from the underlying implementation is unknown, this bound ensures that the join point supports an `invoke` method to execute the join point with provided `thiz`, `target` and `args` arguments.

The selection of advice is handled in lines 6–12 and the application in lines 17–20 (the resolution occurs between these two steps and is discussed below). Advice is selected by iterating over all the deployed bindings and collecting the advice applications that are returned by the pointcuts of enabled bindings to indicate a match (**`null`** is returned to indicate that there is no match). The list of collected `AdviceApplication` instances are applied by 'wrapping' the original join point by each of the advice applications in reverse order (a `ListIterator` is employed to retrieve the list elements from last to first). As a result, when the resulting `RawJoinPoint` is invoked with the arguments from the join point context on line 22, the advice instances in the advice list will be considered first to last, ending with the original join point. This is the typical *chain of around advice* in AOP approaches.

In this procedure, we detail that the `apply` method of an `AdviceApplication` constructs a new `RawJoinPoint` that, when invoked, executes the advice with the relevant arguments routed from the `RawJoinPoint` invocation. Similarly, the `JoinPoint` argument of the advice is bound to a closure that will invoke the `RawJoinPoint` instance that was given as an argument to the `apply` method by appropriately routing the new arguments coming from the advice (or tunneling the arguments that are not considered by the advice). In brief, an advice application does not only provide an `Advice` instance but also routes back and forth between a `RawJoinPoint` interface and the high-level `JoinPoint` that is employed by the advice.

Interaction Resolution As is also explained in Section 3.1, it is often necessary to control the advice combinations that are employed when multiple aspects are deployed in one application, due to the occurrence of undesirable aspect interactions. For interactions where the advice methods are applied to a common join point, ECOSYS provides a simple but powerful dynamic resolution model that was directly inspired by the *combination strategies* proposed for the JASCO approach by Suvée et al. (2003, Sec. 2.3). The `Core` class also registers instances of type `Resolution`, and between the selection and the application of advice, each registered resolution is given an opportunity to manipulate the advice list through the invocation of its `resolve` method (lines 14–15 in Listing 4.3). Note that the model does not presumptively restrict the expressiveness of resolution with stipulations regarding the kind of manipulations that may be applied: the position of advice applications may be changed, new advice applications may be inserted or existing ones removed. (We will consider applications of all of these cases in the following paragraph.) However, one can image that in quite a number of situations, restrictions may be added to avoid excessive run-time overhead[2], or to preserve modular reasoning about the aspect behavior (currently any resolution may control the advice for any join point).

[2]The current resolution model does indeed give rise to worries regarding the execution efficiency. For example, the registered resolutions are consulted for every join point, even when it is not matched by a pointcut. At present, this was not considered, but in the long term the flexibility of resolutions should be *optimized* rather than *maximized*. A number of possible optimizations are readily apparent: (i) if resolutions do not introduce advice for join points unmatched by pointcuts, they need to be consulted for a much smaller number of join points, (ii) if resolutions only involve specific advice instances, this number may be further reduced, (iii) if resolutions always provide the same results for the same

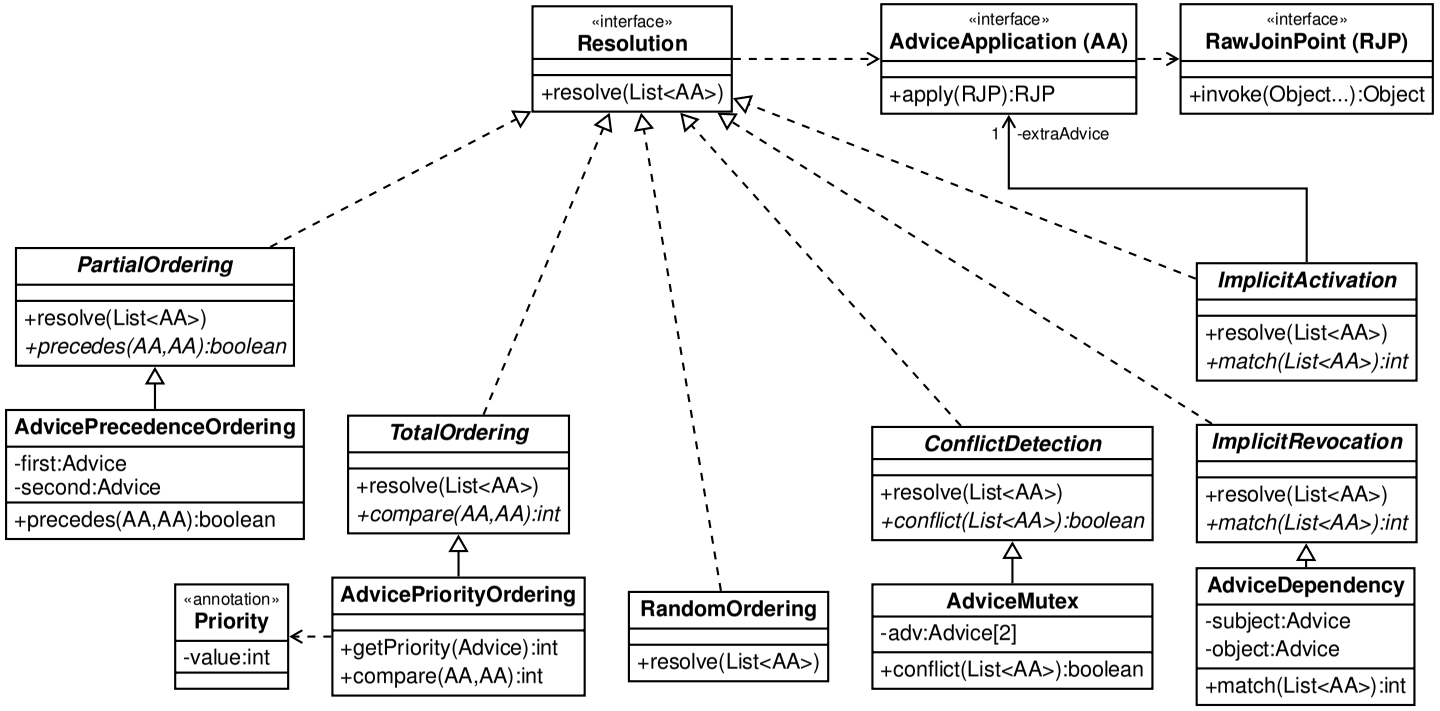


Figure 4.2: Condensed UML class diagram of the hierarchy of predefined EcoSys resolutions and types involved in advice application. The abbreviations `AA` and `RJP` are used for the type names `AdviceApplication` and `RawJoinPoint` respectively.

While JAsCo provides combination strategies in addition to an independent advice ordering mechanism, EcoSys integrates ordering with the aforementioned resolutions, to provide one unified resolution mechanism. In addition, support for several other resolutions described in literature by Brichau et al. (2002, Sec. 5), Douence et al. (2002, Sec. 4) and Tanter and Noyé (2005, Sec. 5) is provided. The support for all of these common interaction resolutions is organized in EcoSys as a class hierarchy of predefined `Resolution` types. This class hierarchy is presented in Figure 4.2; we organize the presentation of its members in three categories:

1. The *permuting resolutions* only reorder the elements of the advice list; they do not add or remove advice applications. The implementation of these resolutions is structured according to the mathematical kind of order they apply.

   (a) Type `TotalOrdering` employs a *total order* specified by the method `compare`. When this method is used to test the relationship between two `AdviceApplication` instances, then either the first should come before the second (indicated by a negative result), the second should come before the first (indicated by a positive result) or the relative position of the two should be retained (indicated by the result zero). Since there is always one of these relationships between any two elements, the relation is total. In addition, it is required that the relation is transitive and antisymmetric[3]. The `resolve` method will effectuate the total order through a classical sorting of the list of advice applications.

   The class `AdvicePriorityResolution` implements a concrete case of such a total order by comparing advice applications according to the numerical priority that is assigned to the involved advice. In this case, advice priorities may be assigned by including the `@Priority` annotation in the definition of the advice class. Advice instances of classes without this annotation are assigned a fixed priority value of zero.

   (b) Type `PartialOrdering` employs a *partial order* specified by the method `precedes`. Unsurprisingly, this method returns a **`true`** result to indicate that the first argument should precede the second argument. However, the result **`false`** indicates any of two possible situations: either the second argument should precede the first, or there is no relationship between the two arguments. (These two cases may be distinguished by invoking the method again with the arguments in reverse order.) When representing the relationships from a partial order as directed edges in a graph the resulting graph should be acyclic, or put differently, the transitive closure of the "precedes" relation should be antisymmetric. The `resolve` method will effectuate the partial order by carrying out a *topological sorting*[4] of the list of advice applications. This procedure finds a total order of the elements that is compatible with the specified partial order (where possible the original order is retained).

   The class `AdvicePrecedenceResolution` may be used to enforce that one advice instance should always precede another when they appear together in the advice ap-

inputs, their results may be cached.

[3] Where antisymmetry means that `compare(x,y)<0` implies `compare(y,x)>0` and vice versa, and `compare(x,y)=0` implies `compare(y,x)=0`. Note that referential equality nor the `equals` relationship are involved in this definition.

[4] One of the better known applications of topological sorting is the linear scheduling of a number of tasks with dependencies, as done by build tools such as Make.

plication list. This resolution is configured with two advice instances; its `precedes` method will indicate corresponding precedence for advice applications that involve these advice instances. The use of this class is similar to a precedence declaration in ASPECTJ, except that it operators at the level of advice instances instead of aspects. ECOSYS also provides a resolution class that may be used to configure a precedence based on advice types rather than specific advice instances, but this has been omitted from Figure 4.2 for reasons of clarity.

(c) The `RandomOrdering` resolution will simply "shuffle" the advice application list. It rearranges the elements into an undetermined order at each invocation. This may seem a toy resolution, but since it causes many different advice orderings to be tried, it may be used to detect dependencies between aspects. This could be considered an instance of *random testing* as described by Hamlet (1994).

2. The *verifying resolutions* do not alter the advice application list; they detect invalid advice combinations at run-time and abort the join point execution in case of a violation. The `resolve` method of the type `ConflictDetection` will raise an exception when its `conflict` method indicates that the advice list contains a conflicting combination. This behavior is used for the implementation of a mutual exclusion between a pair of advices in the type `AdviceMutex`. This type indicates a conflict when an advice application list contains advice applications for both of the two configured advice instances.

3. The *intervening resolutions* alter the advice application list by adding or removing advice applications. They can be considered extensions of the pointcut expression that can take the presence of other advice behavior into account. We organize the implementation of these resolutions according to the kind of modification:

   (a) The `ImplicitRevocation` resolution will remove an advice application from the list when its elements meet a certain condition. The `match` method verifies the condition and indicates the position of the advice application that should be removed. In case there is no match, a negative value is returned. The `AdviceDependency` resolution employs this behavior to realize that the application of one advice instance (the object) is dependent on the presence of another advice instance (the subject). The `match` method of this type will indicate the position of an advice application involving the object in the case of advice application list that do not have elements involving the subject as well. Note that this resolution will not activate the object advice if it was not originally selected; this dependency is called a *twin combination* by Suvée et al. (2003, Sec. 2.3).

   (b) The `ImplicitActivation` resolution will insert additional advice in the advice application list when its elements meet a certain condition. Tanter and Noyé (2005, Sec. 5) refer to this kind of advice activation as an *implicit cut*. The `match` method verifies the condition and indicates the position where the new advice is to be inserted. Note that the additional advice behavior is specified as an instance of `AdviceApplication` rather than `Advice`. Indeed, to apply the advice behavior it is required to organize a binding of context parameters from the join point. As we explained in the first part of this section, an instance of `AdviceApplication` extends an advice with this function.

4.2 Demonstrations of First-Class Deployment Procedures

Since the programming interface of EcoSys exposes all of the deployment primitives as standard programming language elements, the deployment code in EcoSys is written as an ordinary program. This deployment code includes all of the responsibilities discussed in Section 3.1. For example, the following snippet deploys a hypothetical tracing advice using an existing EcoSys core:

```
TracingAdvice adv = new TracingAdvice(Application.getLog());
adv.setOutputLevel(Logging.INFO);
core.deploy(modelManip.bind(adv));
core.addResolution(new AdvicePrecedenceResolution(authorization,adv));
```

The deployment code includes the instantiation of the advice, configuration of the advice instance with program parameters and pointcut, and the specification of interaction resolutions. The configuration of the instance employs an application log handle and the informative logging level. For the binding with the pointcut, an existing `modelManip` pointcut instance is employed. Finally, the resolution configures that an existing `authorization` advice instance precedes the application of the tracing advice (such that join points for which the access is denied are not included in the trace).

Since all the deployment responsibilities are specified using a language that offers the expressive abstraction, quantification and integration mechanisms that are discussed in Section 3.3, EcoSys realizes the concept of first-class deployment procedures from this section. To illustrate this point, we revisit the issues described in Section 3.2 and demonstrate that the required expressiveness is available in EcoSys.

Reuse of deployment logic Deployment logic may be shared between the deployment of multiple aspects by constructing methods that are parameterized with pointcut, advice or resolution entities. When multiple advice instances need to be deployed in a similar manner as the tracing advice from above (for example, because all of these advice instances relate to the registration of quality of service properties), then the commonalities of the deployment code may be placed in a separate deployment method:

```
void deployQoS(OutputAdviceFactory factory) {
    OutputAdvice adv = factory.createAdvice(Application.getLog());
    adv.setOutputLevel(Logging.INFO);
    core.deploy(modelManip.bind(adv));
    core.addResolution(new AdvicePrecedenceResolution(authorization,adv));
}
```

For obvious reasons, this method is formulated in terms of the most general types that offer the employed operations, in this case the `OutputAdvice` type. Since the constructor of the advice class is not a first-class value, it cannot directly be employed as the parameter for this method. The common solution in this case is to create some factory class which makes the constructor available through one of its methods. Instances of this class may be used as the

parameter to the deployment method. (An alternative approach is to use the reflective access of the programming language to obtain some `Class` object: this is also a first-class value which may be employed as a factory to create new instances of the advice class.) This deployment method may now be used for the deployment for multiple advice types (provided that factories for these advice types are available), for example:

```
deployQoS(profilingFactory);
deployQoS(tracingFactory);
deployQoS(invariantCheckFactory);
...
```

Besides sharing the deployment code, this deployment method also abstracts the implementation details involved in the deployment of quality of service advice behavior.

Deployment quantification The available control structures make it very straightforward to express repetitions of deployment behavior. For example, we may deploy one advice instance for each combination of a pointcut and advice from a list of pointcuts and a list of advice factories by employing a nested loop:

```
for(AdviceFactory factory: new AdviceFactory[] {tracingFactory,...})
    for(Pointcut pc: new Pointcut[] {modelQueries,modelManips,viewOps,...})
        core.deploy(pc.bind(factory.createAdvice()));
```

An important characteristic from this piece of deployment code is that a large number of advice instances may be created using a comparatively small specification. In this example, the addition of one advice factory will trigger the creation of many advice instances, one for each pointcut from the pointcut list.

Dynamic and integrated deployment Since the deployment logic is expressed using standard code in the host programming language, it is trivial to integrate deployment code with program events that occur in the base program, simply by placing the deployment code at the program points that correspond to the relevant events. The following code demonstrates the deployment of a new advice instance at a login event. Conveniently, the user and session instance are directly available at this point and can be employed for the configuration of the pointcut and advice behavior at this point:

```
void login(User u) {
    Session s = new Session(u);
    // Deploy tracing for this session:
    core.deploy(sessionOps(s).bind(new TracingAdvice(u.getLog())));
    ...
}
```

Although the direct inclusion of the deployment code is a very straightforward approach, this may cause scattering and/or tangling of the deployment logic if deployment is required at

several events, as explained at the end of section Section 3.3. However, the solution from that section is also applicable here: since the deployment code may be moved to an advice method, it is possible to employ a deployment aspect that will include the deployment of the original aspect(s) at the relevant program points. Since the deployment is crosscutting, it is appropriate to modularize its implementation using an aspect.

4.3 Developing an EcoSys Implementation

ECOSYS is designed independent of a particular implementation technology. In this section, we discuss a prototype implementation of ECOSYS to prove its concepts and to provide a guideline for possible future ECOSYS implementations. We first analyze the possible implementation platform choices and then present the prototype that we have developed.

4.3.1 Choice of Implementation Platform

The ECOSYS programming model includes entirely dynamic aspect deployment facilities. Deployment code may register or remove advice behavior for any of the pointcuts at any point in the execution. Additionally, the powerful model of interaction resolution may also manipulate the advice behavior at any join point. These features need to be taken into account when selecting an implementation platform for ECOSYS.

Aspect-aware Virtual Machines As explained, in Section 3.2.3, the use of aspect-aware execution models has been proposed to optimally support dynamic deployment of aspects. Since the advice application cannot be statically determined in case of dynamic deployment, the bytecode of the application cannot include direct advice invocations. While techniques such as the HOTSWAP API (part of the JVM tool interface) allow the changing of bytecode at run-time, Bockisch et al. (2006) and Bonér et al. (2005) argue that this approach has several disadvantages that direct support for the aspects in the virtual machine can avoid. Additionally, aspect-aware execution models may offer a more direct API for applying advice behavior to join points than the instrumentation of the static join point shadow, since join points are *run-time* program points. Aspect-aware virtual machines therefore appear a promising option for the implementation of ECOSYS.

Unfortunately, comparatively few approaches employ this kind of implementation platform and the choice of aspect-aware virtual machines is limited. The existing approaches have not been declared stable by their authors or do not receive continuous maintenance, making it still a high-risk endeavor to build implementations on top of them. STEAMLOOM and JROCKIT are two concrete aspect-aware virtual machines. STEAMLOOM is an extension of the JIKES research virtual machine proposed as the first virtual machine implementation to natively support dynamic aspects by Bockisch, Haupt, Mezini, and Ostermann. Version 0.5 did not support around advice which is highly problematic for our purposes; this was cited as future work by Bockisch et al. (2004, Sec. 3.2) and later added by Haupt (2005, Sec. 3.6.4). JROCKIT is a proprietary JAVA virtual machine from BEA Systems. An extension providing AOP support was developed by Bonér, Vasseur, and Dahlstedt (2005) and released as a private beta. However, its current status is unclear.

An investigation of the programming API of these approaches reveals that their programming model is directly targeting the end developer of aspects instead of the implementor of an aspect language back-end. For example, the example programs that demonstrate the use of the respective APIs are direct implementations of aspect behavior. This focus on a concrete user-level aspect model proves problematic to realize at least one of the more uncommon features of the ECOSYS model. From the description of its join point dispatch model in Section 4.1.3, it is clear that control of the list of advice applications is needed. While the programming interface of the discussed virtual machine approaches allow to deploy pairs of pointcuts and around advice methods, they will directly construct the advice chain when multiple pointcuts match the same join point, and invoke each around advice method with a closure that represent the remainder of the advice chain. This fixed strategy renders it very complex for an ECOSYS implementation to allow resolutions to manipulate the advice chain. It would be much more practical for our purpose to have a slightly more low-level model where the virtual machine invokes a custom piece of dispatch code with a closure that represents the intercepted join point and a list of the pointcuts that triggered the match.

Bytecode Instrumentation A frequently employed technique for the implementation of aspect-oriented approach is the instrumentation of the application bytecode to include invocations of advice behavior at the static program points that correspond to the relevant join points. If this instrumentation occurs before the program execution, then dispatch behavior will need to be included at all possible join points to check for applicable advice behavior. As explained, some approaches propose to avoid the overhead of such a dispatch mechanism by changing the bytecode of a running application when aspects are deployed.

One of the downsides of bytecode manipulation is that the selection of a run-time join point is done in an indirect manner, through the join point shadow. Additionally, the manipulation of bytecode involves many technical details, even when less low-level bytecode manipulation toolkits such as BCEL from the JAKARTA project or JAVASSIST by Chiba (1998) are employed. To avoid these complications and produce a straightforward prototype ECOSYS implementation, we propose to employ an existing aspect-oriented approach to instrument the bytecode and invoke the dispatch mechanism with relevant information about the join point such that the advice behavior may be applied with full flexibility. Due to its simplicity, this is the technique that is employed for the prototype implementation presented in the next section.

4.3.2 Prototype EcoSys Implementation

A prototype implementation of ECOSYS was developed by employing the facilities of the ASPECTJ approach to realize the interception of the candidate join points in the application. This implementation provides a number of pointcut primitives that may be directly composed in the host programming language. Alternatively, pointcut expressions in the familiar ASPECTJ pointcut language may be employed. This alternative does not diminish the utility of the 'native' ECOSYS pointcuts; rather, it acknowledges that multiple pointcut paradigms may be provide different trade-offs as discussed in Section 4.1.2.

Join point dispatch The first responsibility of an ECOSYS implementation is to complete the join point dispatch mechanism described in Section 4.1.3. To this end, the ASPECTJ around

```
@SuppressAjWarnings("adviceDidNotMatch")
aspect AJCore extends Core<AJJP> {
  pointcut systemScope():
    within(ecosys..*+) && !within(ecosys.Advice+) || within(@DontAdvice *);
  pointcut aroundCapable():
    !handler(*) && !initialization(new(..)) &&
    !preinitialization(new(..)) && !staticinitialization(*);

  pointcut dynamicScope(Object thiz, Object tgt):
    !systemScope() && aroundCapable() && this(thiz) && target(tgt);

  // Dynamic caller and receiver, 0 arguments
  Object around(Object thiz,Object tgt): dynamicScope(thiz,tgt) && args() {
    return dispatch(new AJJP(thisJoinPoint, new RawJoinPoint() {
        public Object invoke(Object thiz, Object tgt, Object... args) {
            assert args.length == 0;
            return proceed(thiz,tgt);
        }
    }),thiz,tgt);
  }

  // Dynamic caller and receiver, 1 argument
  Object around(Object thiz, Object tgt, Object arg1):
    dynamicScope(thiz,tgt) && args(arg1) {
    return dispatch(new AJJP(thisJoinPoint, new RawJoinPoint() {
        public Object invoke(Object thiz, Object tgt, Object... args) {
            assert args.length == 1;
            return proceed(thiz,tgt,args[0]);
        }
    }),thiz,tgt,arg1);
  }

  ... // More cases that follow this pattern

  pointcut staticScope():
    !systemScope() && aroundCapable() && !this(Object) && !target(Object);

  ... // Cases for static caller and receiver
}
```

Listing 4.4: Employing ASPECTJ to instrument all candidate join points with invocations of the dispatch behavior

advice mechanism is employed to instrument all candidate join points with invocations of the dispatch behavior. This is illustrated by the code in Listing 4.4, which shows the extension of the `Core` class specific to this implementation. By defining this extension as an aspect entity with a singleton instantiation strategy on line 2, a single instance of the core registry is automatically created. The body of this aspect defines two named pointcuts to delineate the scope where join point interception should occur (lines 3–4) and to exclude the join point kinds that cannot be intercepted with around advice (lines 5–7). While ECOSYS types and the dispatch mechanism need to be excluded from the instrumentation to prevent an endless loop in the join point interception, the `Advice` types may be included to enable advice behavior to be itself advised (a situation sometimes called "aspects-on-aspects").

The rest of the body consists essentially of a number of around advice methods that will do the actual join points interception. This interception is divided over several advice methods according to the structure of the context of the join point. In case both a caller and receiver are available (as selected by the named pointcut defined in lines 9–10), and there are no other arguments, the advice in lines 13–20 is employed. This advice binds the context parameters and invokes the `dispatch` method that is inherited for the `Core` class with these context values and an instance of the `AJJP` class that groups the information pertaining to the join point. This consists on the one hand of a data structure with the join point properties, as provided by the ASPECTJ **`thisJoinPoint`** construct. On the other hand, the invocation of the **`proceed`** method is wrapped in a closure of type `RawJoinPoint`. This permits the execution of the intercepted join point, possibly with new arguments which are provided to the invocation of **`proceed`** on line 17.

The other around advice methods follow the same pattern and implement the interception for cases with a different number of arguments, or where the caller and/or receiver are not available (this may be the case for the invocation of a static member, for example). Since it may very well be that not all the cases occur in the program the warnings from the ASPECTJ compiler regarding the absence of matches for one of the around advice methods is turned off for the entire aspect with the declaration on line 1.

Pointcut integration The second part of an ECOSYS implementation provides a pointcut language and an integration of the pointcut primitives with the `Pointcut` class from Section 4.1.2. The implementation provides two modes of pointcut integration, according to the two styles that are discussed in this section.

Firstly, a number of pointcut primitives are provided without any dedicated syntax as classes that extend the type `Pointcut<AJJP>` (a pointcut for join point of the implementation type `AJJP`). These primitives implement a `match` method that employs the data structure with properties of the intercepted join point available in the `AJJP` instance to determine a match. Each primitive matches a different kind of join point and they may be configured with the standard types from the reflective facilities of the JAVA language. For example, the primitive `MethodExecution` matches method execution join points and it exposes the receiver of the method to the advice. It may be configured as follows:

```
Pointcut<AJJP> pc = new MethodExecution(
    Modifier.PUBLIC,      // Modifier(s)
    FigureElement.class,  // Declaring type
```

```
    void.class,          // Return type
    "set*"               // Method name pattern
                         // Argument types (none)
);
```

Since the instantiation and composition of these primitives may occur in the JAVA language, it is possible to build abstractions that provide more high-level pointcut primitives, *e.g.*, a method that constructs a pointcut to match the "setter" methods of a class that is provided as an argument.

The second option is to use a standard pointcut expression in the familiar pointcut language of ASPECTJ. The class `AJHostedPointcut` is another pointcut subclass that is configured with a string that contains a pointcut expression. The standard ASPECTJ pointcut parser and matcher[5] is then employed by this class to determine a match.

The combination of the above elements provides a complete implementation of the ECOSYS framework[6]. It provides a very flexible model for the specification of aspects and advice behavior, but since this is a prototype implementation, no efforts were made to provide performance optimizations. The most obvious performance overhead is caused by the invocation of the dispatch mechanism for every candidate join point, and the related construction of data structures for reflective access to this join point, both specified in Listing 4.4. Tracking information about the possible pointcut and resolution entities should provide many opportunities for the optimization of this behavior.

[5] The ASPECTJ implementation makes this functionality available as a part of an independent `org.aspectj.matcher.jar` library. This is done with the specific intent of supporting the pointcut language in other aspect approaches such as SPRING/AOP.

[6] The implementation is available for download at `http://ssel.vub.ac.be/ecosys/`. In addition to the material from this chapter, it includes the typing schemes from Chapter 8.

Chapter 5

Other Approaches for Expressive Deployment

This chapter discusses a number of the aspect-oriented approaches or mechanisms that influence the expressiveness of the deployment logic compared to the situation discussed in Chapter 3. Most of the work discussed in this section has been published simultaneously or after the research that is reported on in Chapter 3 and Chapter 4 was undertaken[1].

Since we do not assume that all readers have specific knowledge of the approaches discussed here, we start the discussion of most approaches with a brief overview of the proposed language elements. After this overview, we evaluate the contributions with respect to the requirements outlined in Section 3.2.

To our knowledge, the set of selected approaches is a reasonably complete representation of the body of related work available in literature. We remark that for some approaches, the authors do not explicitly target the problem of deployment of reusable aspects. In some cases, the approach targets specific parts of the deployment specification (*e.g.*, the instantiation of aspect instances). Other approaches address seemingly unrelated shortcomings in the programming model of the AspectJ-family of aspect languages, or increase the general expressiveness of those languages. The connection to aspect deployment may not be immediately apparent. In such cases, we sketch how the proposed language elements may enable flexible aspect deployment (and perhaps vice versa: some typical use cases of a particular approach may be solved through better deployment support). The discussion in this chapter may therefore help to give an impression of the breadth of deployment issues in aspect languages.

5.1 CaesarJ

CaesarJ is proposed by Aracic, Gasiunas, Mezini, and Ostermann (2006) as an aspect-oriented language which combines aspect-oriented constructs such as pointcuts and advice with advanced object-oriented modularization mechanisms. It shares the aim of the JAsCo approach

[1] One notable exception is the AspectS approach, which exists since 2002.

```
1  abstract cclass TreeDisplay {
2      // visualization
3      abstract void draw();
4      // data model
5      abstract Node getRoot();
6
7      abstract cclass Node {
8          // visualization
9          abstract void draw();
10         abstract boolean isSelected();
11         // data model
12         abstract String getLabel();
13         abstract int getNbChildren();
14         abstract Node getChildAt(int i);
15     }
16 }
```

Listing 5.1: CAESARJ collaboration interface for the display of a tree model

(from Section 2.2.2) to enable aspect components as reusable large-scale aspects. The language also contributes several elements of dynamic aspect control.

5.1.1 Proposal

CAESARJ extends the JAVA language with a number of elements that add roughly one hierarchical composition mechanism and two crosscutting composition mechanisms. We will present each of these mechanisms in succession.

Hierarchical Composition

The designers of CAESARJ consider groups of collaborating classes as the basis for any large-scale piece of functionality. The language therefore allows to group a set of interrelated classes in a modular unit. This grouping is hierarchical: the unit is again a class, and it provides object-oriented concepts such as abstraction, polymorphism and late binding with respect to its member classes. We develop a small example to demonstrate the precise characteristics of this class mechanism.

Consider the display of a tree-like data structure in a GUI tree control. For obvious reasons, it is desirable to have a tree control that can display various data structures that have a tree model. In addition, we may want to support the display of data through multiple tree control implementations. In CAESARJ, the principle elements of interaction for a generic component are laid out in a so called *collaboration interface*. A collaboration interface for the behavior of our example is presented in Listing 5.1. The interface specifies a number of abstract operations as well as types pertaining to the tree display behavior (the new types may again specify new operations and types and so on).

```
1  abstract cclass SimpleTreeControl extends TreeDisplay {
2      void draw() { getRoot().draw(); }
3      void onSelect(Node n) { n.selected = true; }
4      void onDeselect(Node n) { n.selected = false; }
5      abstract cclass Node {
6          boolean selected;
7          boolean isSelected() { return selected; }
8          void draw() {
9              ...getLabel();
10             ...getChildAt(i);
11         }
12     }
13 }
```

Listing 5.2: Implementation of the visualization facet of the `TreeDisplay` collaboration

```
1  abstract cclass DirectModelDisplay extends TreeDisplay {
2      Node root;
3      Node getRoot() { return root; }
4      abstract cclass Node {
5          String label;
6          List children = new LinkedList();
7          String getLabel() { return isSelected() ? "<"+label+">" : label; }
8          int getNbChildren() { return children.size(); }
9          Node getChildAt(int i) { return (Node)children.get(i); }
10     }
11 }
```

Listing 5.3: Implementation of the data model facet of the `TreeDisplay` collaboration

The operations may be logically partitioned according to different facets of the interaction. In our example, a number of operations relate to the visualization of the tree, while others concern its data model. We may define a number of subclasses of `TreeDisplay` that each implement one of these facets. Listing 5.2 presents a subclass that implements the visualization facet and implements a simple tree display control. Listing 5.3 contains a subclass that implements the data model by directly storing root and child nodes as instance data in the classes. Of course, there may be other implementations of these facets as well, for example, a tree control with a different presentation or a data model that stores the children in an array instead of a linked list.

At this point, it is important to highlight that the classes specified with the keyword **`cclass`** are *virtual classes*. This signifies that inner classes are virtual members of the enclosing class that may be overridden in subclasses of the enclosing class. This occurs with the `Node` class in Listing 5.2 and Listing 5.3. The refinement of the inner class is employed in a subclass, for example, in line 3 of Listing 5.2, we access the attribute `selected`, which is a field that is not

```
1 cclass SimpleDirectModelTreeDisplay extends
2     SimpleTreeControl & DirectModelDisplay {}
```

Listing 5.4: Mixin composition of implementations of the facets of the `TreeDisplay` collaboration

present in the definition of `Node` from type `TreeDisplay`. This behavior is noticeably different from standard inner classes in the JAVA language.

More specifically, CAESARJ adopts the *family polymorphism* model proposed by Ernst (2001). In this design, virtual classes and the corresponding types are in fact attributes of *objects*, not attributes of classes. For example, an occurrence of type `Node` in Listing 5.1 is implicitly qualified with the instance of the enclosing class, `TreeDisplay.this`. Similarly, `Node` in Listing 5.2 implicitly stands for `SimpleTreeControl.this.Node`. The `Node` instances of one `TreeDisplay` instance are not compatible[2] with another `TreeDisplay` instance, or in other words, they must be of the same *family* (an enclosing class such as `TreeDisplay` is called a *family class* for this reason). When `Node` is referenced from outside of the family class, it must be explicitly qualified with a `TreeDisplay` instance.

To compose the different subclasses of a collaboration interface, the mechanism of *mixin composition* is employed. Mixins are subclass definitions with a parameterizable parent class: they specify a delta of new or overriding member definitions that may be applied to any class given as an argument. This realizes a variant of multiple inheritance[3] which linearizes the superclasses, thereby avoiding ambiguities with respect to method dispatch and with respect to sharing or duplication of inherited state. (Mixin composition was also briefly considered at the end of Section 2.1.2.) In CAESARJ, the parent link for a subclass such as `SimpleTreeControl` should not be considered the fixed class `TreeDisplay`: another subclass of `TreeDisplay` may take its place. Mixin composition is specified using the `&` operator in CAESARJ, as demonstrated in Listing 5.4. The composition of the two classes in this listing results in a version of the class `SimpleTreeControl` where the place of the superclass is taken by `DirectModelDisplay`. `SimpleDirectModelTreeDisplay` therefore inherits from `SimpleTreeControl`, `DirectModelDisplay` and `TreeDisplay`, in that order. A linearization algorithm defines the result of mixin composition in more complex cases.

It is important to remark that the mixin composition operator also propagates the composition into the inner classes. In the example, the `Node` classes are composed in the same order as the linearization of the enclosing family class. Also, since `SimpleDirectModelTreeDisplay` is a concrete class, the compiler will verify that all the abstract operations have been implemented. This ensures that an implementation for all facets has been provided.

[2] To track this in the type system, *dependent types* are employed. Such types are abstracted over terms, which is rather different from the type systems in this dissertation. We consider the opposite, *i.e.*, terms abstracted over types, in Section 6.2.3.

[3] Because of the availability of multiple *implementation* inheritance, CAESARJ does not need to distinguish between interfaces and abstract classes. This explains why the collaboration interface in Listing 5.1 is defined as an abstract class although none of its operations include an implementation.

Crosscutting Composition

The crosscutting mechanisms supported by CAESARJ are wrapper classes and the pointcut/advice mechanism.

Wrapper classes The above mixin composition mechanism is hierarchical; in order to compose different modules in a non-trivial way, they must have common ancestors because only those inner class definitions are merged that override a common class definition. As a crosscutting alternative, facets can be implemented as adapters of already existing classes. This expresses how the abstractions of a base application should be translated to the vocabulary of a particular collaboration interface (this is termed *on-demand remodularization* by Mezini and Ostermann, 2002). Such an adapter is similar to the object version of the *adapter design pattern* described by Gamma et al. (1995, p.139–150), except that is directly built into the CAESARJ language through the mechanism of *wrapper classes*.

Some examples of wrapper classes are shown in Listing 5.5, where we implement the data model facet of the `TreeDisplay` collaboration using the `FigureElement` classes that we also employed in Chapter 2. (Such a family class which implements a facet of the collaboration interface by adapting external classes is called a *binding*.) The wrapper relationship is established using the keyword **`wraps`**. One wrapper class is needed for each kind of figure element. Wrappers implicitly maintain a reference to an instance of the class that they extend; this reference is available by means of the predefined identifier **`wrappee`**. This wrappee instance is employed to implement the behavior of the required interfaces.

The instantiation of wrappers differs syntactically from the ordinary object instantiation in the omission of the **`new`** keyword (see line 3). The instantiations need to be qualified (implicitly or explicitly) with an instance of the family class that encloses the wrapper class. The constructors of wrappers are implicitly defined; the argument of the constructor is the object to be wrapped, and the result is the wrapper instance. The behavior of a constructor is different from an ordinary constructor since it applies a mechanism called *wrapper recycling*: a new wrapper instance is only created when no previous wrapper associated to the given object exist. The existing wrapper instances are stored in a map that is stored in the outer family class instance. As such, the wrapper instance may be used to store and access additional instance data for the wrapped object relative to the involved family class instance. Multiple wrappers may still exist for the same object when they are each associated with a different family class instance.

Finally, note in Listing 5.5 that we have defined multiple wrapper class with the same name for different classes of the `FigureElement` hierarchy. These wrapper classes will implicitly inherit from each other corresponding to the subtype relationships between the wrapped classes, for example the version of `FigureElementNode` for class `Point` will inherit from the version for class `FigureElement`, but not from the version for `Group`. When the joint constructor of these classes is invoked, it will employ the dynamic type of the object to be wrapped to select the most specific wrapper class (this is called *dynamic wrapper selection*). This feature facilitates to build wrapper classes in a hierarchy that is parallel to the hierarchy of the wrapped classes.

The mechanism of wrapper classes is similar to the open classes mechanisms discussed in Section 2.1.2 in the sense that they allow to add new state and behavior to existing classes from outside of the class definition. However, it is argued by Ostermann and Mezini (2003) that wrappers provide important advantages over the inter-type declarations of ASPECTJ. Most

```
1  abstract cclass FigureTreeDisplay extends TreeDisplay {
2      Figure figure;
3      Node getRoot() { return FigureElementNode(figure.getCanvasGroup()); }
4
5      cclass FigureElementNode extends Node wraps FigureElement { }
6      cclass FigureElementNode wraps Group {
7          String getLabel() {
8              return String.format("Group␣of␣%d␣elements",getNbChildren());
9          }
10         int getNbChildren() { return wrappee.getElementCount(); }
11         Node getChildAt(int i) {
12             return FigureElementNode(wrappee.getElement(i));
13         }
14     }
15     cclass FigureElementNode wraps Point {
16         String getLabel() {
17             return String.format("Point(%d,%d)",
18               wrappee.getX(),wrappee.getY());
19         }
20         int getNbChildren() { return 0; }
21         Node getChildAt(int i) { throw new IndexOutOfBoundsException(); }
22     }
23
24     pointcut figureChange(FigureElement fe):
25         execution(void FigureElement+.move(..)) && this(fe) ||
26         execution(void Group.add(..)) && this(fe) ||
27         execution(void Group.remove(..)) && this(fe);
28     after(): figureChange(fe) {
29         FigureElementNode(fe).draw();
30     }
31 }
```

Listing 5.5: Implementation of the data model facet of the `TreeDisplay` collaboration using existing figure element classes

notably, while for both mechanisms the added behavior may be polymorphic with respect to base object types (wrappees), inter-type declarations do so by invasively changing the classes directly. In contrast, wrappers allow to isolate the extensions with respect to different aspect types and instances. The wrapper mechanism can therefore preserve the property of independent extensibility. Additionally, the "mix-and-match" composition through mixins provides a higher degree of reuse and variability than what is possible with abstract aspects and single inheritance in ASPECTJ.

Pointcuts and Advice The existing classes may also need to adapt their existing behavior within the composition. To this end, CAESARJ adopts the pointcut/advice mechanism from ASPECTJ. This is demonstrated on lines 24–30 of Listing 5.5. Here, we want to trigger a redraw of the display when the figure elements receive changes. This is expressed by advising the relevant methods of the figure element classes. Note that we employ the wrapper constructor of `FigureElementNode` in the advice method (line 30) to retrieve the associated node instance.

The pointcut/advice mechanism of CAESARJ is almost entirely identical to the one from ASPECTJ (although pointcut and advice members may be more flexibly inherited in CAESARJ due to the mixin compositions). One important point of divergence is with respect to aspect deployment, where CAESARJ provides a number of mechanisms for dynamic aspect control. When a concrete subclass is created that inherits advice methods then this class is not implicitly instantiated as is the case for ASPECTJ (see the description in Section 2.2.1). For example:

```
cclass SimpleFigureTreeDisplay extends
    SimpleTreeControl & FigureTreeDisplay {}
```

The advice behavior of this class is only applied after the class has been explicitly instantiated and deployed by means of the built-in **deploy** operation.

```
TreeDisplay d = new SimpleFigureTreeDisplay();
deploy d;
```

After the execution of this operation, the advice methods defined in `FigureTreeDisplay` will be executed in the context of the `d` instance. The advice behavior may be removed with an inverse **undeploy** operation. If such a dynamic deployment is not desired, the modifier **deployed** may be added to the class declaration. This may be considered syntactic sugar for the definition of a static initializer block that creates and deploys a single instance.

One variation in the dynamic deployment is that the **deploy** operation may also be provided with a block:

```
deploy (d) { ... }
```

This will activate the advice behavior before the execution of the statements in the block and deactivate it afterwards. It also limits the advice behavior to join points from the currently executing thread (this could be interpreted as an additional condition that is included in the pointcuts of the advice methods).

```
1  abstract cclass OutputAdvice {
2      abstract pointcut pc(Object o);
3      abstract setOutput(PrintStream o);
4      abstract setOutputLevel(Logging.Level l);
5      abstract void doDeployment();
6  }
7  cclass QoSDeployment extends OutputAdvice {
8      pointcut pc(Object o): ProgramPoints.modelManip(o);
9      void doDeployment() {
10         setOutput(Application.getLog());
11         setOutputLevel(Logging.INFO);
12         deploy this;
13     }
14 }
```

Listing 5.6: CAESARJ interface and example deployment for output advice

5.1.2 Evaluation of Deployment Expressiveness

Without taking anything away from the merits of the CAESARJ programming model, the language elements that contribute to the deployment of pointcut/advice behavior are limited to the mixin inheritance of pointcut and advice members and the facilities for the dynamic deployment of advice behavior.

Reuse of deployment logic

Similar to ASPECTJ, CAESARJ employs the inheritance mechanism for the reuse of deployment elements. In Section 3.2.1, we identified an important restriction for this reuse mechanism due to the single inheritance model of ASPECTJ: it is not possible to inherit both the advice behavior and the deployment elements, so one of these needs to be duplicated. The mixin mechanism of CAESARJ addresses precisely this restriction. Provided that they both descend from the same ancestor, an implementation of the advice behavior and an independent implementation of the deployment elements can be inherited at the same time.

This is illustrated in Listing 5.6, where we present both a general interface for advice behavior that writes to an output stream, and an implementation of the deployment of such advice behavior. The deployment subclass `QoSDeployment` provides the same functionality as the `deployQoS` procedure from page 47, with the exception of the interaction resolution behavior[4]. The class may be composed with other subclasses of `OutputAdvice`, which define advice methods that are bound to the named pointcut `pc` and that implement the remaining methods. As such both the advice behavior and the deployment logic may be inherited.

[4] To the best of our knowledge, the subject of interaction resolution is not discussed in CAESARJ literature.

Deployment control

Compared to ASPECTJ, the deployment model of CAESARJ assigns a different status to some deployment operations. The instantiation of classes that contain advice methods, and the subsequent activation of these advice methods, are no longer predefined operations. They may occur at run-time through the invocation using user-level statements. This brings about advantages for these deployment responsibilities similar to what is the case for ECOSYS, which organizes these operations in a similar manner. However, we remark that we identify at least one other important responsibility of deployment logic in Section 3.1: the configuration of concrete program points by combining the advice method with a pointcut definition. This facet of the deployment is still confined to the static domain in CAESARJ since the combination of pointcuts and advice methods occurs through a static inheritance mechanism. (Put differently, the pointcuts to which the advice methods of a class are bound are fixed at compile-time.) As a result, this configuration of concrete program points does not benefit from these advantages, and unfortunately, there are no provisions to compensate for this on the static level.

Concretely, with respect to deployment quantification, we observe that it is possible to employ the ordinary JAVA control structures to flexibly create a large number of object instances and activate their advice methods. The control structures allow to organize the specification such that it does not grow at the same pace as the number of instances that are created. However, it is not possible to use the control structures to create combinations of a number of pointcuts and a number of advice methods, as in the original example from Section 3.2.2. These combinations are fixed as a part of the static semantics, where no control structures are available. It is therefore required to manually define a new type for every combination of a pointcut and an advice method. This may lead to problematic growth if the number of pointcuts and advice methods is high.

With respect to dynamic deployment, we similarly have the problem that advice methods may not be bound to different pointcuts at run-time (let alone that new pointcuts may be introduced). The only exception is that the application of the advice method may be restricted to join point from the current thread by using a deployment block. Additionally, new object instances may be created at run-time, and since this may be requested through ordinary statements, it is possible to integrate this deployment behavior with the relevant program points of the base application. At this point, data from the program context may be employed for the configuration of the object instance. For example, we may recreate the deployment at the login event from page 48 in CAESARJ:

```
void login(User u) {
    Session s = new Session(u);
    // Deploy instance of tracing advice class for this session:
    deploy new TracingAdvice(s,u.getLog());
    ...
}
```

However, in this example, the advice method of *all* deployed `TracingAdvice` instances will be invoked for operations on *any* session. The advice behavior will first need to check if the intercepted session indeed matches the session for which the `TracingAdvice` instance was triggered.

5.2 Eos

The EOS language by Rajan and Sullivan (2003, 2005) aims to enhance the conceptual integrity of the programming model of aspect languages by unifying classes and aspects, and advice methods and ordinary methods. The authors criticize the model of languages such as ASPECTJ for organizing aspects as module-like constructs (with a single global module instance); it is claimed that the unification contributes to an aspect construct that is more object-oriented, in the sense that (i) aspect instantiation becomes available under program control, and (ii) allows aspects to advise object instances rather than classes.

The motivation for these language changes is the application of aspects for the integration of systems (as described by Sullivan et al., 2002), where it is frequently required to deploy mediators for specific objects.

5.2.1 Proposal

EOS is presented as an extension of the C# programming language. The most important new language element is the *classpect*, which is an ordinary class that may contain named pointcuts and *bindings* among its members, in addition to fields and methods. An example of a classpect is shown in Listing 5.7; a binding is defined on line 6.

A binding consists of a pointcut and the name of a method from the same class. The intent of a binding is to implicitly invoke the specified method as advice behavior at the join points that are matched by the pointcut. Additionally, the binding indicates the advice position (before, after or around) and organizes the transfer of context arguments. The pointcut language is identical to the one from ASPECTJ, with two notable exceptions. In case of after advice, the return value of the join point may be bound using the pointcut designator **`return`**, and in case of around advice, the designator `aroundptr` binds a closure object that may be used to execute the intercepted join point.

Additionally, a binding may be declared static or non-static by including or omitting the keyword **`static`**. In case of a static binding, one singleton instance of the classpect is created. At the join points, the designated advice method is executed in the context of this instance. This is identical to the standard behavior in ASPECTJ. In case of a non-static binding, the classpect must be manually instantiated and *instance-level advising* is employed: only join points that involve a specific set of receivers are advised[5]. This set is configured using the `addObject` and `removeObject` methods that are introduced by the compiler in case of a non-static binding. The binding in Listing 5.7 is non-static, and the instantiation and configuration of the classpect instance are demonstrated on lines 11–12. As a consequence of only adding `s1`, only the first invocation of the `Detect` method is advised, not the second.

Interestingly, the mechanism of instance-level advising is accommodated by EOS with an adjusted implementation strategy. A straightforward implementation would be to store the list of relevant join point objects as a part of the classpect instance data. It is then required to check each classpect instance to determine whether the join point object is relevant, over and over again for each join point. The EOS implementation will 'precompile' the relevant classpect instances per advised object and store this list as a part of the instance data of that object. When

[5] A number of other aspect languages have adopted a similar activation of advice per instance, for example, it is proposed for JASCO by Vanderperren (2004, Sec. 4.2.3.5).

```
class SelectiveTrace {
    void trace(Sensor s) {
        Console.WriteLine("Before␣Sensor␣Detect:␣" + s);
    }

    before(Sensor s): execution(void Sensor.Detect()) && this(s): trace(s);

    static void Main(string[] args) {
        Sensor s1 = new Sensor();
        Sensor s2 = new Sensor();
        SelectiveTrace t = new SelectiveTrace();
        t.addObject(s1);
        s1.Detect();
        s2.Detect();
    }
}
```

Listing 5.7: Example of an EOS classpect that specifies instance-level advising

a join point occurs for the object, the advice method is directly invoked on all of the classpect instances in the list.

5.2.2 Evaluation of Deployment Expressiveness

The critical description by Rajan and Sullivan of ASPECTJ aspects as global modules can be regarded as a report of the rigidness of the aspect deployment model in traditional aspect languages. It should therefore not come as a surprise that the mechanisms proposed by these authors, namely explicit classpect instantiation and instance-level advising, have some positive impact on the flexibility of aspect deployment.

Similar to CAESARJ, EOS organizes the instantiation of aspects as a standard user-level operation. As we discuss in Section 5.1.2 with respect to that language, this gives rise to certain advantages since the operation becomes subject to the abstraction mechanisms and control structures of the programming language, and since the operation can be used from within other parts of the program. However, instantiation is only one element of a deployment specification. Other elements such as the pointcut configuration and the resolution of interactions with other aspects need to be accommodated with similar capabilities.

In this respect, the `addObject` and `removeObject` methods in the case of instance-level advising can be considered user-level deployment operations that allow to configure the pointcut (*i.e.*, the specification of *where* the advice behavior need to be applied). The configuration is restricted to a very specific facet of the pointcut specification, as only object instances may be added to or removed from the pointcut scope, but this particular variation is optimally supported by the implementation technique. The run-time status of this very specific part of the deployment specification makes it amenable to the aforementioned capabilities. We may demonstrate this in the context of our dynamic integration example, where a configuration of

the pointcut to advise only a particular instance is precisely what is required:

```
void login(User u) {
    Session s = new Session(u);
    // Deploy instance of tracing advice class for this session:
    new TracingAdvice(u.getLog()).addObject(s);
    ...
}
```

In comparison to the CAESARJ implementation of this deployment behavior on page 63, the EOS version offers the additional benefit that it is able to configure the pointcut dynamically, based on the available program entities. However, when any other configuration of the pointcut rather than the receiver instances is required (for example, argument instances), the EOS mechanism is no longer sufficient.

5.3 Stateful Aspects and Inter-crosscut Variables

Stateful aspects are proposed by Douence et al. (2002) as a mechanism to employ a protocol of multiple events in the program trace as a pointcut specification for advice behavior. This is not traditionally considered a deployment mechanism, and we will not discuss this mechanism and its different incarnations in full here. However, we remark that this mechanism may be employed to dynamically activate advice behavior at a program event using an aspect that is otherwise statically deployed.

This dynamic deployment behavior is accomplished by specifying a protocol in which the standard application of advice behavior at a program event is 'guarded' (*i.e.*, preceded) by the single occurrence of an activation event (optionally, the advice behavior may be succeeded by a deactivation event as well). In fact, this is not an uncommon use of stateful aspects: for example, 3 of the 6 typical applications presented by Allan et al. (2005, Sec. 2.1) follow this pattern where standard (*i.e.*, not stateful) advice behavior is deployed after a single activation event (and optionally undeployed after a deactivation event).

Douence et al. (2004) additionally propose to extend stateful aspects with *inter-crosscut variables*. This mechanism allows to bind context arguments during the beginning of the protocol, and match new events against these context arguments in the remainder of the protocol. Additionally, the context arguments may be employed in the advice behavior that is eventually triggered. When stateful aspects are employed for the dynamic deployment of advice behavior as we sketch in the previous paragraph, inter-crosscut variables may be employed to configure the pointcut of the advice behavior (and the advice behavior itself) with context information from the activation event. In this way, some integration of the deployment with the context information from the base program is achieved.

A concrete realization of stateful aspects with inter-crosscut variables is provided by the *tracematch* extension of the ASPECTJ language by Allan et al. (2005). Using this approach, aspects may declare tracematches in addition to ordinary advice methods; this is demonstrated in Listing 5.8. A tracematch does not contain a pointcut, but rather defines a number of symbols (lines 3–5) and a regular expression involving these symbols (line 7). The symbols correspond to the event of entry or exit of one of the join points matched by the pointcut given in the symbol

```
1  aspect TraceDeploy extends AbstractTrace {
2      tracematch (Session s) {
3          sym login after returning(s):
4              call(Session.new(..)) && withincode(void login(User));
5          sym sessionOp before: ProgramPoints.sessionOp(s);
6
7          login sessionOp+
8          { traceWith(s.getUser().getLog()); }
9      }
10 }
```

Listing 5.8: Dynamic deployment at a login event using a tracematch

definition (the choice between entry and exit event is specified with the keywords **before** and **after**). The advice body (line 8) is executed when a suffix of the program trace is matched by the specified regular expression. Only program events that correspond to one of the symbols are taken into account; the other events are ignored.

Additionally, the example in Listing 5.8 employs an inter-crosscut variable named `s` which is declared on line 2. Notice that both the symbols `login` and `sessionOp` provide a binding of this variable. Additionally, the value of `s` is referenced in the advice behavior. The semantics of tracematches defines that the advice body is executed for each single variable value that may be bound consistently to all the occurrences of the variable. In our example, this means that an occurrence of the `login` symbol will bind the variable value, and any occurrence of the `sessionOp` symbol will only be considered when its `s` value is equal to a previously bound variable value. As a result, we obtain that at a login event where a session is created, tracing behavior for the operations of that particular session is activated.

We have specifically structured `TraceDeploy` in Listing 5.8 as a deployment of advice behavior that is inherited from an abstract aspect `AbstractTrace`. (Notice that the deployment does only activate the advice behavior but does not create different aspect instances, such that we cannot store the output handle as a part of the instance data and assume that a method `traceWith` is provided where this output handle is provided.) This realization of a form of dynamic and integrated deployment is rather different from the previous implementations of the same behavior in EOS (on page 66) and ECOSYS (on page 48): there is no deployment invocation statement in the program code that corresponds to the activation event (*i.e.*, in the definition of the `inline` method); the activation event is designed by means of a pointcut. If this is considered beneficial, for example because the activation should occur at program points that are scattered across the code base, then this may be supported in the other two approaches as well, by writing a deployment aspects that includes the deployment invocation as a part of its advice code.

5.4 Reflex

REFLEX is a versatile kernel for multi-language AOP proposed by Tanter and Noyé (2005). The intent of the approach is to enable the use of different aspect approaches in the same piece of software, for instance various domain-specific aspect languages. The REFLEX kernel provides a number of building blocks that may be employed by language plug-ins for the implementation of the language. In addition, the kernel provides facilities for the management of interactions between aspects from different approaches.

Although the end developer is not the primary user of REFLEX, it is explicitly considered by the authors that some advanced aspects may be directly implemented as REFLEX configurations. Moreover, the REFLEX building blocks are sufficiently high-level to be used by an end developer in this manner. For this reason, we consider the approach eligible for evaluation with respect to our purposes.

5.4.1 Proposal

At its core, REFLEX defines an API for the transformation of base programs, and one for the composition of such transformations. The API is at a mid-level of abstraction, in between high-level programming languages and low-level code transformation. Language plug-ins translate the aspect specification from a high-level language into REFLEX API invocations. The transformations are then effectuated by the kernel at load-time (using the JAVASSIST library by Chiba, 1998).

The transformation API unifies several aspect-oriented mechanisms (such as pointcuts/advice and open classes) in one weaving model that adapts concepts from (static) metaprogramming. In this model, *links* are the main units of transformation; they consist of a cut and an action and may be either structural or behavioral. A structural link (*S-link*) employs a cut which selects static program elements. The attached action may be structural, in which case new program elements are introduced (similar to the open class mechanism), or behavioral, in which case the effect is similar to ASPECTJ error or warning declarations which signal the existence of certain join points at weaving-time. A behavioral link (*B-link*) employs a cut which selects dynamic program elements; when their action is behavioral, this corresponds to the pointcut/advice mechanism.

Although the actions of behavioral links are applied at run-time, REFLEX will 'set up' these actions during the weaving by inserting hooks which delegate control to a *metaobject*, which is a run-time object which provides the action (*i.e.*, the advice behavior). The concrete usage of the REFLEX API is demonstrated in Listing 5.9. The definition of a B-link consists of a *hookset* (lines 1–5) and a *metaobject definition* (line 7). The hookset is employed while weaving to match the join point shadows (additionally, a B-link may also specify an *activation condition*, which corresponds to the concept of a residue). The metaobject definition specifies which code should be used in the hooks to obtain the metaobject instance. In case of line 7, we specify that new instances of class `Profiling` should be created. Additionally, the link attribute configuration on line 13 indicates that a single hookset instance should be used for all hooks, while the configurations on lines 10–12 indicate that the hook should include invocations of the `start` and `stop` method of the metaobject, respectively before and after the operation of interest.

```
1  Hookset mainOps = new PrimitiveHookset(
2      MsgReceive.class,          // join point kind
3      new NameCS("Main"),        // class selector
4      new NameOS("get","set")    // operation selector
5  );
6
7  MODefinition profiler = new MODefinition.ClassMO(Profiling.class);
8  BLink mainProf = API.links().createBLink(mainOps,profiler);
9
10 mainProf.setControl(Control.BEFORE_AFTER);
11 mainProf.setCall(Control.BEFORE, "Profiling", "start");
12 mainProf.setCall(Control.AFTER, "Profiling", "stop");
13 mainProf.setScope(Scope.GLOBAL);
14
15 API.rules().addRule(new Wrap(tracing,mainProf));
```

Listing 5.9: Deployment of profiling advice behavior using REFLEX

The weaving transformations specified by S-links and B-links are carried out in two consecutive phases: S-link applications and B-link setup (since the structural modifications of S-links may be subject to behavioral cuts, they are considered first). Both phases contains a detection/resolution scheme to manage the interactions between links. When multiple links affect the same program point, this is considered a situation of underspecification, and link composition rules are considered for additional information. For example, on line 15 of Listing 5.9, we define that the link elements of a tracing link should wrap those of the profiling link using the `Wrap` composition operator. If the composition is not fully determined by the composition rules, then an interaction listener is notified; the default implementation will issue a warning regarding the underspecification. Other composition operators, as well as some extensions of this mechanism are proposed by Tanter (2006b).

5.4.2 Evaluation of Deployment Expressiveness

The REFLEX kernel provides an open weaving platform where aspects may be deployed through general JAVA code which invokes the operations of the REFLEX API. It is important to clarify that this deployment code is executed once, *at the time of weaving* (which is normally when the application is loaded). This is fundamentally different from the *run-time* deployment operations that we have discussed for some of the other approaches in this chapter, or for ECOSYS in Chapter 4. It is not possible to attach the operations of the REFLEX API to run-time program events, which hampers the dynamic deployment of aspects and makes it cumbersome to configure the metaobject instances (which are instantiated implicitly or by means of a factory) with run-time program values.

However, by virtue of being specified in a high-level programming language such as JAVA, the deployment code may be structured using procedural abstractions and the common control structures. This enables to specify reusable deployment code, and provides deployment quan-

tification. For example, it is trivial to deploy links for each combination of a number of advice methods (specified by metaobject definitions) and program regions (specified by hooksets).

```
for(MODefinition mod: ...)
    for(Hookset hs: ...)
        API.links().createBLink(hs,mod);
```

As a matter of fact, a dedicated language with a concrete syntax for the operations of the REFLEX API is proposed as an extension by Tanter (2006a). Deployment code specified using this dedicated language lacks the properties of reuse and deployment quantification since the language lacks the high-level features of JAVA. One interesting property of this dedicated language is that REFLEX statements may be placed inside of JAVA programs. This kind of integration may be able to provide the best of both worlds.

5.5 AspectS

ASPECTS is an aspect-oriented approach for the SQUEAK SMALLTALK environment proposed by Hirschfeld (2003). It is designed as a dynamic AOP approach which leverages the features of the dynamic programming language SMALLTALK. One of its applications lies in the domain of run-time adaptability: the modification of systems at run-time in order to carry out changes that were not anticipated during development. This feature is crucial for systems with strong availability demands (for example, telecommunications). This application of ASPECTS is discussed by Hirschfeld and Lämmel (2004).

5.5.1 Proposal

ASPECTS extends the SMALLTALK metaobject protocol in order to provide the aspect-oriented mechanisms of pointcuts/advice and introductions. It employs *method wrappers* to instrument both message sends and receptions. Method wrappers changes the SMALLTALK method lookup process to return a decorated method instead of the original compiled method. ASPECTS is very much a framework approach since it is implemented entirely on top of SMALLTALK itself.

In ASPECTS, aspects are subclasses of the predefined `AsAspect` class. Advice methods are implemented by defining methods with a name that begins with the word `advice`. This method is not the advice itself; instead it should return an `AsAdvice` object which specifies both the advice behavior and its activation criteria. The predefined `AsAdvice` kinds are: `AsBeforeAfterAdvice` (to add behavior before or after a join point), `AsHandlerAdvice` (to catch an exception thrown by a join point), `AsAroundAdvice` (to replace a join point) and `AsIntroduction` (to introduce new behavior).

Because of the implementation using method wrappers, the activation of an advice is specified using two elements.

1. The first is the pointcut, which is specified by means of a list of `AsJoinPointDescriptor` instances. Each join point descriptor denotes one target for the weaving process (*i.e.*, a method which can be wrapped). When multiple join points need to be specified, multiple joint point descriptors can be enumerated manually, or a list of join point descriptors

```
1  AsAspect subclass: #TimeAspect
2      instanceVariableNames: 'qualifier_pointcut_output'
3      classVariableNames: ''
4      poolDictionaries: ''
5      category: 'Demo'
6
7  TimeAspect>>adviceShowTime
8      ^ AsBeforeAfterAdvice
9          qualifier: self qualifier
10         pointcut: self pointcut
11         beforeBlock: [:receiver :arguments :aspect :client |
12             self output show: '[', Time now printString, ']']
```

Listing 5.10: Reusable AspectS aspect to log a timestamp before operations

can be *calculated*. So interestingly, ASPECTS does not employ type or method patterns in order to quantify over join points; instead, some code block will use the reflective facilities of SMALLTALK in order to find all desired methods, and it will construct a join point descriptor for each of them and return the resulting list.

2. The second activation element is the *advice qualifier*. Using a number of attributes, this qualifier determines the activation block for the advice: the activation block returns a boolean to indicate whether the involved method wrapper should be activated or not. The activation block may check for dynamic conditions that complement the selection done by the pointcut. It has access to SMALLTALK's activation stack and may therefore implement conditions related to the sender class, involved sender or receiver instances, or general control-flow. Two frequently used advice qualifier attributes are `#receiverClassSpecific`, for an activation test that applies no further conditions, and `#cfFirstClass`, for an activation test which examines the stack for one or more senders with the same class as the receiver's.

An example of an ASPECTS aspect is shown in Listing 5.10. The aspect defines one before advice (defined in method `adviceShowTime`). This advice will print a timestamp before the execution of the intercepted operation. In order to prepare for the reuse of this aspect behavior, the configuration elements have already been placed in instance variables. These elements are the pointcut (`pointcut`), the advice qualifier (`qualifier`), and the stream where the time stamp needs to be printed (`output`). In order to deploy this reusable aspect, we may create an aspect instance with a value for these configuration elements, and invoke the `install` method in order to trigger the weaving by ASPECTS:

```
(TimeAspect
    qualifier: (AsAdviceQualifier
        attributes: { #receiverClassSpecific. #cfFirstClass. })
    pointcut: [{AsJoinPointDescriptor
        targetClass: TranscriptStream
```

```
        targetSelector: #show:. }]
    output: Transcript) install
```

As a result of this deployment, the advice behavior will be applied for the reception of `show:` messages by instances of the class `TranscriptStream`. The timestamp will be written to the standard console identified by the name **`Transcript`**. Since the advice behavior itself includes an invocation of `show:` on a `TranscriptStream`, we must include the `#cfFirstClass` advice qualifier in order to prevent triggering an endless loop.

5.5.2 Evaluation of Deployment Expressiveness

The fact that ASPECTS is able to organize the deployment specification as ordinary run-time code is very beneficial for the deployment expressiveness. We may observe that it meets the qualifications of first-class deployment procedures, similar to ECOSYS, since it provides: (i) deployment procedures parameterized with aspects, pointcuts, *etc.*, (ii) control structures, (iii) deployment procedure invocation from base program. The only limitation is that ASPECTS does not provide support for the resolution of aspect interactions: to our knowledge, the execution order in case of multiple advice instances that intercept the same join point, is fixed (cfr. Hirschfeld, 2003, Sec. 5.2). The fact that ASPECTS otherwise offers first-class deployment procedures has a direct impact on the possibilities for the reuse, quantification and dynamism/integration of deployment code.

As an example of deployment reuse, we may recreate the `deployQoS` example from page 47 in ASPECTS:

```
deployQoS: anOutputAspect
    (anOutputAspect
        qualifier: (AsAdviceQualifier
            attributes: { #receiverClassSpecific. })
        pointcut: modelManip
        output: Application log) install
```

In this example, we define a `deployQoS` method which takes an aspect as its argument. The method will deploy the given aspect according to a specific strategy. This may be employed to deploy multiple aspects in the same manner:

```
deployQoS: TimeAspect.
deployQos: SecurityAspect.
...
```

As an example of deployment quantification, we may deploy an aspect instance for each combination of an aspect and a pointcut from a set of aspects and a set of pointcuts. This may be done using a specification that does not grow as fast as the number of deployed aspect instances if we employ a nested loop:

```
{ asp1. asp2. asp3. } do: [:asp |
    { pc1. pc2. pc3. } do: [:pc |
        (asp
            qualifier: (AsAdviceQualifier
                attributes: { #receiverClassSpecific. })
            pointcut: pc) install ]]
```

It is clear that an example involving dynamic and integrated activation of deployment logic may be created in a similar fashion.

In retrospect, the relatively large deployment expressiveness of ASPECTS might not come as a surprise given that it is an open system built on top of a full-fledged metaobject protocol. Since the distinction between the framework code and the user code is (deliberately) blurred, practically all of the expressiveness of the metaobject protocol is made available to the user, which is also more than what is needed for our goals. In comparison to this approach, the benefit of first-class deployment procedures and ECOSYS is that the required expressiveness elements are clearly identified, such that they may be harnessed and better support may be provided (for example in the form of an automatic verification of the safety of the deployment logic, as we explore in Part II of the dissertation).

5.6 Summary

A summary of the evaluation of the deployment expressiveness of the highlighted approaches from the previous sections is presented in Table 5.1. This overview indicates that some approaches (*e.g.*, CAESARJ and ASPECTS) address all of the different expressiveness properties that are discussed in Section 3.2, but do not support all of the different deployment responsibilities from Section 3.1. The other way around, other approaches (*e.g.*, REFLEX) support all facets of the deployment logic but are fundamentally lacking certain expressiveness for this deployment logic.

As a conclusion, we say that while the other approaches overlap in different degrees with our work, and provide solutions that may be employed as alternatives to our proposal, only first-class deployment procedures (and their realization in the ECOSYS framework) provides *all* of the discussed expressiveness properties for *all* facets of the deployment logic.

	Reuse	Quantification	Dynamism/Integration
CaesarJ	PC, I/C	I/C	I/C
Eos	(PC), I/C	(PC), I/C	(PC), I/C
Stateful aspects with inter-crosscut variables	-	-	(PC), (I/C)
Reflex	PC, I/C, R	PC, I/C, R	-
AspectS	PC, I/C	PC, I/C	PC, I/C
First-class deployment procedures (EcoSys)	PC, I/C, R	PC, I/C, R	PC, I/C, R

Table 5.1: Summary of the deployment expressiveness of the discussed approaches. The columns correspond to the different expressiveness properties that are proposed in Section 3.2. The values in each column list the facet of the deployment logic for which this property is offered. These facets correspond to the deployment responsibilities from Section 3.1, with the following abbreviations: "PC" for configuration of concrete program points, "I/C" for instantiation and configuration of the instances, "R" for interaction resolution. Values in (parentheses) signify that the property is only supported for specific parts of this facet.

Part II

Safety of Deployment Logic

Chapter 6

Subtype and Parametric Polymorphism

This chapter discusses a number of type systems and type system features outside of aspect-oriented programming. It establishes the preliminaries to work on the typing of pointcuts and advice in the rest of the dissertation. Section 6.1 presents some important type system concepts and terms in an abstract manner. Most notably the central notions of subtype and parametric polymorphism are introduced. Section 6.2 then gradually builds up a concrete type system with both these styles of polymorphism (and their combination) for a simple language with first-class functions. This allows to present the detailed motivation for, and the different realizations of, subtype and parametric polymorphism in a simplified setting with very little complexity from other language features. The resulting type system is directly employed to discuss pointcut and advice typing in Chapter 7. It is also used in Section 6.3 of this chapter, which explains how the Generics feature of JAVA 5 adds parametric polymorphism to the type system with object-oriented subtyping from previous JAVA versions. JAVA 5 Generics is heavily employed to develop a typed version of the ECOSYS framework in Chapter 8.

6.1 Concepts and Terminology

The following two subsections carry the titles and explain the main concepts from two seminal papers in the field. The first paper is by Cardelli and Wegner (1985); it classifies and explores different abstraction techniques for achieving polymorphism in type systems. This work focuses on the expressiveness of type systems. The second paper is by Wright and Felleisen (1994) and proposes a manner to (formally) connect type systems to the programming language semantics to evaluate their soundness guarantees. This technique has since become commonplace.

6.1.1 On Types, Abstraction and Polymorphism

Type systems classify program objects according to their usage or behavior: sets of objects with uniform behavior may be named and are referred to as *types*. Program objects may

be data values here, but also functions, expressions, objects in the sense of object-oriented programming, *etc.* The rules of a type systems are generally designed to ensure that composite program objects are *type-consistent*, *i.e.*, that the abstractions represented by the types of their constituents are compatible. As such, types impose constraints which help to enforce correctness: those programming errors that correspond to type violations can be automatically detected. In a *static type system*, where types are determined for all static program elements, errors can be detected early, before program execution. Additionally, the type information may be exploited by a compiler to organize greater execution-time efficiency, or by a development environment to make the program source code easier to navigate.

The downside of static typing is that its conservative notion of correctness may lead to a loss of flexibility and expressive power. By (prematurely) constraining the behavior of objects to that associated with a particular type, certain valid and useful programs may be rejected. Generally, this problem can be alleviated by employing a more sophisticated type system that can track properties of larger classes of programs. In this respect, *type polymorphism* has emerged as an important technique to generalize conventional type systems. Traditional typed languages, such as PASCAL, are said to be *monomorphic*, since they are based on the idea that their program objects (in this case functions, procedures, and their operands) have a unique type. This may be contrasted with *polymorphic* languages, which allow values, variables and other program objects to have more than one type. Or interpreted differently, they allow program objects to have a polymorphic type, *i.e.*, a type whose operations are applicable to values of more than one type.

Different ways of organizing polymorphism are used in both theory and practice. Cardelli and Wegner (1985) distinguish between the apparent polymorphism of syntactic convenience features such as coercions and overloading (termed *ad-hoc polymorphism*) and the true polymorphism of universal operations (*universal polymorphism*). While the constructs of the first group are replaced by a predefined finite set of monomorphic operations after disambiguation, true universal operations work on an open-ended set of types that meet a certain criterion. In addition, the authors identify two common styles of universal polymorphism:

Subtype (inclusion) polymorphism partially orders types according to a *subtype* relation; the objects of a subtype are considered to be included in all *supertypes*, so objects of a subtype can be uniformly manipulated as if belonging to their supertypes (in this sense the same object has many types). This inclusion structure will require that objects of subtypes support at least the operations of their supertypes.

Inclusion polymorphism can be found in many class-based object-oriented languages (with SIMULA 67 as the earliest example). The inheritance relation between classes is normally used as the basis for the subtype relation.

Parametric polymorphism allows to type program objects with a specific type structure; it allows to express relations between constituents in the type structure while keeping the actual constituents unknown. An example is a function that returns a result of the same type as its argument, regardless of this argument/result type. This kind of polymorphism derives its name from the common practice of connecting related constituents using *type parameters*: the example function takes an argument of type T and returns a result of type T, where T is a formal type parameter that may be bound to any type value.

> The paradigmatic language for parametric polymorphism is ML (Milner, 1984), which is entirely built around this style of typing.

Note that parametric polymorphism allows to treat object uniformly, irrespective of any specific aspects of the constituent types; it is only concerned with the type relations between these constituents. While inclusion polymorphism is also able to treat all objects uniformly by including them in a *top type* that is a supertype of all other types, this is of limited use from a typing perspective, since the top type can only offer completely general operations. In essence, all static information is lost when manipulating program objects by means of a top type. In contrast, parametric polymorphism allows to retain static types even for operations that work uniformly on all objects, since the operation can have a type structure where constituents are interrelated. This difference is concretely illustrated in Section 6.2.3.

Both types of polymorphism are considered complementary and can be combined in the same type system, as we will see in later sections.

6.1.2 A Syntactic Approach to Type Soundness

As we mentioned in the previous section, the rules of a static type system verify the consistency of the types in a composite program object in order to prevent type violations during the execution of the program. A type system is *sound* when the type checking provides an actual guarantee of the absence of type errors for all possible executions of the program. The typing rules are sometimes called the *static semantics* of the language, while the standard semantics that govern the program execution is referred to as the *dynamic semantics*. With this terminology, soundness can be interpreted in a wider sense as the guarantee that the static semantics is a valid abstraction of the dynamic semantics (for some appropriate definition of "valid abstraction"). Soundness is an essential property from which a type system derives most of its usefulness.

To rigorously define and evaluate this (and other) type system properties, we need to deal with the basic aspects of programming languages in a formal (*i.e.*, precise, clear, mathematically tractable) manner. While the means to formally define the syntax and typing rules are seldom varied (typically a formal grammar and type inference rules are used), different styles have been used to describe the evaluation: *e.g.*, denotational, operational and axiomatic definitions of the dynamic semantics may be used. Unsurprisingly, the choice of semantic style has a direct impact on the definition and evaluation of the soundness property.

Wright and Felleisen (1994) show how an operational formulation of a language's semantics as a series of rewriting steps allows simpler soundness proofs that use a standard structure and straightforward proof techniques. Using this technique, the semantics of a programming language is modeled as a calculus, *i.e.*, as a system of symbolic manipulation of programs. Each intermediate step of an evaluation of a program is itself a program, and the evaluation is performed by successive *reductions* into a new state. Reductions may either continue forever, or may reach some final state where the semantics allows no further reduction. When this final program adheres to a certain predefined structure that allows it to be interpreted as a primitive value, the computation has completed. Otherwise the final state is taken to be a type error; a run-time type error is thus indirectly defined as a *stuck state*: a non-primitive program for which the semantics allows no reductions. Soundness can be formulated as the property that

the reduction of well-typed programs will only halt when a primitive result is obtained. The slogan "well-typed programs don't get stuck" is sometimes used to summarize this definition.

The strategy for proving soundness is then based on the observation that, since the intermediate states of evaluation are programs in this formulation of the semantics, the type system may deduce a type for each state. The correspondence of the typing rules to the reduction rules can therefore be shown rather directly, through the following two properties which are sufficient conditions to establish soundness:

Preservation Reductions preserve type, *i.e.*, any reduction of a well-typed program is again a well-typed program.

Progress Well-typed programs are not in a stuck state, *i.e.*, a well-typed program is either a primitive value or can be reduced to another program.

The technique of proving soundness of a type system by modeling the semantics as a calculus and showing progress and preservation properties, is now widely used. Lambda calculus (Barendregt, 1985) (or one of its extensions) typically serves as a foundation for functional and procedural programming languages, while object calculi (Abadi and Cardelli, 1996; Igarashi et al., 1999) have been developed for object-oriented languages.

6.2 Subtype and Parametric Polymorphism for Functions

This section will explore the concepts of subtype polymorphism and parametric polymorphism by gradually building them into a simple functional programming language. This serves two purposes, both of which originate from the fact that such a language can have a simple mathematical foundation. First, it allows to present the typing concepts in a very general form, with minimal interference from other language features. The language is also used for this purpose in Chapter 7. Second, the language is simple enough to define its typing rules *formally*. This section can be used as a light introduction to the notations and techniques from the formal arsenal that will be used in some of the later chapters.

Although the language presented in this section is not derived from one particular source, almost all of its elements are known from literature. The presentation starts with a monomorphic typed functional language that corresponds rather directly to the simply-typed lambda calculus (Church, 1940). It is then gradually extended with features from the FUN language by Cardelli and Wegner (1985), adding support for subtype polymorphism and various forms of parametric polymorphism (and the combination of the two). A typed calculus with corresponding features is known as $F_{<:}$ ("F sub", see Cardelli et al., 1994), a system which has played a central role in studies on the foundations of object-oriented programming. More background is given in the textbook on type systems by Pierce (2002).

One important difference with systems from literature is that we will motivate and include support for quantifying type variables with lower bounds in addition to the traditional upper bounds. This feature anticipates the type systems for aspect languages developed in the following chapters.

Lexical metavariables

x, y	*term variables*
I	*labels*

Term expressions

e	::=	**true** | **false**	*boolean constants*
	|	**if** e_0 **then** e_1 **else** e_2	*boolean test*
	|	0 | 1 | -1 | ...	*integer constants*
	|	$e_1 + e_2 \mid e_1 == e_2$	*integer operations*
	|	x	*variable term*
	|	**fun** x : T => e	*function abstraction*
	|	$e_0\ e_1$	*function application*
	|	$\{I_1 = e_1, \ldots, I_n = e_n\}$	*record construction*
	|	$e.I$	*record projection*

Type expressions

S, T, U	::=	**Bool**	*boolean type*
	|	**Int**	*integer type*
	|	T_1 -> T_2	*function type*
	|	$\{I_1 : T_1, \ldots, I_n : T_n\}$	*record type*

Typing context (assumptions)

A	::=		*empty context*
	|	$A; x : T$	*term variable type*

Term expression typing $\boxed{A \vdash e : T}$

TRUE
$$A \vdash \mathbf{true} : \mathbf{Bool}$$

FALSE
$$A \vdash \mathbf{false} : \mathbf{Bool}$$

IFTHENELSE
$$\frac{A \vdash e_0 : \mathbf{Bool} \qquad A \vdash e_1 : T \qquad A \vdash e_2 : T}{A \vdash (\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) : T}$$

INT0
$$A \vdash 0 : \mathbf{Int}$$

INT1
$$A \vdash 1 : \mathbf{Int}$$

$\cdots$

PLUS
$$\frac{A \vdash e_1 : \mathbf{Int} \quad A \vdash e_2 : \mathbf{Int}}{A \vdash (e_1 + e_2) : \mathbf{Int}}$$

EQUAL
$$\frac{A \vdash e_1 : \mathbf{Int} \quad A \vdash e_2 : \mathbf{Int}}{A \vdash (e_1 == e_2) : \mathbf{Bool}}$$

VAR
$$\ldots; x : T; \ldots \vdash x : T$$

ABS
$$\frac{A; x : T_1 \vdash e : T_2}{A \vdash (\mathbf{fun}\ x : T_1 \Rightarrow e) : T_1 \to T_2}$$

APPL
$$\frac{A \vdash e_0 : T_1 \to T_2 \qquad A \vdash e_1 : T_1}{A \vdash (e_0\ e_1) : T_2}$$

RECD
$$\frac{A \vdash e_1 : T_1 \qquad \cdots \qquad A \vdash e_n : T_n}{A \vdash \{I_1 = e_1, \ldots, I_n = e_n\} : \{I_1 : T_1, \ldots, I_n : T_n\}}$$

PROJ
$$\frac{A \vdash e : \{I_1 : T_1, \ldots, I_n : T_n\}}{A \vdash e.I_i : T_i}$$

Figure 6.1: Definition of syntax and typing rules for simply-typed functions. Syntactic categories are represented by metavariables (in *italic*) which always bind elements of that category. They are the nonterminals of the language (the terminals are in `teletype`). The typing judgment is defined as a series of inference rules: the statements above the horizontal bar represent conditions; the statement below the bar is the conclusion (the horizontal bar may be omitted if there are no conditions). The appearing metavariables are always implicitly scoped over the entire rule.

6.2.1 Simply-Typed First-Class Functions

The definition of the syntax and typing rules for a functional language with simply-typed functions is given in Figure 6.1. The main elements of the language are terms and types. The typing rules will associate a type with certain well-formed terms based on a number of assumptions regarding the context where the term appears. The notation for this typing judgment is $A \vdash e : T$, to be read as "term e has type T under assumptions A".

Language constructs

The language contains a number of base types and values and their primitive operations. To keep the presentation simple, only integers and booleans are included. The values of these base types have a fixed type regardless of their context, as indicated by the unconstrained metavariable A in rule TRUE, rule INT0, *etc.* The operations on these types are well known from almost any programming language.

The language also allows to build anonymous functions (or *lambdas*) to abstract terms over other terms, represented by variables. Although only functions with one argument are allowed to keep the language minimal, multiary functions can be easily simulated using a series of unary functions, as we will see below. As in any functional language, the functions in our language are first-class terms. Such terms have a function type T_1 -> T_2 that captures the domain (T_1) and codomain (T_2) of the function. This information allows to adequately type the application of a function to an argument in rule APPL.

The typing of the function abstraction itself (rule ABS) will require a typing of the function body in a context that is enriched with a term variable of the declared type. So when typing the function **fun** x **:** T => e, it is verified that the body e is well-typed under the assumption that variable x has type T, as well as any other assumptions from the context of the function, if it is a nested term. To avoid confusion between the new variable and variables from the context, we (silently) require that the name x be chosen so that it is distinct from the names of the variables that already exist in the context. This is not a big restriction, since variables bound by function abstractions can always be renamed when it is required (it is a general convention to only distinguish function abstractions up to renaming of bound variables). Also, we note that we explicitly declare the type T of the argument x in a function abstraction to make the type checking process trivial. In some concrete programming languages, such a type specification may sometimes be omitted. The compiler will then reconstruct the type of the argument based on its usage in the function body. Type reconstruction is discussed by Pierce (2002, Ch. 22).

Example. We illustrate the complete process of typing the term **fun** x **: Int** => x+1 through a derivation tree that employs multiple type rules in succession, starting from type rules that have no conditions, up to a conclusion that is a typing for this term:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x} : \mathbf{Int} \vdash \mathtt{x} : \mathbf{Int}}\,\textsc{Var} \qquad \dfrac{}{\mathtt{x} : \mathbf{Int} \vdash 1 : \mathbf{Int}}\,\textsc{Int1}}{\mathtt{x} : \mathbf{Int} \vdash \mathtt{x}+1 : \mathbf{Int}}\,\textsc{Plus}}{\vdash (\mathbf{fun}\ \mathtt{x} : \mathbf{Int} \Rightarrow \mathtt{x}+1) : \mathbf{Int} \to \mathbf{Int}}\,\textsc{Abs}$$

It may seem a little strange that type checking is presented as a free reasoning process: for any particular term, can the compiler always decide whether it is well-typed, and if so, will the type

be fixed? Note however that the typing rules satisfy the following properties: (i) there is exactly one typing rule for each syntactic form (the typing rules are *syntax-directed*), (ii) as conditions, each typing rule requires only typings of subterms of the term treated in the conclusion, (iii) the type used in the conclusion of a type rule is never *free*: it is either non-variable or a variable bound by one of the conditions. As a consequence of these properties, it is easy to convert the type rules to a recursive procedure that can always determine a type for each term, or otherwise indicate a type error in a subterm.

Lastly, the language also includes a record construct that acts as a general compound data structure. A record groups multiple terms, possibly of different type, where each term is associated with a unique static identifier (a *label*), for example, `{fst=1, snd=true}` (we say this record has *fields* `fst` and `snd`). The type of a record term will be a record type that contains both the names of each field, as well as their type (rule RECD), in this case, `{fst : Int, snd : Bool}`. Once again, this information allows to adequately type the term where a field is accessed from a record (called a *projection*) in rule PROJ.

We point out that records are included in our language mainly for didactic purposes: record types provide a very lightweight approximation of object interfaces, and they allow for a simple and intuitive subtype relation that we introduce below, in Section 6.2.2. Records also play an important role in the object encodings from literature (see for example Pierce, 2002, Ch. 18, 27, 32), but we stress that we do not recreate those results here. As an educational device, we will favor the intuitive simplicity of plain record types over a complete modeling of all of the features of object orientation (which would include encapsulation, inheritance, open recursion, ...).

Semantics

Since a rigorous evaluation of the soundness of the type system is beyond the scope of this introduction, we will discuss the dynamic semantics of this language informally. The semantics of the language constructs corresponds to their well-known behavior from functional programming languages.

Concretely, the evaluation of programs can be defined as the rewriting of terms into primitive terms that we categorize as values (*cf.* Section 6.1.2). The rules for rewriting are:

(i) primitive operations on argument values of the appropriate base types may be replaced by their result,
(ii) the boolean test with **`true`** (*resp.* **`false`**) as *condition*-term may be replaced by its *then*-term (*resp.* its *else*-term),
(iii) the application of an argument term to a function abstraction may be replaced by the body term of the function, where the argument term is substituted[1] for the function variable in the body, and
(iv) the projection of a record according to a label may be replaced by the term associated with that label in the record (if there is such a term).

Inside a larger term, subterms may be rewritten according to these rules as well. There is no fixed evaluation strategy that determines the order in which this should occur.

[1] With the standard caveat that the substitution should avoid the capture of free variables. This is explained by Pierce (2002, Sec. 5.3).

Top-level bindings

To avoid repeating complex terms in the presentation, we will occasionally define top-level bindings of term variables. The variables can be considered to be bound in a global context: after definition, the variables can be used in any context where it is not overridden by a local binding of the same variable. For clarity, we will show both the term (after keyword **let**) and type (after keyword **val**) when defining a top-level binding of a variable, for example:

```
let one = 1
val one : Int
```

The variable one can now be used in any succeeding term, including other top-level bindings:

```
let succ = (fun x : Int => x + one)
val succ : Int -> Int
```

When binding variables to function terms, such as the above succ, we can use the following abbreviation:

```
let succ (x : Int) = x + one
val succ : Int -> Int
```

When specifying multiple arguments this is taken to be an abbreviation of multiple nested single-argument functions (a technique called *currying*). For example:

```
let plus (x : Int) (y : Int) = x + y
val plus : Int -> Int -> Int
```

Note here that the arrow operator in function types is right associative, so the type **Int** -> **Int** -> **Int** is to be interpreted as **Int** -> (**Int** -> **Int**).

6.2.2 Subtype Polymorphism

Motivation

The type system presented so far is entirely monomorphic. We encounter a motivation for a first form of polymorphism when considering a function term such as **fun** x => succ (x.fst). The argument x can be declared with any of a series of record types, all of which allow to obtain a well-typed function body because they all have a field fst of type **Int** that can be accessed through projection (rule PROJ):

$$\begin{gathered}\{\texttt{fst} : \textbf{Int}\} \\ \{\texttt{fst} : \textbf{Int}, \texttt{snd} : \textbf{Int}\} \\ \{\texttt{fst} : \textbf{Int}, \texttt{snd} : \textbf{Bool}\} \\ \{\texttt{fst} : \textbf{Int}, \texttt{snd} : \textbf{Int}, \texttt{trd} : \textbf{Bool}\} \\ \dots\end{gathered}$$

However, if we pick one of these types as argument type, the function cannot be applied to terms of the other types (rule APPL). Similarly, consider the following term which cannot be

Type expressions

S, T, U ::= … *(previous forms)*
| **Top** *top type*
| **Bot** *bottom type*

Term expression typing $\boxed{A \vdash e : T}$

$$\ldots \qquad \textsc{Sub}\ \frac{A \vdash e : S \qquad S <: T}{A \vdash e : T}$$

Subtyping $\boxed{S <: T}$

$$\textsc{SRefl}\ \frac{}{T <: T} \qquad \textsc{STrans}\ \frac{S <: T \qquad T <: U}{S <: U} \qquad \textsc{STop}\ \frac{}{T <: \mathbf{Top}} \qquad \textsc{SBot}\ \frac{}{\mathbf{Bot} <: T}$$

$$\textsc{SFun}\ \frac{T_1 <: S_1 \qquad S_2 <: T_2}{(S_1 \to S_2) <: (T_1 \to T_2)} \qquad \textsc{SRecd}\ \frac{S_1 <: T_1 \qquad \cdots \qquad S_n <: T_n}{\{I_1 : S_1, \ldots, I_n : S_n, \ldots, I_m : S_m\} <: \{I_1 : T_1, \ldots, I_n : T_n\}}$$

Figure 6.2: Extension of the language of Figure 6.1 with subtypes

typed because the two branches of the boolean test do not have the same type (rule IfThenElse), although both of these types support the operation that is eventually applied to the result:

```
(if c then {fst=1} else {fst=0, snd=true}).fst
```

To support these examples, we need to formalize the intuition that some types are more specific than others. We say that a type is a *subtype* of another type to mean that any term of the first type can safely be used in a context where a term of the second type is expected (in the other direction, the term *supertype* is used). This view of subtyping is often called the *principle of safe substitution*. It requires that the subtype offers at least the operations (or satisfies at least the contract of) the supertype. For example, `{fst : Int, snd : Bool}` is a subtype of `{fst : Int}`, since the first type also admits the sole operation admitted for the second type, namely the projection of a field with name `fst` and type `Int`.

The subsumption rule

An extension of the functional language of the previous section with subtype polymorphism is shown in Figure 6.2. We define a subtype relation between types (written $S <: T$ to stand for "S is a subtype of T"), and add one important typing rule, generally known as the *subsumption rule* (rule Sub). This rule attaches a meaning to the subtype relation by connecting it to the derivation of term typings. Concretely, it states that if a term has a certain type, it can be regarded as a term of any supertype of that type as well. The subtype relation needs to be defined in such a way that we cannot obtain term typings that allow unsafe operations.

Note that contrary to the typing rules we have seen before, the subsumption rule has the potential to type any kind of term (indicated by the bare metavariable e). Subtyping can therefore be considered a crosscutting extension, one that interacts with most other language features in non-trivial ways. We will treat each of the language features by considering the subtype relation for each kind of type below.

The subtype relation

The definition of the subtype relation is given through a set of inference rules for the subtype judgment $S <: T$. The first two general subtyping rules state that the subtype relation is both reflexive and transitive (*i.e.*, that is a *preorder*). These properties follow from the intuition of safe substitution, but they can also be considered a regularization of properties that are already present in the structure of the subsumption rule. Indeed, in case of rule SREFL, the conclusion $T <: T$ does not allow to derive additional typings, since the subsumption rule will simply have a conclusion that is equal to its first condition in that case. And in case of rule STRANS, we could also apply the subsumption rule twice (as shown on the left) to obtain the same typing that the conclusion $S <: U$ enables in one application of the subsumption rule (as shown on the right):

$$\textsc{Sub}\ \dfrac{\textsc{Sub}\ \dfrac{A \vdash e : S \qquad S <: T}{A \vdash e : T} \qquad T <: U}{A \vdash e : U} \qquad\qquad \dfrac{A \vdash e : S \qquad \dfrac{S <: T \qquad T <: U}{S <: U}\ \textsc{STrans}}{A \vdash e : U}\ \textsc{Sub}$$

Two special new types are added to the type system, defined axiomatically through their special subtype relations. The type **Top** ("the top type") is a supertype of any type (rule STOP), while type **Bot** ("the bottom type") is a subtype of any type (rule SBOT). Together with the subsumption rule, this entails that any well-typed term also has type **Top**, while a term of type **Bot** can be used as a term of any other type. No operations are defined for terms of the 'most-general' type **Top**: according to the safe substitution principle, only operations common to all types could be allowed, but no such operations exist in our language[2]. Dually, no typing rules are defined that assign terms the 'most-specific' type **Bot**, since such terms should support all operations of all types. These properties do not imply that these types have no use: the type **Top** corresponds to the `Object` type found in most object-oriented languages, while **Bot** is similar to the type of a `null` reference that can be used as a 'dummy value' for any (reference) type. We specifically include both types here because they will serve as convenient type variable bounds in a following section on bounded quantification.

Next, we consider subtyping relations for the original type forms from Figure 6.1. For the base types **Int** and **Bool**, we will not install any specific subtype relations. For function types, a subtype relation is defined recursively (rule SFUN), based on the respective subtype relations of the types of the domain and codomain. Note however that, when changing between supertype and subtype, the codomain evolves in the same direction (it is *covariant*), while the domain evolves in opposite direction (it is *contravariant*). For example, every function type is a subtype of **Bot**`->`**Top**, but not of **Top**`->`**Top**. This can be understood through the intuition that a function type T_1 `->` T_2 guarantees a function that provides *some* result of type T_2 for *any* argument of type T_1. We strengthen this contract (to obtain a subtype S_1 `->` S_2) by accepting *wider* arguments ($T_1 <: S_1$) and providing *narrower* results ($S_2 <: T_2$).

Additionally, without claiming complete rigorous treatment, we can give a more formal justification of this subtyping rule by considering the operations that the type system allows for term typings that can be obtained from this subtyping rule (via the subsumption rule). In this case, the assigned types will be function types, and the only operation admitted for these types

[2] In other languages, some generally available operations may be defined for the top type.

is the function application, which is typed through rule APPL. The complete picture is therefore that the type system might admit a function application $e_0\ e_1$ through the following derivation:

$$\dfrac{\dfrac{A \vdash e_0 : S_1 \to S_2 \qquad \dfrac{T_1 <: S_1 \qquad S_2 <: T_2}{(S_1 \to S_2) <: (T_1 \to T_2)}\ \textsc{SFun}}{A \vdash e_0 : T_1 \to T_2}\ \textsc{Sub} \qquad A \vdash e_1 : T_1}{A \vdash (e_0\ e_1) : T_2}\ \textsc{Appl}$$

However, we observe that it is possible to derive the same function application typing from the same conditions, by employing the subsumption rule only for the conditions of the subtyping rule, namely the relations $T_1 <: S_1$ and $S_2 <: T_2$:

$$\dfrac{\dfrac{A \vdash e_0 : S_1 \to S_2 \qquad \dfrac{A \vdash e_1 : T_1 \qquad T_1 <: S_1}{A \vdash e_1 : S_1}\ \textsc{Sub}}{A \vdash (e_0\ e_1) : S_2}\ \textsc{Appl} \qquad S_2 <: T_2}{A \vdash (e_0\ e_1) : T_2}\ \textsc{Sub}$$

So by using the validity of subsumption for the conditions of the subtyping rule as an hypothesis, we can demonstrate the validity of subsumption for the conclusion of the subtyping rule (where the 'validity' is demonstrated by recreating all operation typings that could be derived from the subtyping rule). Although by no means a rigorous proof, this illustrates how the inductive step for this subtyping rule could be realized in a proof by induction[3] on the derivation of the subtype judgment.

Finally, for record types, rule SRECD stipulates that a record type is a subtype of another record type, if the first has all the fields of the second (and possibly more), and the types of the common fields are respectively in the subtype relation. An argument similar to the one for function types could be made to demonstrate that this subtype rule preserves the validity of its operation, the record projection. The type rule establishes the subtype relations that we discussed in the examples:

$$\dfrac{\dfrac{}{\texttt{Int} <: \texttt{Int}}\ \textsc{SRefl}}{\{\texttt{fst} : \texttt{Int}, \texttt{snd} : \texttt{Bool}\} <: \{\texttt{fst} : \texttt{Int}\}}\ \textsc{SRecd}$$

Together with the subsumption rules, this alleviates the inflexibility that motivated the addition of subtyping.

6.2.3 Parametric Polymorphism

Motivation

On page 84, we have shown how to 'wrap' the addition operation in a function `plus` of type `Int -> Int -> Int`. From a typing perspective, this functionality could be offered as a library

[3] In general, a *proof by structural induction* establishes a property $P(e)$ for all expressions e, by demonstrating for each possible expression form that $P(e)$ can be derived from $P(e_1), \ldots, P(e_n)$, where $e_1, \ldots, e_n$ are the immediate subexpressions of e in that form. This reasoning requires that expressions have a finite, acyclic tree structure, such that we are guaranteed to eventually encounter base cases. We can similarly reason by induction on the structure of a derivation tree.

function instead of a primitive operation. Now, it is somewhat more complicated to do the same for the primitive boolean test. We have to define multiple functions that each represent this test with branches of a different type. All of these functions share the same behavior (they have the same body term) but accept arguments of a different type:

```
let if_int (c : Bool) (x : Int) (y : Int) = if c then x else y
val if_int : Bool -> Int -> Int -> Int

let if_bool (c : Bool) (x : Bool) (y : Bool) = if c then x else y
val if_bool : Bool -> Bool -> Bool -> Bool

let if_top (c : Bool) (x : Top) (y : Top) = if c then x else y
val if_top : Bool -> Top -> Top -> Top

let if_fint (c : Bool) (x : (Int -> Int)) (y : (Int -> Int)) = if c then x else y
val if_fint : Bool -> (Int -> Int) -> (Int -> Int) -> (Int -> Int)
```

Of course, this duplication is unacceptable from a software engineering point of view. Notice that subtype polymorphism does not remedy this problem: while the function `if_top` can be used with arguments of any type, its result will be of the general type **Top**, which does not allow any further operations (and rightfully so, because there is very little checking on the arguments of the branches).

Universal quantification Instead, the solution will be to define one polymorphic function that abstracts over the varying parts of these functions, the types. We will enable such abstractions in a way that is largely parallel to the existing function mechanism. While function abstraction terms abstract terms over other terms (represented by term variables), we will add a new kind of term that abstracts terms over types (represented by type variables). This is written **all** X => e, similar to **fun** x => e. (The keyword **all** is not to be confused with the keyword **forall**, employed below.) And while function abstractions have a function type that contains sufficient information to type a function application, these *type abstractions* will have a new kind of type, called a *universal type*, which provides information to determine the type of a *type application*, which is the application of a type abstraction to a type. The ordinary function application is written by juxtaposing the abstraction and the argument (e_0 e_1), and for the type application a similar juxtaposition is used with rectangular brackets placed around the type argument to mark the difference: e [T]. Note here that although types are applied, there is no implication that types can be manipulated as values: types and terms are still distinct and type abstractions and applications serve type-checking purposes only, with no run-time implications.

We will further integrate these elements in the formal language definition below. Here, we will already continue the example by defining a polymorphic version of the `if` function, which generalizes the above versions by nesting the function in a type abstraction. For top-level bindings, we will use an abbreviation for type abstractions, similar to the one for function abstractions:

```
let if_gen [X] (c : Bool) (x : X) (y : X) = if c then x else y
```

```
val if_gen : forall X, Bool -> X -> X -> X
```

Here, X is a type variable that represents the type parameter of the type abstraction if_gen. if_gen has a universal type, named as such because the type variable is universally quantified (indicated by the keyword **forall** in the syntax of such types). This means simply that we can bind the type variable to any type through a type application. For example, the application if_gen [**Int**] has a type obtained by substituting **Int** for X, yielding **Bool** -> **Int** -> **Int** -> **Int**, while if_gen [**Bool**] has type **Bool** -> **Bool** -> **Bool** -> **Bool**. These terms are equivalent to respectively if_int and if_bool: they have the same type and behavior as those functions.

The interesting characteristic of type variables is that they not only allow to abstract types (**Top** does this as well), they also allow to connect types in different positions because they represent an *unknown but fixed* type. We have used this to connect the domain and codomain types of a function type. It can also be used to connect the types of the fields in a record type: for example, {const : X, op : X -> **Int**} is the type of a record which contains a constant term of a certain type (record const) as well as an operation to produce an integer from that same type (record op). Several concrete records (with different types) fit this description. For example:

```
let int_rcd = {const = 1, op = succ}
val int_rcd : {const : Int, op : Int -> Int}

let bool_rcd = {const = true, op = (fun c : Bool => if c then 1 else 0)}
val bool_rcd : {const : Bool, op : Bool -> Int}
```

We do not need specific information about type X in {const : X, op : X -> **Int**} to make use of such a record. We can define a polymorphic function for all such X to apply its operation field to its constant field:

```
let use [X] (x : {const : X, op : X -> Int}) = x.op x.const
val use : forall X, {const : X, op : X -> Int} -> Int
```

Similar to if_gen, use is a type abstraction of a function term: it can be 'instantiated' with different types to obtain different functions that have different function types. We can use this to obtain two different functions to work with respectively int_rcd and bool_rcd. Both are used in the following term:

```
if c then (use [Int] int_rcd) else (use [Bool] bool_rcd)
```

Note however that, in this case, the function types of use [**Int**] and use [**Bool**] differ only in the type of their argument; the type of the result is constant (**Int**). Since there is no relation between argument type and result type, it would be possible to have a single ordinary use function, with an argument type that is the generalization of the different types with structure {const : X, op : X -> **Int**}. In fact, such a solution would be preferable, since it would allow to move the use application outward in the above term, thus avoiding the duplication:

```
use (if c then int_rcd else bool_rcd)
```

We will therefore add a new kind of type[4], called an *existential type*, that functions as a generalization of the different types with structure {const : X, op : X -> **Int**} for some binding of X.

Existential quantification An existential type is in many ways the dual of a universal type. It consists of a type expression parameterized with a type variable that is existentially quantified, for example, **exists** X, {const : X, op : X -> **Int**}. The type of a term e is generalized to an existential type by means of an explicit *packing* operation, which is written:

$$\textbf{pack}\ e\ \textbf{as}\ T_1\ [X = T_0]$$

The result of this operation will be assigned the existential type **exists** X, T_1, provided that e has the type T_1 where type variable X is bound to (replaced by) T_0. So it is verified that e indeed has type T_1 from some binding of X, namely to T_0 (the type T_0 is sometimes called the *witness type* for this reason). We illustrate packing by continuing the example from above:

```
let pkg1 = pack int_rcd as {const : X, op : X -> Int} [X = Int]
val pkg1 : exists X, {const : X, op : X -> Int}

let pkg2 = pack bool_rcd as {const : X, op : X -> Int} [X = Bool]
val pkg2 : exists X, {const : X, op : X -> Int}
```

We will not directly allow operations on terms of existential type. For example, the direct projection pkg1.const will be prohibited. The reason is that the typing of this operation will involve the variable type X, where it is not clear what is the scope of this type variable. By first using an *opening* (or *unpacking*) operation, the scope can be explicitly controlled. This operation is written:

$$\textbf{open}\ x\ [X] = e_1\ \textbf{in}\ e_0$$

Where e_1 must be a term of an existential type and X is a new type variable in the scope of body e_0. The ordinary variable x can be used, also in the scope e_0, to refer to the term that is 'packed' inside e_1, only is x typed with the abstract type variable X in place of the concrete witness type. For example, in **open** x [X] = pkg1 **in** e_0, the name x can be used in e_0 to refer to int_rcd, only does it now have the type {const : X, op : X -> **Int**} instead of {const : **Int**, op : **Int** -> **Int**}. This will allow to operate on both pkg1 and pkg2 in a flexible and safe manner:

```
open x [X] = (if c then pkg1 else pkg2) in (x.op x.const)
```

This entire term is well-typed and has type **Int**. Since the witness type is available as type variable X, we can obtain the same result using the polymorphic use function from above:

```
open x [X] = (if c then pkg1 else pkg2) in (use [X] x)
```

[4] Strictly speaking, existential types do not warrant a new type form as they can be encoded using universal types and function types (see Pierce, 2002, Sec. 24.3). However, it is common to explicitly support them in the type system.

Remark. While we have introduced existential quantification of type variables as a polymorphism mechanism here, they are traditionally used for typing *abstract data types*, as developed by Mitchell and Plotkin (1988). The type `exists X,{const:X,op:X->Int}` may indeed be interpreted as a simple abstract type, packaged with its set of operations. The variable `X` is the abstract type itself, which hides a concrete representation provided by the witness type (which is also called the *hidden representation type* for this reason). The record `{const:X,op:X->Int}` is the set of operations on that abstract type. Since client code is typed with a type variable instead of the underlying representation type, it has no way of manipulating the abstract data, except passing it to the provided set of operations. As such, the typing can enforce the *principle of representation independence* (although other measures exist to protect programmer-defined abstractions, for example, using function closures).

Bounded quantification So far, we have considered parametric and subtype polymorphism as orthogonal features. One interesting way to mix them is by considering subtype constraints (or *bounds*) in the quantification of type variables. We encounter a motivation for such bounded quantification when typing a function that 'passes an argument around' (which is most flexibly typed with universal quantification), while at the same time operating on it (which will require some information about its type). Consider for example:

```
let bnd_up [X<:{val:Int}] (x:X)={fst=succ x.val,snd=x}
val bnd_up : forall X<:{val:Int},X->{fst:Int,snd:X}
```

In the type abstraction, we have quantified the type variable `X` as "any type which is a subtype of `{val:Int}`". This upper bound is required to operate on variable `x` of type `X` when constructing the first field of the result; this function would not admit a typing if `X` were quantified without the upper bound. And while it *is* possible to type this function with `x` declared to have a non-variable type (such as `{val:Int}`), we must construct different versions to account for the fact that the type of the second record in the result is guaranteed to be equal to the argument type. The type abstraction with a bounded type variable can be seen as a generalization of these different versions. Obviously, such a type abstraction can only be applied to a type that meets the upper bound, *e.g.*, the type application `bnd_up [Bool]` will be rejected.

Similarly, supertype constraints for type variables can be considered as well. Although this is less frequently considered in literature and not supported in any real programming language we are aware of, we motivate here that this naturally complements the case of subtype constraints. While a subtype constraint allows to include a term of a variable type into an existing type (the upper bound), a supertype constraint allows to include a term of an existing type (the lower bound) into the variable type. This provides a way to introduce new values of this variable type. For example, consider a function that will return either its argument or some fixed value, depending on some condition:

```
let bnd_lo [X:>{fst:Int,snd:Bool}](x:X)=if c then {fst=1,snd=true} else x
val bnd_lo : forall X:>{fst:Int,snd:Bool},X->X
```

Here, the lower bound is crucial to be able to assign the same type (`X`) to the two branches of the boolean test. This example can again be interpreted as a generalization of different function with non-variable type, such as `{fst:Int,snd:Bool}->{fst:Int,snd:Bool}`.

Subtype and supertype constraints are similarly useful for existential quantification, where they allow to construct 'less general' generalizations of types by only partially abstracting types in the type expression structure. We will see in Section 6.3.2 that such existential types form the underpinning for a form of more flexible parametric object types, known as wildcards in JAVA.

We also note that all universal and existential types that we consider can be united in two general forms that include both a lower bound type (S) and an upper bound type (U), written

$$\textbf{forall}\ X\ \textbf{in}\ S..U\,,\,T \qquad \text{and} \qquad \textbf{exists}\ X\ \textbf{in}\ S..U\,,\,T$$

The previous forms of quantification that were unbounded in upward or downward direction (or both) is equivalent to respectively the usage of type **Top** as upper bound or type **Bot** as lower bound (or both). However, double bounded quantification allows some extra terms to be typed, for example:

```
let bnd_in [X in {fst : Int, snd : Bool}..{fst : Int}] (x : X) =
      if c then {fst = succ x.fst, snd = true} else x
val bnd_in : forall X in {fst : Int, snd : Bool}..{fst : Int}, X -> X
```

Formal Definition

The formal definition of the language syntax and the term typing rules is extended in Figure 6.3 to account for the forms of parametric polymorphism we have discussed in this section. The inclusion of variable types in type expressions is a change that warrants some extra infrastructure in our type system as the meaning of a type expression now depends on the context where it appears. The typing context information is extended to include type variables and their declared range (similar to how we track term variables and their declared type). Based on this context information, we will first have to determine if a type expression containing type variables even denotes some sensible type. This judgment is called *type well-formedness*, written $A \vdash T$ ok, to indicate "type expression T is well-formed under context assumptions A". Next, the subtype relation has also become dependent on the type variable declarations from the context, and the old subtype judgment ($S <: T$) is therefore replaced by a new one ($A \vdash S <: T$) that takes this information into account (and the subsumption rule is accordingly adapted). Both judgments are defined in Figure 6.4.

Type well-formedness Basically, we will consider a type expression well-formed if it does not contain free type variables. The ground types such as **Bool**, **Int**, *etc.* will be well-formed in any context, while function types and record types are well-formed if their constituent types are well-formed. (In rule OKRECD, note that we write condition $A \vdash T_1, \ldots, T_n$ ok, to abbreviate n separate well-formedness conditions $A \vdash T_1$ ok, ..., $A \vdash T_n$ ok.) A type denoted by a type variable is well-formed within the scope of that type variable (rule OKVAR). Both a universal and existential type bind a new type variable in the contained type expression (second condition of rule OKALL and rule OKEX); we will also require the bounds of this type variable to be well-formed (first condition).

The subtyping judgment has been set-up in such a way that it will only admit a subtype relation between well-formed types. Similarly, the typing judgment for term expressions has been set-up to only assign well-formed types to terms. (In both cases with the explicit provision

Lexical metavariables

$\ldots$	*(previous vars)*
X, Y	*type variables*

Term expressions

e	::=	$\ldots$	*(previous forms)*
	|	$\texttt{all}\ X\ \texttt{in}\ S - U\ \texttt{=>}\ e$	*type abstraction*
	|	$e\ [\,T\,]$	*type application*
	|	$\texttt{pack}\ e\ \texttt{as}\ T_1\ [\,X\ \texttt{in}\ S - U = T_0\,]$	*packing*
	|	$\texttt{open}\ x\ [\,X\,] = e_1\ \texttt{in}\ e_0$	*opening*

Type expressions

S, T, U	::=	$\ldots \mid \texttt{Top} \mid \texttt{Bot}$	*(previous forms)*
	|	X	*variable type*
	|	$\texttt{forall}\ X\ \texttt{in}\ S - U\ \texttt{,}\ T$	*universal type*
	|	$\texttt{exists}\ X\ \texttt{in}\ S - U\ \texttt{,}\ T$	*existential type*

Typing context (assumptions)

A	::=	$\ldots$	*(previous forms)*
	|	$A\texttt{;}\ X\ \texttt{in}\ S - U$	*type variable range*

Term expression typing $\boxed{A \vdash e : T}$

$$\ldots \quad \frac{A \vdash T_1\ \text{ok} \qquad A\texttt{;}\ x\texttt{ :}\ T_1 \vdash e : T_2}{A \vdash (\texttt{fun}\ x\texttt{ :}\ T_1\ \texttt{=>}\ e) : T_1\ \texttt{->}\ T_2}\ \textsc{Abs} \qquad \frac{A \vdash e : S \qquad A \vdash S <: T}{A \vdash e : T}\ \textsc{Sub}$$

$$\frac{A \vdash S, U\ \text{ok} \qquad A\texttt{;}\ X\ \texttt{in}\ S - U \vdash e : T}{A \vdash (\texttt{all}\ X\ \texttt{in}\ S - U\ \texttt{=>}\ e) : \texttt{forall}\ X\ \texttt{in}\ S - U\ \texttt{,}\ T}\ \textsc{TAbs}$$

$$\frac{A \vdash e : \texttt{forall}\ X\ \texttt{in}\ S - U\ \texttt{,}\ T_1 \qquad A \vdash S <: T_0 <: U}{A \vdash e\ [\,T_0\,] : [X \mapsto T_0]T_1}\ \textsc{TApp}$$

$$\frac{A \vdash e : [X \mapsto T_0]T_1 \qquad A \vdash S <: T_0 <: U}{A \vdash (\texttt{pack}\ e\ \texttt{as}\ T_1\ [\,X\ \texttt{in}\ S - U = T_0\,]) : \texttt{exists}\ X\ \texttt{in}\ S - U\ \texttt{,}\ T_1}\ \textsc{Pack}$$

$$\frac{A \vdash e_1 : \texttt{exists}\ X\ \texttt{in}\ S - U\ \texttt{,}\ T_1 \qquad A\texttt{;}\ X\ \texttt{in}\ S - U\texttt{;}\ x\texttt{ :}\ T_1 \vdash e_0 : T_0 \qquad A \vdash T_0\ \text{ok}}{A \vdash (\texttt{open}\ x\ [\,X\,] = e_1\ \texttt{in}\ e_0) : T_0}\ \textsc{Open}$$

Figure 6.3: Extension of the language of Figure 6.1 with parametric polymorphism. The auxiliary judgments regarding subtyping and well-formed types are presented in Figure 6.4. These definitions supersede the ones from Figure 6.2 since they include a new subtype judgment that integrates context information. The rule ABS replaces the old rule by the same name.

Well-formed types $\boxed{A \vdash T \text{ ok}}$

$$\textsc{OkBool}\ \frac{}{A \vdash \mathbf{Bool}\ \text{ok}} \qquad \textsc{OkInt}\ \frac{}{A \vdash \mathbf{Int}\ \text{ok}} \qquad \textsc{OkTop}\ \frac{}{A \vdash \mathbf{Top}\ \text{ok}} \qquad \textsc{OkBot}\ \frac{}{A \vdash \mathbf{Bot}\ \text{ok}} \qquad \textsc{OkVar}\ \frac{}{\ldots ; X\ \mathbf{in}\ S - U ; \ldots \vdash X\ \text{ok}}$$

$$\textsc{OkFun}\ \frac{A \vdash T_1\ \text{ok} \qquad A \vdash T_2\ \text{ok}}{A \vdash (T_1 \to T_2)\ \text{ok}} \qquad \textsc{OkRecd}\ \frac{A \vdash T_1, \ldots, T_n\ \text{ok}}{A \vdash \{I_1 : T_1, \ldots, I_n : T_n\}\ \text{ok}}$$

$$\textsc{OkAll}\ \frac{A \vdash S, U\ \text{ok} \qquad A ; X\ \mathbf{in}\ S - U \vdash T\ \text{ok}}{A \vdash (\mathbf{forall}\ X\ \mathbf{in}\ S - U, T)\ \text{ok}} \qquad \textsc{OkEx}\ \frac{A \vdash S, U\ \text{ok} \qquad A ; X\ \mathbf{in}\ S - U \vdash T\ \text{ok}}{A \vdash (\mathbf{exists}\ X\ \mathbf{in}\ S - U, T)\ \text{ok}}$$

Subtyping $\boxed{A \vdash S <: T}$

$$\textsc{SRefl}\ \frac{A \vdash T\ \text{ok}}{A \vdash T <: T} \qquad \textsc{STrans}\ \frac{A \vdash S <: T \qquad A \vdash T <: U}{A \vdash S <: U} \qquad \textsc{STop}\ \frac{A \vdash T\ \text{ok}}{A \vdash T <: \mathbf{Top}} \qquad \textsc{SBot}\ \frac{A \vdash T\ \text{ok}}{A \vdash \mathbf{Bot} <: T}$$

$$\textsc{SFun}\ \frac{A \vdash T_1 <: S_1 \qquad A \vdash S_2 <: T_2}{A \vdash (S_1 \to S_2) <: (T_1 \to T_2)}$$

$$\textsc{SRecd}\ \frac{A \vdash S_1 <: T_1 \qquad \cdots \qquad A \vdash S_n <: T_n \qquad A \vdash S_{n+1}, \ldots, S_m\ \text{ok}}{A \vdash \{I_1 : S_1, \ldots, I_n : S_n, \ldots, I_m : S_m\} <: \{I_1 : T_1, \ldots, I_n : T_n\}}$$

$$\textsc{SVarU}\ \frac{}{\ldots ; X\ \mathbf{in}\ S - U ; \ldots \vdash X <: U} \qquad \textsc{SAll}\ \frac{A \vdash S_1 <: T_1 \qquad A \vdash T_2 <: S_2 \qquad A ; X\ \mathbf{in}\ T_1 - T_2 \vdash S_0 <: T_0}{A \vdash \mathbf{forall}\ X\ \mathbf{in}\ S_1 - S_2, S_0 <: \mathbf{forall}\ X\ \mathbf{in}\ T_1 - T_2, T_0}$$

$$\textsc{SVarS}\ \frac{}{\ldots ; X\ \mathbf{in}\ S - U ; \ldots \vdash S <: X} \qquad \textsc{SEx}\ \frac{A \vdash T_1 <: S_1 \qquad A \vdash S_2 <: T_2 \qquad A ; X\ \mathbf{in}\ S_1 - S_2 \vdash S_0 <: T_0}{A \vdash \mathbf{exists}\ X\ \mathbf{in}\ S_1 - S_2, S_0 <: \mathbf{exists}\ X\ \mathbf{in}\ T_1 - T_2, T_0}$$

Figure 6.4: Definition of the judgments regarding well-formed types and subtyping

that context for the judgment only contains well-formed types, of course.) To this end, a well-formedness condition has been added to some of the rules that we have previously encountered. For subtyping, this is the case for rule SREFL, rule STOP and rule SBOT. For term expression typing, this is the case for rule ABS. In case of the other rules, the well-formedness follows from existing subtype or typing conditions, it is not necessary to demand it explicitly.

New typing rules The typing rule for type abstractions (rule TABS) is entirely analogous to the one for ordinary function abstraction (rule ABS). The body of the abstraction is checked in a context that is extended with the new type variable and its declared bounds. As we did for ordinary function abstraction in Section 6.2.1, we silently require that the name for the new type variable be chosen distinct from the names of the type variables that already exist in the context, to avoid variable capture. The type of a type abstraction (a universal type) contains sufficient information for the typing of type applications (rule TAPP). This rule requires that the type argument T_0 meets the prescribed bounds, written $A \vdash S <: T_0 <: U$ as an abbreviation for two conditions $A \vdash S <: T_0$ and $A \vdash T_0 <: U$. The type assigned to the type application will be the type expression contained in the universal type (this is type expression T_1), where type variable X is bound to the type argument T_0. This binding is effectuated by means of a *type substitution*: $[X \mapsto T_0]T_1$ is the notation for "the type expression obtained by substituting T_0 for X in type expression T_1".

To understand rule PACK, recall that the packing operation will generalize the type of term expression e to a type expression T_1 parameterized with type variable X; this type variable takes the place of some witness type T_0. The typing rule will therefore require that e admits type T_1 where X is bound to T_0 (this binding is similarly effectuated with a type substitution). It is also verified that T_0 meets the declared bounds S and U. The opening operation (rule OPEN) allows to employ a term e_1 of some existential type with type expression T_1 as a variable x of that type T_1. In the second condition, the body e_0 is therefore typed with a context that is extended with the type variable X (declared to have the appropriate bounds) and this variable x. The type of the body (type T_0) is used as the type of the entire open operation in the conclusion. The third condition will however require that type expression T_0 is well-formed in the original context of the open operation. We have to check this explicitly to prevent that type variable X appears in this type; in the original context this type variable has no meaning (*i.e.*, the type variable would escape its scope).

Example. Employing the following abbreviation of a set of context assumptions:

$$A_0 \stackrel{\text{def}}{=} \texttt{X}\ \textbf{in}\ \textbf{Bot}\ \texttt{-}\ \textbf{Top}\ \texttt{;}\ \texttt{x}\ \texttt{:}\ \{\texttt{const}\ \texttt{:}\ \texttt{X}\ \texttt{,}\ \texttt{op}\ \texttt{:}\ \texttt{X}\ \texttt{->}\ \textbf{Int}\}$$

We can derive the following term expression typing using the typing rules from Figure 6.1:

$$\dfrac{\dfrac{\dfrac{}{A_0 \vdash \texttt{x}\ \texttt{:}\ \{\texttt{const}\ \texttt{:}\ \texttt{X}\ \texttt{,}\ \texttt{op}\ \texttt{:}\ \texttt{X}\ \texttt{->}\ \textbf{Int}\}}\ \textsc{Var}}{A_0 \vdash \texttt{x.op}\ \texttt{:}\ \texttt{X}\ \texttt{->}\ \textbf{Int} \qquad A_0 \vdash \texttt{x.const}\ \texttt{:}\ \texttt{X}}\ \textsc{Proj},\textsc{Proj}}{A_0 \vdash (\texttt{x.op}\ \texttt{x.const})\ \texttt{:}\ \textbf{Int}}\ \textsc{Appl}$$

This term can therefore be used in the body of an opening operation on top-level variable pkg1

from the motivating examples:

$$\frac{\vdash \texttt{pkg1} : \textbf{exists}\ \texttt{X}\ \textbf{in}\ \textbf{Bot}\ \texttt{-}\ \textbf{Top}\texttt{,}\ \{\texttt{const : X, op : X -> Int}\} \qquad A_0 \vdash (\texttt{x.op x.const}) : \textbf{Int} \qquad \vdash \textbf{Int}\ \text{ok}}{\vdash \textbf{open}\ \texttt{x}\ [\texttt{X}] = \texttt{pkg1}\ \textbf{in}\ (\texttt{x.op x.const}) : \textbf{Int}}\ \textsc{Open}$$

Here, the well-formedness condition ($\vdash$ **Int** ok) can be trivially established using rule OkInt. Note that while we have also derived the typing $A_0 \vdash$ `x.const` : X, a corresponding opening operation with the term `x.const` as the body could not have type X, since we cannot establish the condition $\vdash$ X ok. The type variable X would escape its scope if it were used as the type of the opening operation. A workaround could be to use subsumption to derive another type for the body term:

$$\frac{A_0 \vdash \texttt{x.const} : \texttt{X} \qquad \dfrac{\dfrac{}{A_0 \vdash \texttt{X}\ \text{ok}}\ \textsc{OkVar}}{A_0 \vdash \texttt{X} <: \textbf{Top}}\ \textsc{STop}}{A_0 \vdash \texttt{x.const} : \textbf{Top}}\ \textsc{Sub}$$

There is no problem to establish that X is well-formed in A_0 (alternatively, rule SVarU could have been used to establish the subtype relation). And since **Top** is well-formed in any context, we can use this type as the result of the opening operation.

New subtyping rules The first 6 rules of the new subtype judgment are simply updated versions of the rules from Figure 6.2, modified to include a typing context A (and to carry it over between conditions and conclusion) and to demand a well-formed type where this is not guaranteed by the other conditions of the rule. We also define 4 new subtyping rules to define subtyping for the new kinds of types that have been added. For variable types, two subtype relations are considered. In rule SVarU, a variable type is defined to be a subtype of the upper bound U that was declared for the type variable in the context, while rule SVarS defines it to be a supertype of the lower bound S. These rules are the equivalent of rule Var for term variable type assumptions.

The combination of these two new subtyping rules entails (by transitivity) that the lower bound is a subtype of the upper bound. This may lead to suspicious conclusions for certain (equally suspicious!) context assumptions. For example, we can derive the conclusion **Int** <: **Bool** from an assumption such as X **in Int** – **Bool**. It is of course unsound to include integers in booleans, but note there is no concrete type T to which X could be bound under those restrictions, *i.e.*, there is no T for which we can obtain $\vdash$ **Int** <: T <: **Bool**. So the conclusions derived from the assumption that there *is* such a T have no effect in a context without that (or an equivalent) assumption. More concretely, while we can type certain type abstractions with a type of the form **forall** X **in Int** – **Bool**, T (and employ the relation **Int** <: **Bool** when typing the body of that abstraction, see rule TAbs), type applications of those abstractions will not admit a typing, because, save for other assumptions, there is no type argument that meets both of the bounds (second condition of rule TApp). And dually, while it would be possible to open a term of existential type **exists** X **in Int** – **Bool**, T (and employ the relation **Int** <: **Bool** when typing the body of that operation, see rule Open), we cannot pack a term to have that existential type since there is no witness type that meets these bounds (second condition of rule Pack).

Finally, we also include a subtyping rule for comparing universal types (rule SALL), as well as one for comparing existential types (rule SEX)[5]. To intuitively understand rule SALL through the principle of safe substitution, recall that the universal type **forall** X **in** T_1 – T_2 , T_0 describes a collection of type abstractions, each of which can be instantiated with a type between T_1 and T_2, to obtain an instance of T_0. Similar to function abstractions, such a type abstraction may safely be replaced with an type abstraction with a larger domain and a smaller codomain. The larger domain is ensured by only lowering the lower bound and raising the upper bound (first and second conditions). Ensuring a smaller codomain is slightly more involved, as with a type application, the result type depends on the argument type (the result type will be the instance of T_0 obtained by binding X to the argument type). We require a smaller codomain, no matter what is the argument: for all types U that are acceptable argument types for the original type abstraction (so that are between T_1 and T_2), the U-instance of the new return type S_0 should be a subtype of the U-instance of the original return type T_0. This is enforced by generalizing all types U as the variable type X with the assumption X **in** T_1 – T_2 (third condition).

We can also explain the rule rule SEX using the principle of safe substitution, although in the other direction: recall that the existential type **exists** X **in** S_1 – S_2 , S_0 is obtained by packing a term with a type that is the U-instance of S_0, where this witness type U is between S_1 and S_2. We can equally pack this term to another existential type **exists** X **in** T_1 – T_2 , T_0, if the witness type U is guaranteed to lie within T_1 and T_2, and if the term is also of a type that is the U-instance of T_0. The first requirement is ensured by demanding that bounds T_1 and T_2 are wider than bounds S_1 and S_2 (first and second condition). For the second requirement, we demand that the U-instance of S_0 is a subtype of the U-instance of T_0, for every possible witness type U between S_1 and S_2. This translates to the third condition.

The subtyping rule for existential types will explain the flexibility of the wildcard mechanism in Section 6.3.2. The rule for covariant overriding of generic methods is a restricted form of the subtyping rule for universal types.

6.3 Java 5 Generics: Parametric Polymorphism for Objects

Being a typed class-based object-oriented programming language, JAVA has supported subtype polymorphism since its initial conception. The generics feature of JAVA 5 (Gosling et al., 2005) adds comprehensive support for parametric polymorphism and pioneers new mechanisms for combining it with object-oriented subtyping. Note that contrary to C++ templates, which are essentially a form of *macros* that must be expanded and interpreted over and over for each instantiation, the generic mechanisms in JAVA 5 are first-class language features.

The general design goal of generics is to allow a high amount of flexibility while ensuring both forwards and backwards interoperability (*i.e.*, JAVA 5 code should be able to interface pre-JAVA 5 code and vice versa). In the next two sections, we discuss respectively the GJ and wildcard proposals that contributed the major elements of the generics design. We then conclude with a short discussion of the opportunities offered by generics to the designers of frameworks. A

[5]The rules included here correspond to ones for the "full" variant of $F_{<:}$, where subtyping is theoretically undecidable (see Pierce, 2002, Sec. 28.5). Discussion of the practical implications of this result is beyond the scope of this introduction.

comprehensive discussion of JAVA 5 Generics is available in the textbook by Naftalin and Wadler (2006).

6.3.1 Generic Java: Invariant Type Parameters

The basis for JAVA 5 Generics is the GJ proposal by Bracha, Odersky, Stoutamire, and Wadler (1998). GJ extends the original JAVA with generic types and methods. A GJ program is implemented by *erasing* all type parameters, mapping typing variables to their bounds and inserting casts where needed (the result closely mimics the way generics were previously emulated in the unextended language).

Generic methods

As a first mechanism, GJ allows to declare type variables at the level of methods, possibly with an upper bound, for example:

```
<X extends Number> X poly(X x) {
    System.out.println(x.intValue());
    return x;
}
```

This method declares a type variable `X` to denote a variable type with upper bound `Number`. Both the argument and the result of the method are declared to have this variable type.

A method whose declaration includes type variables is called a *generic method*. This construct corresponds rather directly to the bounded universal quantification we have discussed for type abstractions (with the important difference that methods are not first-class values). The functional counterpart of this method would have the following universal type:

```
poly : forall X in Null - Number , X -> X
```

(Here, the *null type* `Null` is the JAVA equivalent of the bottom type; it is the type of the special reference **`null`** and a subtype of every reference type.) Note that inside this function, as well as inside the above method, a value of type `X` may be treated as a `Number`, due to upper bound that is declared for type variable `X`.

When invoking the generic method, type arguments can be given explicitly (this corresponds to type application), for example `<Integer>poly(5)`. If type arguments are omitted, the compiler will infer them from the type of the arguments, so we can write the call as simply `poly(5)`. When typing these invocations, the method type will have `Integer` substituted for `X`, so the invocation will have result type `Integer`.

Inside their scope, type variables can be used as type annotations, but new objects of a type denoted by a type variable cannot be created. There are two reasons for this: first, the type variable could be bound to an abstract or interface type, types which cannot be instantiated; second, as mentioned above, the implementation of generic entities erases the type variables, so the type values are not known at run-time.

Generic classes and interfaces

Type variables can also be defined at the level of class or interface declarations (*resp.* called *generic class* and *generic interface* declarations). The scope of the type variable extends over all members of the declaration. For example, we can define a small container class:

```
class Container<X extends Number> {
    X val;
    Container(X val) { set(val); }
    X get() { return val; }
    void set(X x) { val = x; }
    int intValue() { return val.intValue(); }
}
```

Generic classes and interfaces are collectively called *generic types*. Generic types such as `Container` cannot be directly used as types. The appropriate number of type parameters needs to be provided, and the type parameters must respect the bounds. `Container<Float>` is a valid parameterized class type; instances of this class can be created (since `Container` is a concrete class) and the parameterized type offers all the members of `Container`, where `X` is substituted by `Float`. We demonstrate this in the following example, where we assume that `doubleFloat` is the name of a method that takes a `Float` argument and provides a `Float` result:

```
Container<Float> c = new Container<Float>(5.0);
c.set(doubleFloat(c.get()));
```

This is a big advantage over the pre-JAVA 5 version of `Container` (where every `X` would be replaced by `Number`). As the result of the `get` method is already a `Float`, it does not need to be cast from `Number` to `Float`. And since the argument of `set` is verified to be a `Float`, we cannot erroneously pass in another kind of `Number`.

Within its scope, a type variable may be used as a type annotation, but also as a type bound for another type variable or as a type parameter in a parameterized type. Both cases are demonstrated in the following example.

```
class CopyContainer<X extends Number> extends Container<X> {
    <Y extends X> void copyFrom(Container<Y> c) {
        set(c.get());
    }
}
```

Here, the generic method `copyFrom` of parameterized class `CopyContainer<Float>` will accept any `Container` parameterized with a subtype of `Float`, *i.e.*, any container whose value can always be copied. Also note from the example that the scope of a class type variable includes the declaration of the parent class. However, a parent class may not be specified as a type variable, for much the same reasons why a type variable cannot be instantiated.

Interestingly, the scope of a type variable also includes its own bound, as well as the bounds of other type variables that are declared at the same time (although a cycle is prohibited by

restricting such a *forward reference* to a type variable from being used directly as a bound). This is different from the functional language we have described in Section 6.2.3. For example, the following type variable declarations are supported:

```
<X extends Comparable<X>>
<X extends Comparable<Y>, Y extends Comparable<X>>
```

This feature was termed *F-bounded polymorphism* by Canning et al. (1989). It can be used to assert certain operations of the type with respect to itself. For example, we can select the greatest of two values of any type, as long as that type is comparable to itself:

```
interface Comparable<E> {
    int compareTo(E that);
}
<E extends Comparable<E>> E max(E x, E y) {
    return x.compareTo(y) < 0 ? y : x;
}
```

We can invoke the generic `max` method with two `Byte` values since `Byte` implements the interface `Comparable<Byte>` (*i.e.*, a byte is comparable to itself).

Invariant Subtyping

The parameterized type `CopyContainer<Float>` is a subtype of `Container<Float>`. Indeed, we obtain this parent if we substitute `Float` for `X` in the parent declaration of the generic class `CopyContainer`, and `CopyContainer<Float>` inherits all members of this parent.

But what about comparing parameterized types based on their type arguments: for example, what is the relation between `Container<Float>` and `Container<Number>`? Intuitively, we might say a floating-point number container *is a* number container, but this interpretation is too simplistic: we remark that `Container<Float>` does not satisfy the contract of `Container<Number>` (as the `set` method of the first will not accept any `Number`), nor the other way around (as the `get` method of the latter is not guaranteed to provide a `Float`). For that reason, there is no safe substitution between values of these types.

To illustrate this further, we may interpret the interfaces of instances of these two parameterized types as the following two record types[6]:

```
{get : {} -> Float, set : Float -> {}}
{get : {} -> Number, set : Number -> {}}
```

(We note that this interpretation is not a complete modeling of the features of objects: for example, the implicit argument which represents the receiver has been omitted. We only employ this crude simplification to illustrate the connection to the functional language from the beginning of this chapter.)

[6] We employ the empty record type `{}` to represent **`void`** or the case of no arguments. This is a common generalization in functional programming languages: type `{}` is the *unit type*, its only inhabitant is the empty record value `{}`. An argument or result of this type is entirely fixed and therefore conveys equally little information as no argument or result.

There is no subtype relation (in either direction) between the above two record types. While for the `get` field we have `{}->Float` <: `{}->Number`, we have a relation in the opposite direction for the `set` field: `Number->{}` <: `Float->{}`. Because rule SRECD requires a subtype relation in the same direction for all common fields, neither of these two types is a subtype of the other.

Accordingly, types `Container<Float>` and `Container<Number>` have no subtype relation in GJ. In general, two parameterized types that stem from the same generic type are not in relation unless their type parameters are pointwise equal. Since the subtype relations of the type parameters is not considered to determine the subtype relation of parameterized types, we say the type parameters in GJ behave invariantly with respect to subtyping.

6.3.2 Wildcards: Use-site Variant Type Parameters

Nevertheless, despite the soundness argument, the invariance of generic types proves to be quite a strong restriction in practice, and several proposals safely relax this requirement.

Declaration-site Variance

A first straightforward observation is that a generic class or interface may only offer members with a restricted structure, which allows to install a subtype relation that is variant with respect to the type arguments. More precisely, a generic type $C\langle X\rangle$ is said to be *covariant* with respect to X, if $S <: T$ allows to safely conclude $C\langle S\rangle <: C\langle T\rangle$. This is the case if X is only used in read-only positions in the class or interface declaration: as method return type or as the type of an unassignable instance variable. For example, for the following read-only container, `SourceContainer<Float>` will satisfy the interface of `SourceContainer<Number>`, and a subtype relation could therefore safely be installed between the two parameterized types:

```
class SourceContainer<X extends Number> {
    final X val;
    SourceContainer(X val) { this.val = val; }
    X get() { return val; }
}
```

Oppositely, $C\langle X\rangle$ is said to be *contravariant* with respect to X if $S <: T$ allows to safely conclude $C\langle T\rangle <: C\langle S\rangle$; this corresponds to the restriction of X appearing only in write-only positions. The combination of covariant and contravariant is called *bivariant*, and requires that X does not appear in the interface of the generic type at all.

Based on these observations, *declaration-site variance* is proposed in approaches by America and van der Linden (1990); Bracha and Griswold (1993). This technique consists of attaching *variance annotations* to the type variables in generic type declarations, to indicate these kind of restrictions. The usage of the type variables in the class body is verified to match explicitly given annotations, or, alternatively, the annotations may be implicit and derived from the usage in the body. In either case, this approach requires great care from class designers, choosing between a rich set of subtypes thanks to variance or a rich functionality because of unrestricted usage of type variables.

Use-site Variance

Following an idea by Thorup and Torgersen (1999), Igarashi and Viroli (2002) developed a proposal for *use-site variance* in GJ. This technique allows to defer the decision about which variance is desirable until a generic type is employed, rather than when it is declared. This proposal was redesigned and further extended by Torgersen, Hansen, Ernst, von der Ahé, Bracha, and Gafter (2004) to obtain the *wildcard* mechanism as it exists in JAVA 5.

The basic idea of use-site variance is to include the variance annotation in the parameterized type, rather than in the declaration of the generic type. An example of such a *variant parameterized type* is `Container<+Number>`, where the + marks a covariant type parameter. Variant parameterized types can be considered 'automatic' equivalents of types such as `SourceContainer<Number>`: they restrict access to certain members in exchange for covariant or contravariant subtyping relations. For example, the subtypes of `Container<+Number>` will now include `Container<Number>`, `Container<Integer>`, `Container<Float>`, ... However, instead of defining this new type with a restricted interface and then determining its set of safe subtypes, it is also helpful to reason in the opposite direction: consider a type that is defined to include all (invariant) parameterized types of the form `Container<`*S*`>`, where *S* is some subtype of `Number`, what interface can this type safely allow?

JAVA 5 employs the latter reasoning: by using (bounded) wildcards as type arguments, we can introduce types such as `Container<? extends Number>`, which corresponds to this interpretation of "a container of some type that is a subtype of `Number`". Although this type generalizes several container types with a different parameter type, the parameter type will in any case be a subtype of `Number`. So when *obtaining* a value of this parameter type through the `get` method, we know it will at least be a `Number`. Oppositely, this upper bound has no effect when we have to *provide* a value of this parameter type for the `set` method: only the **`null`** reference is guaranteed to work, simply because it is included in any reference type. To summarize the access restrictions, the type `Container<? extends Number>` appears to have a `get` method with return type `Number` and a `set` method with argument type `Null`.

Figure 6.5 depicts the result of repeating this exercise for other wildcard parametrization of the generic type `Container` as a UML diagram. Besides wildcards with an upper bound, we also use unbounded wildcards, written simply `?`, or wildcard type arguments with a lower bound, such as `? super Number`. Note that in the cases where the wildcard does not have an upper bound, we would normally conclude that only an `Object` result is guaranteed for the `get` operation. However, because the generic type `Container` is defined with upper bound `Number` for the type parameter on page 99, this can be refined to return type `Number`. For other generic types, where the declaration does not include bounds on the possible values of the type variables, this relaxation is not possible.

Besides the apparent member signatures, Figure 6.5 also demonstrates the direct subtype relations between the different parameterized types. Two parametrizations of the same generic class are defined (by Gosling et al., 2005, §4.10.2) to be in a subtype relation if the type arguments of the first *contain* the respective type arguments of the second (where containment corresponds to set inclusion for the sets of types denoted by invariant or wildcard type arguments). This subtype relation corresponds to our informal intuition about parameterized types with wildcard type arguments. For example, "a container of some subtype of `Float`" will also be "a container of some subtype of `Number`" (because any subtype of `Float` is also a subtype of

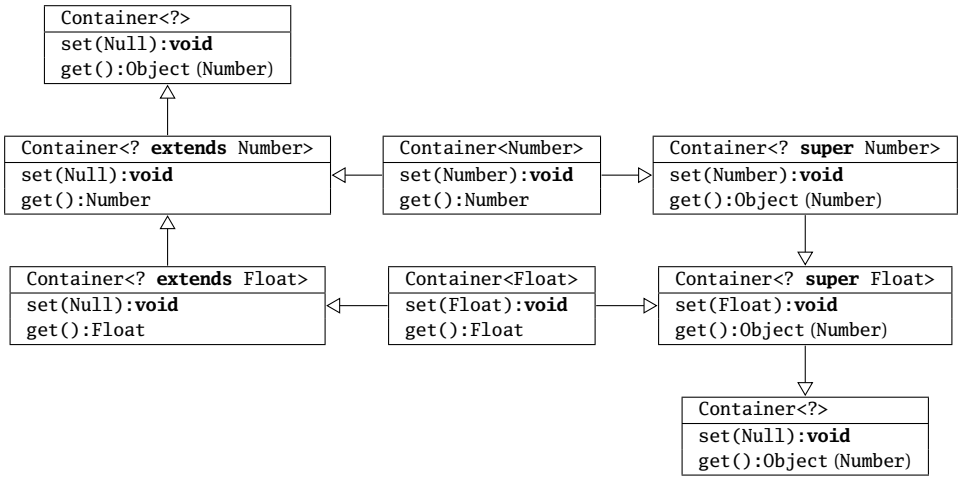


Figure 6.5: Apparent member signatures and subtype relations for wildcard type arguments

`Number`). Alternatively, we can verify that the apparent signatures of the members of a subtype indeed constitute a stronger contract than the corresponding signatures in the supertype.

Irrespective of the justification, it is clear from this diagram that the parametrization with wildcard type arguments enables a very flexible subtyping scheme for generic types. While there is still no direct relation between `Container<Number>` and `Container<Float>`, the similarity of their `get` operation can be exploited through the common supertype:

```
Container<? extends Number>
```

while the similarity of `set` is captured in supertype:

```
Container<? super Float>
```

Wildcards also offer a simpler alternative for certain uses of generic methods that could be considered overkill. For example, in the `CopyContainer` example on page 99 the type variable of generic method `copyFrom` is only used to allow variation of the type argument of one type annotation (it is not used to connect different type annotations). Using wildcards, this method can be defined as an ordinary (non-generic) method with an argument type that contains a wildcard type argument (which is arguably a more straightforward signature):

```
class CopyContainer<X extends Number> extends Container<X> {
    void copyFrom(Container<? extends X> c) {
        set(c.get());
    }
}
```

Correspondence to Existential Types

We have already encountered a type generalization similar to `Container<? extends Number>` in Section 6.2.3; the description of this wildcard parameterized type corresponds to the bounded existential type

$$\texttt{exists X in Null - Number, Container<X>}$$

where `Container<X>` is an abbreviation for the record type `{get : {} -> X, set : X -> {}}`. The existentially quantified type variable is used to hide information about the specific type argument in parameterized types such as `Container<Number>` and `Container<Float>`. `Number` and `Float` are the respective witness types for these two cases.

Igarashi and Viroli, as well as Torgersen et al., exploit this correspondence to existential types in the type systems with use-site variance that they propose. Chiefly, the access restrictions for types with wildcard type arguments are enforced by enclosing all the operations on such types in an implicit opening operation, and using a typing rule similar to that of rule OPEN from Figure 6.3. For example, an operation on variable x of type `Container<? extends Number>` is type checked using a context where the variant type parameter is captured using a fresh type variable X, *i.e.*, using the context assumptions:

$$\texttt{X in Null - Number; x : Container<X>}$$

For this context, the standard typing rules will allow to conclude that the result of calling method `get` on x (or its functional counterpart `x.get {}`) will have type X, or by subsumption, `Number`. Oppositely, we cannot type the invocation of the `set` operation with a `Number` argument (or the counterpart, `x.set` *e* where *e* is some term of type `Number`).

So we observe that we can recreate our conclusions from above, but this mechanism also allows to decide more complex cases. For example, consider the invocation of method `copyFrom` on a variable of type `CopyContainer<? super Number>`. We can also type the operations on nested parameterized types, such as `List<? super List<? extends Number>>`, through the correspondence with nested existential types:

$$\texttt{exists X in (exists Y in Number - Object, List<Y>) - Object, List<X>}$$

All of these cases are uniformly treated by the standard typing rules and a relatively simple opening operation.

As explained by Torgersen et al. (2005, Sec. 3.1), the opening operation for JAVA 5 wildcards occurs through so-called *capture conversion* (defined in Gosling et al., 2005, §5.1.10), which implicitly converts a type parameterized with wildcards into a type that is parameterized with fresh type variables that are defined with appropriate bounds in a sufficiently large scope. For example, consider the variable x of type `List<?>`. Capture conversion will convert the type of this variable to `List<`*X*`>` where *X* is some fresh type variable (without a bound). This conversion is sufficient to enforce the access restrictions of wildcard parameterized types. This organization of JAVA 5 explains why we talk about "apparent" member signatures in the previous section: the presented signatures are not explicitly considered by the type system, it will instead employ capture conversion and type the members with fresh type variables.

As opposed to an explicit opening operation, capture conversion does not allow the programmer to refer to the fresh type variable that is introduced. In the above example with type

`List<?>`, we have no name to refer to the fresh type variable X[7]. This is a problem for some operations, *e.g.*, we need define a variable of type X to use as temporary storage when swapping two elements in the list. Wildcards are designed such that, in such cases, access to this type variable can be recovered through a generic method. Observe that we can still invoke a generic method `m` with type X as a type argument: thanks to the inference of type arguments, we do not have to specify this type upon invocation. We can write the call `m(x)` and this will be taken to mean `<`X`>m(x)`, where X is inferred from the type of `x` (note that the inference is no longer a convenience mechanism here). Thus if we place the swapping operation inside a generic method:

```
<Y> void swap(List<Y> y) {
    Y tmp = y.get(0);
    y.set(0,y.get(1));
    y.set(1,tmp);
}
```

We will be able to invoke `swap(x)` and have access to X through the name `Y` in the method body. This mechanism of allowing a type variable to be instantiated to a wildcard is known as *wildcard capture*.

Finally, we observe that subtype relations for parameterized types with wildcards can be explained in terms of the subtyping rule for existential types, rule SEx in Figure 6.4 on page 94. The fact that we obtain subtypes of `Container<?>` by narrowing the wildcard corresponds to the narrowing relations that are required for the bounds of the existentially quantified type variable in the first two conditions of this rule:

```
Container<? extends Number>
 Container<? super Number>
          ...
```

The relation of the third condition corresponds to the other dimension of subtyping we have for this type, by considering subclasses of the generic class `Container`:

```
CopyContainer<?>
      ...
```

Note however that the rule describes subtype relations between existential types only, and therefore only explains those subtype relations between types parameterized with wildcards. The introduction of a wildcard that occurs when allowing invariant types, such as `Container<Float>`, as subtypes of `Container<?>`, corresponds to an implicit packing operation where `Float` fulfills the role of the witness type. In accordance with the second condition of rule PACK, this witness type must adhere to the bounds (if any) of the wildcard by which it will be hidden.

[7] Although the introduced type variables are sometimes referred to as "capture of ?" in compiler error messages, such a reference cannot be used by the programmer.

6.3.3 Discussion: Opportunities for Framework Designers

If there is a single conclusion to be taken from this chapter, then it should be that, with its generics feature, JAVA 5 transferred some advanced results from type systems research into the mainstream. This is quite surprising for a language that is criticized by many for at least lagging behind in the adoption of recent programming language technology, if not obstructing further development of that technology.

Nevertheless, the reception of JAVA 5 generics by the JAVA community has certainly not been unanimously positive. Thought leaders such as Arnold (2005); Eckel (2005); Bloch (2007) have criticized generics for its complexity; most complaints originate from constraints of the implementation technique (erasure), interaction with other language features, and sometimes the "under the hood" introduction of new type variables through capture conversion. Undeniably, a significant learning process is required to become proficient with generics, and this is sometimes overlooked because of the deceptive simplicity of a concept like `List<String>`.

However, it is equally true that generics have dramatically increased the expressiveness of the type system. Its 'killer app' is the collections framework from the standard library (Naftalin and Wadler, 2006, Part II), but we argue here that parametric polymorphism also enables typed versions of language frameworks: libraries that offer otherwise primitive operations as user-level functions. It is not a coincidence that our first example in Section 6.2.3 involves the wrapping of the primitive boolean test in an ordinary function with a type that is equivalent to the dedicated typing rule IFTHENELSE. Similarly, the type `java.lang.Class` in the standard library offers many examples of reflective features that are typed as generic methods and types (Naftalin and Wadler, 2006, Sec. 6.1). We will further exploit these possibilities when building a typed version of our AOP framework in Chapter 8.

Chapter 7

Typing Principles for Pointcut/Advice Bindings

In this chapter, we derive a set of principles for the flexible typing of pointcut and advice bindings. These principles are the basis for the type systems for the pointcut/advice mechanism presented in the remainder of this dissertation. More specifically, the goal of this chapter is to obtain a typing for pointcuts and one for advice, and a set of rules that judge the compatibility of pointcut and advice based on their respective types. The main criterion for designing such a typing, is that it should be a 'faithful' abstraction of the pointcut/advice mechanism: it should only allow pointcut/advice bindings for which the application of the advice to join points matched by the pointcut cannot cause type errors, and it should be sufficiently expressive to support common advice used in practice.

Section 7.1 characterizes the typical capabilities of advice behavior, and motivates that a function that 'wraps' a join point is a general model of advice. Section 7.2 presents the core contribution of this chapter: it formulates a soundness condition for advice application for general join points, and it develops two sets of typing principles by employing respectively subtype polymorphism and parametric polymorphism to meet this condition. It also considers the relation between these two sets of typing principles, and describes how one can be incorporated into the other. Section 7.3 specializes the resulting typing principles for the case of function join points, which are highly relevant for many concrete aspect languages. Finally, although the typing principles of Section 7.2 are expressive enough to support a large set of practical pointcut/advice pairs, Section 7.4 discusses the possibility of an extension to support join points with a specific type structure.

7.1 Characterization of Advice Behavior

As we have discussed before, different advice kinds are offered by most aspect approaches that support the pointcut/advice mechanism: before advice, after advice, around advice, *etc.* Of these different advice kinds, the around advice is the most expressive: it can emulate the other advice kinds. It is also the most interesting to type: because of its expressiveness, the typing will

be more complex, but once accomplished, the typing for the other advice kinds will follow in a straightforward manner. We will focus exclusively on around advice in this chapter.

An around advice is executed in place of the advised join point, and it will produce a result to return to the join point client. However, the advised join point may be used at will during the execution of the advice (it is typically available in around advice under the name **proceed**). For example, when advising a method execution join point, the advice can execute this method once, multiple times or not at all. The method arguments can be made available to the advice as context parameters, and new argument values may be specified by the advice for the invocation of the join point. Similarly, the advice can employ the results from join point invocations, and a different result may be returned to the join point client. In other words, the advice 'wraps' the advised join point, similar to the wrapping of a component in the *Decorator design pattern* described by Gamma et al. (1995, p. 175–184). The name "around" advice also stems from this interpretation. Yet another interpretation is that around advice is a function that receives a join point and returns a join point to be used by the join point client.

Note that, due to the use of pointcuts, a single advice may be applied to a number of join points of different types. Preferably, these different join point types are generalized in one pointcut type. When verifying the pointcut/advice binding, the compatibility between the advice type and the pointcut type should entail the compatibility of the advice with all join points matched by the pointcut.

7.2 Typing Principles for Join Points of General Type

We will develop principles for the typing of pointcut/advice bindings using the terminology and notations of the functional language from Section 6.2. We will consider any usage of a term as a join point; the context where the term is used is the *join point client*. For the sake of generality, we will first consider join points of any type. The join point will have a certain most specific type, but the join point client may be able to operate with a less specific type (recall that is generally possible to assign less specific types by means of the subsumption rule). We can summarize this situation as the following two propositions:

$$A_{jp} \vdash jp : S_{jp} \qquad A_{jp} \vdash S_{jp} <: T_{jp} \qquad \text{if } A_{jp} \vdash jp : T \text{ then } A_{jp} \vdash S_{jp} <: T$$

Here, we use the following metavariables: jp is the join point term, S_{jp} is the most specific join point type, T_{jp} is the type that the join point client expects from the join point and A_{jp} are the assumptions regarding term and type variables from the join point context. It is established that S_{jp} is indeed the most specific join point type by requiring in the third proposition that it is a subtype of any other type T of the join point term.

Following the description of advice behavior from the previous section, we will model an advice as a function that receives the join point as an argument and produces a result for the join point client. In case of an ordinary function, its type is a general function type:

$$\vdash adv : T_{pro} \text{ -> } T_{adv}$$

Here, adv is the advice function, T_{pro} is the argument type (with notation T_{pro} because this argument fulfills the role of **proceed**) and T_{adv} is the result type. Note that we consider only top-level functions as advices: the advice is therefore typed with no context assumptions.

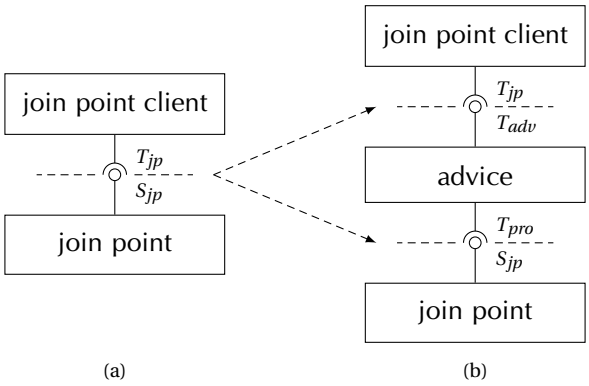


Figure 7.1: Join point interface types (a) before and (b) after advice weaving, in UML ball-and-socket notation (Object Management Group, 2005). The additional dashed lines illustrate the invasive effect of advice weaving.

As explained, advices are not directly connected to join points by the programmer. Pointcuts are used as intermediate language elements that (declaratively) describe a number of join points. We will not consider the specific details of any particular pointcut language here. We simply employ *pc* as a metavariable that stands for a pointcut expression, and we consider two judgments that involve pointcut expressions. One judgment holds when *pc* selects join point *jp* and is written:

$$A_{jp} \vdash pc \text{ matches } jp$$

Free term variables and type variables may appear in the join point term, and since the matching may depend on the assumptions regarding these variables in the context of the join point term, these assumption A_{jp} are included in the judgment. The other judgment concerns the valid binding between pointcut *pc* and advice *adv*:

$$\vdash (pc, adv) \text{ ok}$$

This judgment will determine if the pointcut and advice (or their respective types) are compatible. Our goal is to set-up the definition of this judgment in such a way that we are able to allow flexible pointcut/advice bindings, but still obtain a sound type system.

7.2.1 A Sufficient Condition for Soundness

To derive the typing principles, we start from the following observation. The weaving of an advice *adv* will replace some join points *jp* by the application of *adv* to *jp, i.e.*, it will replace term *jp* by term *adv jp*. However, this replacement occurs in a manner that is transparent to any client of the join point: such a client was specified and typed with the original term. We argue (informally) that the result of weaving will be sound if, for any type admitted by the original term, the replacement term admits the same type. In that case, a well-typed client of the original join point cannot use the replacement term in a manner that causes a run-time type error.

This situation is illustrated graphically in Figure 7.1 in terms of provided and expected elements. Originally, the join point is directly used by the join point client. The client will expect a certain join point type (T_{jp}), which is provided by the most specific type of the join point (S_{jp}). The advice weaving will change this situation: the advice will service the request of the join point client instead of the original join point; in turn, the advice may use this original join point. However, the join point client has not been changed and still expects a term of the same join point type (T_{jp}). The advice should therefore *preserve* this type, for any type that was expected. In addition, the join point has not changed either, so it will provide its original type (S_{jp}) to the advice.

We will formulate the observation regarding type preservation by advice application as a property that — for the explanatory purpose of this chapter — can be considered a sufficient condition for the soundness of the typing of pointcut/advice bindings (it is of course not a rigorous treatment of the problem):

$$\frac{\vdash (pc, adv)\ \text{ok} \qquad A_{jp} \vdash pc\ \text{matches}\ jp \qquad A_{jp} \vdash jp : T_{jp}}{A_{jp} \vdash adv\ jp : T_{jp}}\ \textsc{PSound}$$

Note that this is *not* the definition of a new typing rule! (This is the reason why we don't write the name PSOUND on the top.) It is simply a concise formulation of a property that we will consider sufficient for soundness: if the conditions above the horizontal bar guarantee the conclusion below the bar, for all bindings of the occurring metavariables, then we will consider the type system sound. Essentially, the property states that for any join point matched by a validly bound pointcut for an advice, the application of the advice to the join point should preserve any of the join point types (which are any of the types the join point client may expect). In the following sections we will start from this property and reason backwards to obtain typing rules.

7.2.2 Typing Advice with Subtype Polymorphism

When the advice is an ordinary function of type $T_{pro} \text{->} T_{adv}$ (*ordinary advice*), the typing of its application to a join point is governed by rule APPL. The join point will need to admit type T_{pro}, and the result of the application has type T_{adv}. However, in property PSOUND, both argument and result have the type T_{jp} instead. We will require appropriate subtype relations to be able to transition between these types. To discover these relations, we apply subsumption both before and after the function application. We will start from the most specific type of the join point (written S_{jp}) and obtain the same conclusion as property PSOUND:

$$\textsc{Sub}\ \frac{\textsc{Appl}\ \dfrac{\textsc{Sub}\ \dfrac{A_{jp} \vdash jp : S_{jp} \qquad A_{jp} \vdash S_{jp} <: T_{pro}}{A_{jp} \vdash jp : T_{pro}} \qquad \textsc{PWeak}\ \dfrac{\vdash adv : T_{pro} \text{->} T_{adv}}{A_{jp} \vdash adv : T_{pro} \text{->} T_{adv}}}{A_{jp} \vdash adv\ jp : T_{adv}} \qquad A_{jp} \vdash T_{adv} <: T_{jp}}{A_{jp} \vdash adv\ jp : T_{jp}}$$

Here, property PWEAK has been used. This is the *weakening property*, which states that derived typing judgments are not affected when extending the typing context with additional assumptions, *i.e.*, that reasoning is *monotonic*. This property is standard in most type systems, also the one from Section 6.2.

Pointcut expressions

$pc \quad ::= \quad S_{pc} - U_{pc} \quad$ *type range*

Pointcut matching $\boxed{A_{jp} \vdash pc \text{ matches } jp}$

$$\textsc{Match}\ \frac{A_{jp} \vdash jp : S_{jp} \qquad A_{jp} \vdash S_{pc} <: S_{jp} <: U_{pc} \qquad \text{if } A_{jp} \vdash jp : T \text{ then } A_{jp} \vdash S_{jp} <: T}{A_{jp} \vdash S_{pc} - U_{pc} \text{ matches } jp}$$

Valid pointcut/advice binding $\boxed{\vdash (pc, adv) \text{ ok}}$

$$\textsc{Bind}\ \frac{\vdash adv : T_{pro} \text{ -> } T_{adv} \qquad \vdash T_{adv} <: S_{pc} \qquad \vdash U_{pc} <: T_{pro}}{\vdash (S_{pc} - U_{pc}, adv) \text{ ok}}$$

$$\textsc{GBind}\ \frac{\vdash adv : \texttt{forall}\ X\ \texttt{in}\ S_{adv} - U_{adv}\texttt{,}\ X \text{ -> } X \qquad \vdash S_{adv} <: S_{pc} \qquad \vdash U_{pc} <: U_{adv}}{\vdash (S_{pc} - U_{pc}, adv) \text{ ok}}$$

Figure 7.2: Definition of pointcut matching and pointcut/advice binding for general join points

We will need to set up the typing of the pointcut/advice binding in such a way that the conditions of property PSOUND guarantee those judgments that are not accounted for in this derivation:

$$A_{jp} \vdash jp : S_{jp} \qquad A_{jp} \vdash S_{jp} <: T_{pro} \qquad A_{jp} \vdash T_{adv} <: T_{jp} \tag{7.1}$$

Note that the two subtype relations are also present in Figure 7.1: a term of type S_{jp} is provided by the join point to the advice where T_{pro} is expected, and the advice provides a result of type T_{adv} where the join point client expects T_{jp}.

As explained, we ensure the judgments (7.1) through the intermediate step of pointcut expressions and their types. Since the join point types are bounded on both ends in the relations, we propose to have pointcut types with the form of a *type range*: $S_{pc} - U_{pc}$, where a pointcut of this type should only match join points with a most specific type below upper bound U_{pc} and above lower bound S_{pc}. In fact, since we are not discussing a particular pointcut languages here, we will directly use such type ranges as pointcut expressions (see Figure 7.2) and define their matching behavior accordingly. In rule MATCH, the first and third condition require type S_{jp} to be the most specific join point type (specifically, the third condition requires *any* other type T_{jp} to be equal or more general). The second condition requires this most specific type S_{jp} to lie within the specified bounds. Note that the choice of a concrete pointcut expression form does not affect the generality of our conclusions: other pointcut languages may be employed, as long as we can derive a type range from a pointcut, where it is guaranteed that the pointcut will only match join points with a most specific type that lies within this type range.

The last two hypotheses of property PSOUND are $A_{jp} \vdash pc \text{ matches } jp$ and $A_{jp} \vdash jp : T_{jp}$. From the conditions of rule MATCH (and since this rule is the only way to establish pointcut matching), we then already have (by instantiating T to T_{jp}):

$$A_{jp} \vdash jp : S_{jp} \qquad A_{jp} \vdash S_{pc} <: T_{jp} \qquad A_{jp} \vdash S_{jp} <: U_{pc} \tag{7.2}$$

In rule BIND, we require that the pointcut type range $S_{pc} - U_{pc}$ is contained within the range $T_{adv} - T_{pro}$. When this rule is used to establish the first hypothesis of property PSOUND, we will

be able to extend the conclusions of (7.2) to obtain the relations of (7.1). Concretely, the first relation is obtained as follows:

$$\dfrac{A_{jp} \vdash S_{jp} <: U_{pc} \qquad \dfrac{\vdash U_{pc} <: T_{pro}}{A_{jp} \vdash U_{pc} <: T_{pro}}\ \textsc{PWeak}}{A_{jp} \vdash S_{jp} <: T_{pro}}\ \textsc{STrans}$$

The case of the second relation is entirely analogous. This completes the demonstration of property PSound for the case of this section (discussion of rule GBind is postponed to the following section).

Example. The following top-level function receives a record argument and constructs a new record with an updated `fst` field and a fixed `snd` field:

```
let adv_sub (proceed : {fst : Int}) = {fst = succ proceed.fst, snd = true}
val adv_sub : {fst : Int} -> {fst : Int, snd : Bool}
```

According to rule Bind, this function can be bound as an advice to pointcuts with the following type range (or a narrower range):

$$\texttt{\{fst : Int, snd : Bool\}} - \texttt{\{fst : Int\}}$$

It can therefore be applied to record join points with a most specific type within this type range.

Remark. Observe in rule Match that a pointcut $S_{pc} - U_{pc}$ can only match join points when $S_{pc} <: U_{pc}$. And for a pointcut where this relation holds, rule Bind will ensure that it can only be bound to an advice of type $T_{pro} \texttt{ -> } T_{adv}$ when $T_{adv} <: T_{pro}$. So while advice functions without $T_{adv} <: T_{pro}$ or pointcuts without $S_{pc} <: U_{pc}$ are not directly prohibited, they cannot be used to advise any join points.

7.2.3 Typing Advice with Parametric Polymorphism

Besides subtype relations, there is another important technique to have the application of an advice preserve the type of the join point (and thus satisfy property PSound). If the advice is a generic function of type $X \texttt{ -> } X$ (*generic advice*), where X is some universally quantified type variable, we can conduct the application in two steps. First we instantiate the type abstraction with the most specific join point type S_{jp}:

$$\dfrac{\dfrac{\vdash adv : \texttt{forall } X \texttt{ in } S_{adv} - U_{adv}, X \texttt{ -> } X}{A_{jp} \vdash adv : \texttt{forall } X \texttt{ in } S_{adv} - U_{adv}, X \texttt{ -> } X}\ \textsc{Weak} \qquad A_{jp} \vdash S_{adv} <: S_{jp} <: U_{adv}}{A_{jp} \vdash adv\ [S_{jp}] : S_{jp} \texttt{ -> } S_{jp}}\ \textsc{TApp}$$

(Again, the weakening property is used to first bring the advice function in the context of the join point.) The resulting advice function can then be applied to the join point term. We can then obtain the conclusion of property PSound as follows:

$$\dfrac{\dfrac{A_{jp} \vdash jp : S_{jp} \qquad A_{jp} \vdash adv\ [S_{jp}] : S_{jp} \texttt{ -> } S_{jp}}{A_{jp} \vdash adv\ [S_{jp}]\ jp : S_{jp}}\ \textsc{Appl} \qquad A_{jp} \vdash S_{jp} <: T_{jp}}{A_{jp} \vdash adv\ [S_{jp}]\ jp : T_{jp}}\ \textsc{Sub}$$

In this derivation, we have used the following hypotheses:

$$A_{jp} \vdash S_{adv} <: S_{jp} <: U_{adv} \qquad A_{jp} \vdash jp : S_{jp} \qquad A_{jp} \vdash S_{jp} <: T_{jp} \tag{7.3}$$

Similar to the case of subtype polymorphism from previous section, we set-up matching and binding such that we can derive these judgments from the hypotheses of property PSOUND. Matching is not modified, and from the last two hypotheses and the conditions of rule MATCH (and by instantiating T to T_{jp}), we already have:

$$A_{jp} \vdash jp : S_{jp} \qquad A_{jp} \vdash S_{jp} <: T_{jp} \qquad A_{jp} \vdash S_{pc} <: S_{jp} <: U_{pc} \tag{7.4}$$

In rule GBIND, which is also defined in Figure 7.2, the conditions require that the range S_{adv}–U_{adv} of the advice's type variable bounds contains the pointcut range S_{pc} – U_{pc}. This allows extend the relation from (7.4) to derive the remaining relation from (7.3):

$$\dfrac{A_{jp} \vdash S_{pc} <: S_{jp} <: U_{pc} \qquad \dfrac{\vdash S_{adv} <: S_{pc} \qquad \vdash U_{pc} <: U_{adv}}{A_{jp} \vdash S_{adv} <: S_{pc} \qquad A_{jp} \vdash U_{pc} <: U_{adv}}\ \textsc{PWeak},\textsc{PWeak}}{A_{jp} \vdash S_{adv} <: S_{jp} <: U_{adv}}\ \textsc{STrans},\textsc{STrans}$$

With these derivations we have completed a second way to ensure property PSOUND.

Example. We revisit the function `bnd_lo` from page 91. It returns its argument or some fixed record value, depending on some condition `c`. Reformulated with `proceed` as the argument name, this becomes:

```
let adv_lo [X in {fst : Int, snd : Bool} – Top] (proceed : X) =
      if c then {fst = 1, snd = true} else proceed
val adv_lo : forall X in {fst : Int, snd : Bool} – Top, X –> X
```

According to rule GBIND, this function can be bound as an advice to pointcuts with a type range that is equal or narrower than the type range of type variable `X`. It can therefore be applied to a join point with a most specific type that lies within this range, for example, a record join point of type `{fst : Int}`.

Incorporation of subtype polymorphism Thanks to the preliminary material of Section 6.2, we have been able to formulate rule GBIND for generic advices directly with double bounded quantification of the type variable. To illustrate the expressiveness of this advanced form, we demonstrate that it is capable of incorporating the case of subtype polymorphism from the previous section.

Consider an ordinary advice function *adv* with type T_{pro} –> T_{adv}. This function can be used as a function `X –> X` when the type variable `X` is declared to lie within the type range T_{adv} – T_{pro}: it will accept any value of type `X`, since `X` is a subtype of T_{pro}, and its result will always be of type `X` since T_{adv} is a subtype of `X`. More formally, using the shorthand $A_0 \stackrel{\text{def}}{=} \mathtt{X}\ \mathbf{in}\ T_{adv} - T_{pro}$ and some of the subtyping rules from Figure 6.4, we can derive:

$$\dfrac{\dfrac{\dfrac{\vdash adv : T_{pro} \text{ -> } T_{adv}}{A_0 \vdash adv : T_{pro} \text{ -> } T_{adv}}\ \textsc{Weak} \qquad \dfrac{\dfrac{}{A_0 \vdash \mathtt{X} <: T_{pro}}\ \textsc{SVarU} \qquad \dfrac{}{A_0 \vdash T_{adv} <: \mathtt{X}}\ \textsc{SVarS}}{A_0 \vdash (T_{pro} \text{ -> } T_{adv}) <: (\mathtt{X} \text{ -> } \mathtt{X})}\ \textsc{SFun}}{A_0 \vdash adv : \mathtt{X} \text{ -> } \mathtt{X}}\ \textsc{Sub}}{\vdash (\mathbf{all}\ \mathtt{X}\ \mathbf{in}\ T_{adv} - T_{pro} \text{ => } adv) : \mathbf{forall}\ \mathtt{X}\ \mathbf{in}\ T_{adv} - T_{pro}, \mathtt{X} \text{ -> } \mathtt{X}}\ \textsc{TAbs}$$

(For the sake of completeness, we add that the application of rule TABS additionally requires that types T_{pro} and T_{adv} are well-formed. However, this can be straightforwardly established from the given that *adv* has type composed of these two types, namely T_{pro} -> T_{adv}.)

We obtain a universal type for the type abstraction **all** X **in** T_{adv} - T_{pro} => *adv*. If we now consider rule GBIND with its metavariable *adv* bound to this type abstraction, and metavariables S_{adv}, U_{adv} and X bound to respectively T_{adv}, T_{pro} and X, we obtain:

$$\frac{\vdash (\textbf{all}\ \mathtt{X}\ \textbf{in}\ T_{adv} - T_{pro}\ \Rightarrow adv) : \textbf{forall}\ X\ \textbf{in}\ T_{adv} - T_{pro}, \mathtt{X} \to \mathtt{X} \qquad \vdash T_{adv} <: S_{pc} \qquad \vdash U_{pc} <: T_{pro}}{\vdash (S_{pc} - U_{pc}, \textbf{all}\ \mathtt{X}\ \textbf{in}\ T_{adv} - T_{pro}\ \Rightarrow adv)\ \text{ok}}$$

The first condition is the typing of the type abstraction **all** X **in** T_{adv} - T_{pro} => *adv* we obtained above. The second and third conditions are exactly the same as those of rule BIND. So we can bind this type abstraction as an advice to exactly the same pointcuts as was possible for the original advice *adv*.

The conclusion is that rule BIND is a special case of rule GBIND: any pointcut/advice binding admitted by the former can also be admitted by the latter, if the advice *adv* is wrapped in a type abstraction **all** X **in** T_{adv} - T_{pro} => *adv*, where T_{adv} and T_{pro} are respectively the result and argument type of *adv*.

Example. We can abstract the advice adv_sub from page 112 as a function of type X->X when X is declared with the appropriate bounds:

```
let adv_gsub [X in {fst : Int, snd : Bool} - {fst : Int}] = adv_sub as (X -> X)
val adv_gsub : forall X in {fst : Int, snd : Bool} - {fst : Int}, X -> X
```

So far, we have always assigned the most specific type to top-level bindings, but here the *ascription* operation (Pierce, 2002, Sec. 11.4) is used in the body term to obtain a more general type (note that type X->X is a subtype of the original type of adv_sub). The ascription operation is the equivalent of an explicit *up-cast* in languages such as JAVA. When a term e is well-typed with some type S, then its admits all supertypes of S as well (because of the subsumption rule). By ascribing e the type T (written e **as** T), where T is one of the supertypes of S, the admissible types are restricted to a subset consisting of T and its supertypes. The formal typing rule for the ascription operation is simple:

$$\text{ASC}\ \frac{A \vdash e : T}{A \vdash (e\ \textbf{as}\ T) : T}$$

To meet the condition of this rule, the type checker must always at least subsume the original type of e by T. As such, the ascription operation can force a less specific type for e.

As an alternative to function adv_gsub, we can directly specify the behavior of adv_sub as a generic advice method of type X->X. The proceed argument is of a more specific type than it needs to be, and the result is again ascribed to a more general type:

```
let adv_gsub' [X in {fst : Int, snd : Bool} - {fst : Int}] (proceed : X) =
    {fst = succ proceed.fst, snd = true} as X
val adv_gsub' : forall X in {fst : Int, snd : Bool} - {fst : Int}, X -> X
```

We thus obtain adv_gsub' with the same type as adv_gsub. According to rule GBIND, both functions can be used as advice with pointcuts with a type range that is equal or narrower than the range of type variable X. This is the same range as allowed by rule BIND for the original function adv_sub.

7.3 Typing Principles for Function Join Points

We will now formulate versions of the typing rules of the previous section that apply to the specific case of function join points. The motivation for this exercise is the large importance of function join points in practical AOP approaches, as explained below. The formulation of these specialized typing rules will be helpful to transfer the results from this chapter to such approaches in the next chapters.

7.3.1 Relevance of Function Join Points

The pointcut/advice mechanism in practical aspect approaches will typically model join points as functions (or procedures, if the term "function" is restricted to subroutines without side effects). This occurs not only when the underlying join point is a method or constructor invocation (constructs which can be interpreted rather easily as some kind of procedure), also field references or updates, initializers and exception handlers are presented as functions to the advice. Recall that join points are nodes in the execution tree, and such a node can generally be modeled as a function.

For example, when an around advice intercepts a field reference, then it will have access to this join point through the **proceed** function or method[1]. The invocation of **proceed** will trigger the execution of the join point. The arguments of this function correspond to certain designated parameters from the context of the join point, for example, the object from which the field is to be retrieved. The result of **proceed** will be the result of the join point execution, which is the value of the field in this case.

Note that the unification of different join point kinds as functions may be practical rather than conceptual: by presenting every kind of join point as a function to the advice, the same advice can be used for join points of different kinds.

7.3.2 Function Join Points and Pointcuts

The development of specific versions of the rules of the previous sections proceeds by replacing the general type of a join point by a function type consisting of an argument type and a result type. The application of the subtyping rule for function types will then require separate relations for both the arguments types and the result types, but in opposite direction.

As a first example, consider a function join point. This join point has a general function type, which the join point client may expect, and which we will write as follows:

$$A_{jp} \vdash jp : T^{i}_{jp} \text{ -> } T^{o}_{jp}$$

[1] And typically a closure function is created to implement this functionality, as explained by, *e.g.*, Hilsdale and Hugunin (2004).

Pointcut expressions

$pc \quad ::= \quad (U^i_{pc} - S^i_{pc}) \text{ -> } (S^o_{pc} - U^o_{pc}) \quad$ *type range function*

Pointcut matching $\boxed{A_{jp} \vdash pc \text{ matches } jp}$

$$\text{FMATCH}\ \frac{A_{jp} \vdash jp : S^i_{jp} \text{ -> } S^o_{jp} \qquad A_{jp} \vdash U^i_{pc} <: S^i_{jp} <: S^i_{pc} \qquad A_{jp} \vdash S^o_{pc} <: S^o_{jp} <: U^o_{pc} \quad \text{if } A_{jp} \vdash jp : T^i \text{ -> } T^o \text{ then } A_{jp} \vdash T^i <: S^i_{jp} \text{ and } A_{jp} \vdash S^o_{jp} <: T^o}{A_{jp} \vdash (U^i_{pc} - S^i_{pc}) \text{ -> } (S^o_{pc} - U^o_{pc}) \text{ matches } jp}$$

Valid pointcut/advice binding $\boxed{\vdash (pc, adv) \text{ ok}}$

$$\text{FBIND}\ \frac{\vdash adv : (T^i_{pro} \text{ -> } T^o_{pro}) \text{ -> } (T^i_{adv} \text{ -> } T^o_{adv}) \quad \vdash T^i_{pro} <: U^i_{pc} \qquad \vdash S^i_{pc} <: T^i_{adv} \qquad \vdash T^o_{adv} <: S^o_{pc} \qquad \vdash U^o_{pc} <: T^o_{pro}}{\vdash ((U^i_{pc} - S^i_{pc}) \text{ -> } (S^o_{pc} - U^o_{pc}), adv) \text{ ok}}$$

$$\text{FGBIND}\ \frac{\vdash adv : \mathbf{forall}\ I\ \mathbf{in}\ U^i_{adv} - S^i_{adv}, O\ \mathbf{in}\ S^o_{adv} - U^o_{adv}, (I \text{ -> } O) \text{ -> } (I \text{ -> } O) \quad \vdash U^i_{adv} <: U^i_{pc} \qquad \vdash S^i_{pc} <: S^i_{adv} \qquad \vdash S^o_{adv} <: S^o_{pc} \qquad \vdash U^o_{pc} <: U^o_{adv}}{\vdash ((U^i_{pc} - S^i_{pc}) \text{ -> } (S^o_{pc} - U^o_{pc}), adv) \text{ ok}}$$

Figure 7.3: Definition of pointcut matching and pointcut/advice binding for function join points

Here, T^i_{jp} is the argument type of the function (input type) and T^o_{jp} is the result type (output type). A pointcut to select function join points types will consist of a type range between two function types:

$$pc ::= (S^i_{pc} \text{ -> } S^o_{pc}) - (U^i_{pc} \text{ -> } U^o_{pc})$$

According to rule MATCH, this pointcut will match join points with a most specific type between these two bounds:

$$A_{jp} \vdash (S^i_{pc} \text{ -> } S^o_{pc}) <: (S^i_{jp} \text{ -> } S^o_{jp}) <: (U^i_{pc} \text{ -> } U^o_{pc})$$

The subtyping rule for function types (rule SFUN) splits each of these subtype relations in a relation between the result types and another relation, in opposite direction, between the argument types:

$$A_{jp} \vdash U^i_{pc} <: S^i_{jp} <: S^i_{pc} \qquad A_{jp} \vdash S^o_{pc} <: S^o_{jp} <: U^o_{pc}$$

The above pointcut for function join points has thus been split in two separate type ranges: $U^i_{pc} - S^i_{pc}$ for the argument type and $S^o_{pc} - U^o_{pc}$ for the result type.

Because of this observation, we will use an alternative notation for pointcuts for function join points in Figure 7.3: rather than specifying the pointcut as a range of function types, we will use the notation of a function of type ranges[2]. The specialized matching rule itself is given

[2]This is not very different from the previous notation, it only specifies the types S^i_{pc}, U^i_{pc}, S^o_{pc} and U^o_{pc} in a different order. But it may improve the understandability of the matching and binding rules.

as rule FMATCH in this figure. The last condition (to ensure that we are working with the most specific type of the join point) has also been specialized for function types.

7.3.3 Ordinary Function Advice

An ordinary advice function for function join points is a function that receives a **proceed** function, and returns a new function. Its general type is:

$$\vdash adv : (T^i_{pro} \text{ -> } T^o_{pro}) \text{ -> } (T^i_{adv} \text{ -> } T^o_{adv})$$

Example. The following advice increases an integer argument value before passing it to the function join point; the result is decreased before returning it to the join point client (we assume that there exist functions `succ` and `pred` for increasing and decreasing integer values):

```
let adv_int (proceed : Int -> Int) (x : Int) = pred (proceed (succ x))
val adv_int : (Int -> Int) -> Int -> Int
```

Recall here that a function with multiple arguments is an abbreviation of multiple nested functions, and that the arrow operator for function types is right associative. The type of `adv_int` is the same as (**Int** -> **Int**) -> (**Int** -> **Int**).

We can specialize rule BIND for advice functions of the above type. The application of this rule will require the following subtype relations in this case:

$$\vdash (T^i_{adv} \text{ -> } T^o_{adv}) <: (S^i_{pc} \text{ -> } S^o_{pc}) \qquad \vdash (U^i_{pc} \text{ -> } U^o_{pc}) <: (T^i_{pro} \text{ -> } T^o_{pro})$$

Again, rule SFUN will split each of these subtype relations in separate relations for argument types and result types. This leads to the specialized rule FBIND in Figure 7.3. Notice in this rule how the direction of the advice range similarly differs between the argument and result types: $T^i_{pro} - T^i_{adv}$ and $T^o_{adv} - T^o_{pro}$. Both of these two ranges should contain the corresponding type ranges of the pointcut (*resp.* $U^i_{pc} - S^i_{pc}$ and $S^o_{pc} - U^o_{pc}$).

7.3.4 Generic Function Advice

We can similarly consider a generic advice function for function join points. This will be a function of type X -> X, where the bounds for type variable X are function types:

$$\vdash adv : \mathbf{forall}\ X\ \mathbf{in}\ (S^i_{adv} \text{ -> } S^o_{adv}) - (U^i_{adv} \text{ -> } U^o_{adv}),\ X \text{ -> } X$$

Here, type variable X may be bound to any type that lies within the range of the two function types. Notice that the only type arguments that can meet these bounds will be functions types, which, according to rule SFUN, must have an argument type within range $U^i_{adv} - S^i_{adv}$ and a result type within range $S^o_{adv} - U^o_{adv}$. A more direct representation of the type of this generic function can therefore be obtained by replacing type X by function type I -> O, where we use the metavariable I to represent the type variable for argument types, and the metavariable O to represent the one for return types:

$$\vdash adv : \mathbf{forall}\ I\ \mathbf{in}\ U^i_{adv} - S^i_{adv},\ O\ \mathbf{in}\ S^o_{adv} - U^o_{adv},\ (I \text{ -> } O) \text{ -> } (I \text{ -> } O)$$

(Here, the nesting of two universal types is given a less heavyweight notation by not repeating the keyword **forall**.)

Example. We can consider a variation of function adv_lo from page 113. In the following advice function, some condition c will determine whether to invoke the original join point or not. If yes, the original argument is used and the result is directly returned. If no, a new term is constructed using the original argument.

```
let adv_flo [I in Bot - {fst : Int}] [0 in {fst : Int, snd : Bool} - Top] (proceed :
    I -> 0) (x : I) = if c then proceed x else {fst = succ x.fst, snd = true}
val adv_flo : forall I in Bot - {fst : Int}, 0 in {fst : Int, snd : Bool} - Top, (I ->
    0) -> I -> 0
```

The bounds on type variables I and 0 are both required for the second branch of the boolean test: the upper bound of I allows to access field fst of the argument x; the lower bound of 0 allows to assign type 0 to the entire result of the branch.

For the binding of advice functions of the above form, rule GBIND will require the following subtype relations:

$$\vdash (S^i_{adv} \text{ -> } S^o_{adv}) <: (S^i_{pc} \text{ -> } S^o_{pc}) \qquad \vdash (U^i_{pc} \text{ -> } U^o_{pc}) <: (U^i_{adv} \text{ -> } U^o_{adv})$$

We obtain rule FGBIND from Figure 7.3 by again incorporating rule SFUN to establish these relations. According to the conditions of this rule, range $U^i_{adv} - S^i_{adv}$ of type variable I must contain the pointcut argument range $U^i_{pc} - S^i_{pc}$, while range $S^o_{adv} - U^o_{adv}$ of type variable 0 must contain the pointcut result range $S^o_{pc} - U^o_{pc}$.

Mixing subtype and parametric polymorphism We will still have that ordinary advice functions can be incorporated in generic advice functions with appropriate bounds, *i.e.*, this conclusion from Section 7.2.3 stays in effect. However, it is recommended to only employ type variables and quantification with bounds when the expressive power of these constructs is strictly required to obtain more flexible pointcut/advice bindings. Otherwise, an advice function is generally easier to understand without these constructs.

In the case of advice for function join points, we additionally remark that an advice function may mix the two styles of polymorphism. The advice may represent the argument type of the join point with a regular type and the result with a type variable or vice versa. In general, it becomes beneficial to use a type variable for the argument type when the advised join point is invoked with the original argument value. Similarly, it is useful to represent the result type with a type variable when the result of a join point invocation is used as the result of the advice.

Example. We can demonstrate this by considering two variations of function adv_int from page 117. The first will modify the argument, but return the result directly, while the second will employ the argument directly, but modify the result:

```
let adv_mix1 [0] (proceed : Int -> 0) (x : Int) = proceed (succ x)
val adv_mix1 : forall 0, (Int -> 0) -> (Int -> 0)

let adv_mix2 [I] (proceed : I -> Int) (x : I) = decr (proceed x)
val adv_mix2 : forall I, (I -> Int) -> (I -> Int)
```

The valid pointcut bindings for these advice functions can be determined through appropriate combinations of rule FBIND and rule FGBIND. In case of `adv_mix1` no benefit is gained from representing the argument type by a type variable (the argument type will always need to be **`Int`**). However, by employing a type variable for the result type, the typing can express that the advice always preserves this type, which allows more flexible pointcut bindings. There is a similar situation in the opposite direction for function `adv_mix2`.

Conclusion This section concludes the presentation of typing principles that are the basis for the type systems presented in the remainder of this dissertation. The polymorphism mechanisms presented in this chapter provide highly expressive mechanisms for the typing of pointcut/advice bindings. We have considered subtype polymorphism, as well as parametric polymorphism with bounded quantification, and we have applied the resulting typing principles to the special case of function join points. This will be sufficient to support most of the advice specifications that are commonly used in practice. To illustrate that it is possible to imagine support for bindings beyond the mechanisms presented in this chapter, we devote a short section to another case below.

7.4 Join Points with a Special Type Structure

In this section, we consider the case of (advice for) join points with a type structure where different parts are interrelated. As an example, consider the following advice function that swaps the first two fields of a record join point (and adds a third field):

```
let adv_stru [X] (proceed : {fst : X, snd : X}) =
    {fst = proceed.snd, snd = proceed.fst, trd = true}
val adv_stru : forall X, {fst : X, snd : X} -> {fst : X, snd : X, trd : Bool}
```

Intuitively, we observe that this advice function can be safely applied to a record join points with a most specific type that adheres to a certain type structure. Some example types:

```
{fst : Int, snd : Int}
{fst : Int, snd : Int, trd : Bool}
{fst : Bool, snd : Bool}
{fst : Bool, snd : Bool, trd : Bool}
...
```

In all of these cases, the `fst` and `snd` field may be interchanged (*i.e.,* these fields have the same type), and there may or may not be a `trd` field of type **`Bool`**. Although subtype polymorphism allows us specify that a `trd` field is optional, our current mechanisms cannot express the relation between the types of the `fst` and `snd` fields of the join point.

A first step to type the bindings of `adv_stru` to join points with the described type structure, would be to add a pointcut type that guarantees that only join points with the specific type structure are matched. Such a pointcut type would be reminiscent of a type range with a type variable that is existentially quantified:

```
exists X, {fst : X, snd : X, trd : Bool} - {fst : X, snd : X}
```

Join points matched by a pointcut of this type would need to have a most specific type that lies within the specified type range, for some binding of type variable X. The pointcut could then be safely bound to an advice with a complementary universal type such as the one from `adv_stru`.

The practical utility of advising such join points with a special type structure remains to be investigated. Exploring the subject in further detail is beyond the scope of this dissertation.

Chapter 8

Safe Deployment Logic in EcoSys

This chapter incorporates the typing principles from Chapter 7 in the EcoSys approach from Chapter 4. Since EcoSys is a framework-based AOP approach, Section 8.1 briefly reviews the type checking facilities in current AOP frameworks. Section 8.2 then develops a typed version of the EcoSys programming interface using a novel technique, by employing the advanced mechanisms of Java 5 generics. We also present the incorporation of a form of typed pointcuts in the EcoSys implementation. Next, Section 8.3 explores the currently unsolved topic of typed interaction resolutions. Finally, Section 8.4 presents some examples of real-life advice behavior expressed (and typed) using the EcoSys approach. The examples illustrate how both the typing schemes of ordinary and generic advice are employed in practice. The breadth of the examples also provides a reasonable indication that the typing supports all the common classes of advice behavior.

8.1 Current AOP Framework Typing

As explained in Section 2.2.3, AOP frameworks employ only standard language constructs to describe aspect behavior. The advice code is compiled using a standard compiler that is not aware of the aspects and the weaving is carried out at load-time or run-time. As a consequence, these approaches typically offer limited static type safety guarantees in comparison to language extensions such as AspectJ.

For example, the AOP Alliance specification by Pawlak et al. (2004), which is adopted by a number of approaches including JAC and Spring/AOP, predefines a number of interfaces to be used for the specification of advice behavior. In these interfaces, argument and return types are made generic by reducing all types to the general `Object` type. And while JBoss/AOP employs its own set of interfaces, these are also typed using the general `Object` type, as demonstrated in Listing 2.8 on page 22. This measure precludes any static type checking and forces the developer to include numerous casts, which mark the points where type checks will occur at run-time. The *untyped*[1] version of EcoSys in Chapter 4 is very similar to this approach.

[1] The untyped version of EcoSys is in fact dynamically typed, but since dynamic typing is a minimum for prevalent languages today, and entirely untyped languages are seldom considered in our context, we use the terms *typed* and

Some improvement is achieved by the annotation-based style of ASPECTJ (Colyer et al., 2005), which implements advice methods as regular JAVA methods with predefined annotations Since advice methods have a concrete signature, their body can be checked by the standard JAVA compiler. Additionally, the signature is checked for compatibility with the bound pointcut by the aspect weaver at load-time. However, to implement around advice and `proceed` invocations, the advice method must declare a parameter of the predefined interface `ProceedingJoinPoint`. This interface again employs general `Object` argument and return types, and therefore reduces safety guarantees to mere dynamic type checking.

8.2 Typed EcoSys

The advanced generics feature of JAVA 5 (among others), which we discussed in Section 6.3, offer new possibilities for the static typing of AOP frameworks. We will employ features such as type variables and wildcard parameterized classes to enforce the typing principles proposed in Chapter 7 (and more specifically Section 7.3) using a standard JAVA 5 compiler. We do this by developing a typed version of the ECOSYS AOP framework. This version offers expressive aspects deployments as well as a compile-time type checking of the advice behavior. Both the advice bodies and the pointcut/advice bindings are checked, and the checking of `proceed` invocations is supported; this provides clear improvements over the safety guarantees provided by current AOP frameworks.

8.2.1 Adaptations to the Programming Interface

Advice interfaces The `JoinPoint` interface of ECOSYS represents a function join point that receives one argument and returns a result. In Section 7.3, we model advice for function join points as an advice function that transforms a function into a new function. This corresponds to the `Advice` interface of ECOSYS, although in a more direct sense: the interface defines an `around` method which receives a function join point as a parameter, and after this parameter has been bound, the resulting method may be employed as a new function join point.

Additionally, we defined two kinds of typing for function advice in Section 7.3. Ordinary advice (Section 7.3.3) is typed using subtype polymorphism, while parametric polymorphism and type variables are used for generic advice (Section 7.3.4). Listing 8.1 presents a typed version of the `JoinPoint` and `Advice` interfaces, according to the function types for advice functions from these sections:

- The `JoinPoint` interface is equipped with type parameters that represent the result type (named `O` for output) and argument type (named `I` for input). As a result, the parameterized type `JoinPoint<`O`,`I`>` corresponds directly to the function type I –> O.
- Different versions of the `Advice` interface are defined according to the different kinds of typing for function advice from Section 7.3:
  - The interface `OrdAdvice` represents ordinary advice. It is parameterized with four type parameters that represent input and output types for the advice (with suffix a

untyped to refer to static typing only.

```
1  public interface JoinPoint<O,I> {
2      O invoke(I arg);
3  }
4
5  public interface OrdAdvice<Oa,Op,Ia,Ip> {
6      Oa around(Ia i, JoinPoint<Op,Ip> proceed);
7  }
8
9  public interface GenAdvice<Ou,Iu> {
10     <O extends Ou, I extends Iu> O around(I i, JoinPoint<O,I> jp);
11 }
12
13 public interface MixAdvice1<Oa,Op,Iu> {
14     <I extends Iu> Oa around(I i, JoinPoint<Op,I> jp);
15 }
16
17 public interface MixAdvice2<Ou,Ia,Ip> {
18     <O extends Ou> O around(Ia i, JoinPoint<O,Ip> jp);
19 }
```

Listing 8.1: Join point and advice interface classes of typed EcoSys

for advice) and the join point argument (with suffix `p` for proceed). This interface defines an ordinary `around` method with a signature that corresponds to the advice function type:

$$(T^i_{pro} \text{ -> } T^o_{pro}) \text{ -> } T^i_{adv} \text{ -> } T^o_{adv}$$

The four component types of this function type are represented by the respective type variables `Ip`, `Op`, `Ia` and `Oa`.

– Generic advice is represented by the interface `GenAdvice`. It is parameterized with two type parameters that represent the upper bounds (suffix `u`) for the input and output type of the join point. (Unfortunately, Java 5 generics does not support lower bounds for type variables.) The interface defines a generic `around` method with a signature that corresponds to the advice function type:

$$\textbf{forall } I <: S^i_{adv}, O <: U^o_{adv}, (I \text{ -> } O) \text{ -> } I \text{ -> } O$$

In the signature, type variables `I` and `O` assume the roles of I and O, while `Iu` and `Ou` assume roles S^i_{adv} and U^o_{adv}.

– Additionally, the possibility to mix both typing styles within one and the same advice function is also introduced in Section 7.3.4. The interfaces `MixAdvice1` and `MixAdvice2` represent the two possible combinations. Interface `MixAdvice1` is defined with ordinary result types and a variable argument type, while interface

```
1  public abstract class Pointcut<O,I,JP> {
2      public Binding<JP>
3      bind(GenAdvice<? super O, ? super I> adv) {
4           ...
5      }
6
7      public Binding<JP>
8      bind(OrdAdvice<? extends O, ? super O, ? super I, ? extends I> adv) {
9           ...
10     }
11
12     public Binding<JP>
13     bind(MixAdvice1<? extends O, ? super O, ? super I> adv) {
14          ...
15     }
16
17     public Binding<JP>
18     bind(MixAdvice2<? super O, ? super I, ? extends I> adv) {
19          ...
20     }
21 }
```

Listing 8.2: Pointcut class of typed EcoSys

`MixAdvice2` has a variable result type and ordinary argument types. The definition of their type variables and their `around` methods is obtained by making an appropriate combination of the above two cases.

Binding type relations The corresponding new version `Pointcut` class is presented in Listing 8.2. It realizes the type relations for pointcut/advice bindings from Section 7.3. The class defines two additional type variables `O` and `I` that represent the result type and argument type of the join points matched by the pointcut. Obviously, a pointcut may match join points with multiple result types or multiple argument types. However, wildcard parameterizations of the generic `Pointcut` type may be employed in such cases; if the wildcard type parameter includes an upper or a lower bound, then this corresponds to guarantees about the result or argument types of the matched join points. For example, a pointcut of type `Pointcut<?, ? extends Person>` will match only join points where the argument type is a subtype of `Person` (the result type is not constrained and may be any type). Similarly, a wildcard type parameter `? super Person` indicates that only supertypes of `Person` are matched. It is an important assumption of typed EcoSys that the employed pointcut formalism only constructs `Pointcut` objects where the matching behavior corresponds to the type information given in the type parameters. This is the responsibility of the particular EcoSys implementation that provides the pointcut entities. We elaborate on how this can be realized in Section 8.2.2.

```
1 class IntAdvice implements OrdAdvice<Integer,Number,Number,Integer> {
2     Integer around(Number arg, JoinPoint<Number,Integer> proceed) {
3         return proceed.invoke(arg.intValue() + 1).intValue() - 1;
4     }
5 }
```

Listing 8.3: Example of ordinary advice behavior in typed EcoSys

The interface of the `Pointcut` class now overloads the `bind` method to provide a different version for each of the advice types. The signatures of these method implement the actual type relations from the binding rules specified in Figure 7.3 on page 116:

- For the case of generic advice, instances of type `GenAdvice<`O_u`,` I_u`>` are only accepted when O_u is provably a supertype of all join point result types, and I_u is provably a supertype of all join point argument types. This is the case if the wildcard pointcut type arguments have upper bounds which are subtypes of O_u and I_u. Consequently, this enforces type relations that correspond to the following conditions from rule FGBIND:

  $$\vdash S^i_{pc} <: S^i_{adv} \qquad \vdash U^o_{pc} <: U^o_{adv}$$

  Since it is not possible to declare generic advice with lower bounds for its type variables, the other conditions are always met.

- For ordinary advice, instances of type `OrdAdvice<`O_a`,` O_p`,` I_a`,` I_p`>` are only accepted when O_a is a provably subtype of all join point result types, and O_p a supertype, and when I_a is provably a supertype of all join point argument types, and I_p a subtype. This is the case if the wildcard pointcut type argument for the result has a lower bound which is supertype of O_a and an upper bound which is a subtype of O_p, and the wildcard pointcut type argument for the argument has an upper bound which is a subtype of I_a, and a lower bound which is a supertype of I_p. Consequently, this enforces type relations that correspond to the conditions from rule FBIND, in the following order:

  $$\vdash T^o_{adv} <: S^o_{pc} \qquad \vdash U^o_{pc} <: T^o_{pro} \qquad \vdash S^i_{pc} <: T^i_{adv} \qquad \vdash T^i_{pro} <: U^i_{pc}$$

- For advice that mixes both typing styles, an appropriate combination of the above cases is employed.

Example (Ordinary advice). In Listing 8.3, we recreate the `adv_int` advice function from page 117 in the context of Java. This is ordinary advice behavior since the advice body invokes a join point with an argument that is increased by one, and return the result decreased by one. For the increase and decrease operations, we employ the `intValue` method from the `Number` interface, followed by an integer arithmetic operation. As a consequence, the increase and decrease operations accept any `Number` object and return an `Integer`. (The operations are not strictly an increase and decrease, but rather a rounding of a number to the next or previous integer value.)

```
1  public class BeforeAfterAdvice<Iu> implements GenAdvice<Object,Iu> {
2      public void before(Iu arg) {}
3      public void after(Iu arg) {}
4
5      public final <O,I extends Iu> O around(I arg, JoinPoint<O,I> proceed) {
6          before(arg);
7          try { return proceed.invoke(arg); }
8          finally { after(arg); }
9      }
10 }
```

Listing 8.4: Redefinition of `BeforeAfterAdvice` as generic advice in typed EcoSys

In the advice declaration, note that the signature of the advice method is the most general signature that can be given to an ordinary method with this advice body. An instance of this advice is accepted by the `bind` method of an instance of any of the following types:

`Pointcut<Integer,Integer,JP>`,
`Pointcut<Integer,Number,JP>`,
`Pointcut<Number,Integer,JP>` and
`Pointcut<Number,Number,JP>`

Other pointcut types with type arguments outside of the bounds `Integer` and `Number` are not accepted, and we can demonstrate that the advice behavior is indeed unsafe for such cases. For example, a first type argument `? extends Number` is not safe since the advice does not provide a result that is guaranteed to be an instance of some unknown subtype of `Number`, and a second type argument `? extends Number` is not safe either since the advice does not invoke the intercepted join point with an argument that is guaranteed to be an instance of some unknown subtype of `Number` (*e.g.*, the argument is not a `Float`).

Example (Generic advice). In Listing 8.4, we recreate the `BeforeAfterAdvice` from Listing 4.1 on page 39 using typed EcoSys. This advice executes the join point unmodified, but includes configurable behavior before and after this execution. Since the original join point argument and result are retained by the advice, it is possible to represent the corresponding join point types as type variables (*i.e.*, this is generic advice behavior). For the result we employ the most general upper bound `Object`, but since the before and after behavior may access the join point argument, we employ the variable type `Iu` as the upper bound for the argument type. This variable type is itself provided as a type argument to the generic class `BeforeAfterAdvice`.

As a consequence of the interface declaration of `BeforeAfterAdvice`, an instance of *e.g.*, the type `BeforeAfterAdvice<Number>` may be bound to pointcuts where the join point argument is guaranteed to be a `Number`, for example `Pointcut<?,? extends Integer>`.

8.2.2 Integration of Typed Pointcuts

In the above, we explain that we assume that only pointcut entities can be obtained where the matching behavior of the pointcut corresponds to the type information given in its type

parameters. We now briefly discuss how this correspondence is realized in case of the pointcut entities provided by the ECOSYS implementation presented in Section 4.3.2.

Recall that these pointcut entities are configured with instances of class `Class`, a type from the JAVA reflection facilities which represents information about a type at run-time. An instance of this class can be considered a class token and the class loader ensures that the same type is always represented by the same class token. As explained by Naftalin and Wadler (2006, Sec. 6.1), one of the changes of JAVA 5 is this class now takes a type parameter, so that the class token for type T now has type `Class<`T`>`. Class literals and the invocations of the `getClass` method of the `Object` type are treated specially by the compiler such that it is possible to write the first and last of the following assignments without employing cast operations:

```
Class<Integer> ki = Integer.class;
Number n = new Integer(42);
Class<? extends Number> kn = n.getClass();
```

The advantage of having the type represented by a class token reified in the type parameter of the token's own `Class` type, is that this type information may be employed in the signature of the reflection operations specified in the interface of type `Class`.

However, we may also employ this type information in the interface that is offered to create pointcut entities. For example:

```
static <CT,RT> Pointcut<? super RT, ? extends CT, AJJP> methodExecution(
    int modifierPos,
    Class<CT> receiverClass,
    Class<RT> returnType,
    String methodPat);
```

This method execution pointcut will expose the receiver as a context argument. It matches those join points where the receiver class is a subtype of the type represented by the second argument, and the return type is a supertype of the third argument. This information is represented in the type arguments of the parameterized `Pointcut` return type of this method.

8.2.3 Typed First-Class Deployment Procedures

In general, it is straightforward to program *typed* variants of first-class deployment procedures in the typed ECOSYS system. In order to demonstrate this, we will revisit the first of the examples presented in Section 4.2 using the untyped variant of ECOSYS. The other examples may be treated in a similar fashion.

The example concerns a deployment specification that is shared between different quality of service concerns in the system; all of these concerns have advice behavior that needs to be bound to the same pointcut (`modelManip`), configured with the same log handle and resolution behavior, and so on. The shared deployment behavior is specified as a method `deployQoS`, which now becomes:

```
void deployQoS(OutputAdviceFactory<FigureElement> factory) {
    OutputAdvice<FigureElement> adv =
        factory.createAdvice(Application.getLog());
    adv.setOutputLevel(Logging.INFO);
    core.deploy(modelManip.bind(adv));
    core.addResolution(new AdvicePrecedenceResolution(authorization,adv));
}
```

In this code, we have assumed that the `modelManip` pointcut exposes the manipulated model objects, which are of type `FigureElement`. The pointcut therefore has type:

```
Pointcut<Void, ? extends FigureElement, AJJP>
```

Additionally, we assume that `OutputAdviceFactory<T>` is a generic type of factories which produce instances of type `OutputAdvice<T>`, which is a subtype of `BeforeAfterAdvice<T>`, which, in turn, is shown in Listing 8.4 to be a subtype of:

```
GenAdvice<Object,T>.
```

The compatibility between the above `Pointcut` type and `GenAdvice` type is verified when typing the `modelManip.bind(adv)` expression inside the deployment procedure. The compiler will deduce that this invocation is well-typed according to the first `bind` method in Listing 8.2.

While the above example is a deployment procedure which concerns specific types only, we observe that it is possible in typed EcoSys to construct *generic* deployment procedures as well. This is demonstrated in a generalization of the above example, where the employed pointcut has become a parameter of the deployment procedure:

```
<T> void deployQoS(OutputAdviceFactory<T> factory,
                   Pointcut<?, ? extends T, AJJP> pc) {
    OutputAdvice<T> adv = factory.createAdvice(Application.getLog());
    adv.setOutputLevel(Logging.INFO);
    core.deploy(pc.bind(adv));
    core.addResolution(new AdvicePrecedenceResolution(authorization,adv));
}
```

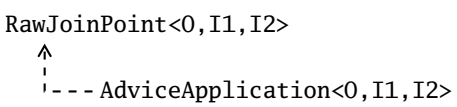
Now, `deployQoS` is a generic method with the type of the exposed object as a type parameter `T`. The signature of this method requires that an `OutputAdviceFactory` is combined with a `Pointcut` that involves the *same* type `T`. Otherwise, the combination would not be guaranteed to be type-safe, and consequently, the body of the deployment procedure would be rejected by the compiler.

8.3 Typed Interaction Resolutions

The interaction resolution mechanism that we present for EcoSys in Section 4.1.3 employs general `Object` types (by means of the `RawJoinPoint` interface, as is presented in Figure 4.2 on page 44). It is clear that these powerful interaction resolutions may cause run-time type errors

as well, so the question naturally arises whether we can install a typing discipline for these resolutions. This is not supported in the current version of ECOSYS and it seems that at least a slightly different organization is needed in order to do so. Below, we provide a brief provisional evaluation of the problem.

First, recall that resolutions compose the behavior of advice applications obtained using different pointcuts. This is logical since interactions may very well occur between advice instances that employ different pointcuts to advise the same join point. This organization entails that compatibility of a resolution should be directly considered against a `RawJoinPoint`, rather than the more high-level abstractions of the types `JoinPoint` and `Pointcut`. To this end, it would be possible to equip the types `RawJoinPoint` as well as the type `AdviceApplication` with type variables representing the types of the relevant join point (if we assume a variable `O` for the type of the result, `I1` for the type of the caller, and `I2` for the type of the receiver):



The type `Resolution` would also need to carry some type information which would be used to specialize the `AdviceApplication` that it operates on with some type arguments. Correspondingly, it would be necessary only to invoke a registered `Resolution` when a *compatible* join point is encountered (currently, registered `Resolution` instances are considered for *every* join point). This would need to be considered when a `Resolution` instance is registered in the `Core`.

Alternatively, we observe that a wide class of useful interaction resolutions may be carried out without specific knowledge of the type of the join point at hand, and are therefore compatible with any join point. For example, resolutions that only rearrange, duplicate or remove existing advice applications will never produce a failing advice composition, if the list of advice application they operate on is safe for the current join point to begin with. In order to track that a resolution do add advice applications in the typing, it seems that wildcards may again be employed. Consider the following `Resolution` type:

```
interface SafeResolution {
    void resolve(List<AdviceApplication<?,?,?>> advs);
}
```

Since the type arguments of `AdviceApplication` are unknown (as represented by unbounded wildcards), it is essentially impossible from an implementation of `resolve` to produce a new advice application that it may add to the list. However, it is still possible to rearrange, filter or duplicate the existing advice applications[2].

[2]Although for some of these operations, it may be necessary to employ the mechanism of wildcard capture to assign type variables to the different types represented by the ?, as explained in Section 6.3.2.

```
1  import java.util.*;
2
3  class Caching implements MixAdvice1<Number,Number,DataProvider> {
4      Map<DataProvider,Number> cache = new HashMap<DataProvider,Number>();
5
6      <C extends DataProvider>
7      Number around(C ctx, JoinPoint<Number,C> proceed) {
8          Number result = cache.get(ctx);
9          if(result == null) {
10             result = proceed.invoke(ctx);
11             cache.put(ctx,result);
12         }
13         return result;
14     }
15 }
```

Listing 8.5: Typing of caching advice using typed EcoSys

8.4 Some Real-life Examples

In this section, we present a number of examples of common aspect applications expressed using typed EcoSys. Beside enabling a better understanding of the proposed typing constructs, these examples also illustrate the usefulness of the introduced mechanisms for practically-relevant, realistic advice behavior. They provide a reasonable indication that the proposed typing schemes are sufficient to support most common application of pointcuts and advice (in particular around advice).

To categorize the demonstrated advice, we use some terminology of Rinard et al. (2004). This work distinguishes between *augmentation advice* (which always executes the original behavior entirely), *narrowing advice* (which either executes the original behavior or raises an error) and *replacement advice* (which replaces the original behavior with entirely new behavior).

Caching Caching is a common example of a concern that can be implemented using aspects (see *e.g.*, Colyer, 2004). In Listing 8.5, we show a straightforward caching aspect that stores the numeric return value of an expensive operation in a map, and then retrieves the cache value on subsequent invocations. The `DataProvider` instance that is associated with the operation is used as the key to access the cache values.

When no cache value is available, the advice behaves as an augmentation advice that stores the original return value. Otherwise, it is a replacement advice that directly returns a value without executing the original behavior. Since a result from a proceed invocation may be written to the cache, the proceed result type should be a subtype of `Number`. And since a value read from the cache may be used as a result of the advice, the advice result type should be a supertype of `Number`. As a result of the interface declaration of the class `Caching`, an instance of this advice may be bound to a pointcut with a result type that is exactly `Number`. Indeed, in case of a more specific join point return type, the cache value may not be too general for the join point client.

```
1  import java.awt.Component;
2  import javax.swing.JTextArea;
3  import javax.swing.JScrollPane;
4
5  class Factory {
6      Component createTextArea(String t) {
7          return new JTextArea(t);
8      }
9
10     ...
11 }
12
13 class Decorator implements MixAdvice1<JScrollPane,Component,Object> {
14     <I> JScrollPane around(I arg, JoinPoint<Component,F> proceed) {
15         return new JScrollPane(proceed.invoke(arg));
16     }
17 }
```

Listing 8.6: Factory Method design pattern with decorator in typed ECOSYS

And in case of a more general join point return type, the result of the join point is too general for the cache.

With respect to the argument, the original argument from the join point client is always employed to invoke the join point, although the original join point is not always invoked. This is the behavior of narrowing advice.

Factory Method Pattern In previous research, it has been recognized that the implementation of a number of common design patterns benefit from the application of aspect-oriented programming[3]. In Listing 8.6, we provide an example of the *Factory Method design pattern* (Gamma et al., 1995). The intent of this pattern is to create an interface for object creation that defers instantiations to its specializations. Our example defines a factory that creates GUI components (only one factory method is shown). An aspect specializes the factory methods to decorate the created components with scrollbars.

In this case, the aspect performs replacement advice that returns a newly created component. Since the invocation of the constructor of `JScrollPane` requires a `Component` while it provides an instance of `JScrollPane`, we employ these types as respectively the proceed and advice return types. Consequently, the advice can be bound to pointcuts where the first type argument has upper bound `Component` and lower bound `JScrollPane`. The original join point

[3] As a matter of fact, the aspect-oriented implementation of design patterns is sometimes used as case study for the evaluation of the expressiveness of aspect approaches (with regard to deployment and other features). Systematic investigations of all of the design patterns from Gamma et al. (1995) have been conducted by Hannemann and Kiczales (2002), Hirschfeld and Lämmel (2004), Rajan (2007) and Miles (2004), while Ostermann and Mezini (2003) considers improvements to the implementation of one design pattern. Such a study is a part of the future work for (typed) ECOSYS, and at present we limit ourselves to the discussion of the typing of one design pattern example.

```
1  import java.util.*;
2
3  class Profiling implements GenAdvice<Object,DataProvider> {
4      Map<DataProvider,List<Long>> timings =
5          new HashMap<DataProvider,List<Long>>();
6
7      List<Long> getEntry(DataProvider key) {
8          List<Long> entry = timings.get(key);
9          if(entry == null) {
10             entry = new LinkedList();
11             timings.put(key,entry);
12         }
13         return entry;
14     }
15
16     <R, DP extends DataProvider> R around(DP dp, JoinPoint<R,T> proceed) {
17         long start = System.currentTimeMillis();
18         try { return proceed.invoke(dp); } finally {
19             long stop = System.currentTimeMillis();
20             getEntry(dp).add(stop - start);
21         }
22     }
23 }
```

Listing 8.7: Typing of profiling advice using typed EcoSys

argument from the join point client is always directly employed for the proceed invocation (*i.e.*, augmentation advice), so there are no restrictions with respect to the possible argument types.

Note that in case the specialized factory would *refine* the existing `Component` instead of creating a new value (*e.g.*, a border can be defined for an existing component using the `setBorder` method of class `JComponent`), the advice would qualify as augmentation advice. It would also become possible to specify the advice behavior as an instance of `GenAdvice`.

Profiling Profiling is another example of a crosscutting concern that is often implemented using aspects (see *e.g.*, Laddad, 2003, sec. 5.6.2). Listing 8.7 presents profiling advice that measures the execution time of operations and stores the measurements in a list that is associated with the `DataProvider` join point argument. As can be expected of profiling behavior, this is purely augmentation advice. The join point argument should at least be a `DataProvider`, so this type is employed as the upper bound for the corresponding type variable. No specific bound is employed for the return types.

Chapter 9

StrongAspectJ: Recovering Mainstream AOP Type Safety

In this chapter, we apply the typing principles from Chapter 7 to the mainstream aspect language ASPECTJ. The result is a concrete proposal for a language extension, called STRONGASPECTJ[1]. This contribution transfers the results of Part II of the dissertation to conventional aspect languages: STRONGASPECTJ offers safe aspect deployments in the context of traditional deployment mechanisms; it does not provide the advanced deployment expressiveness that is present in ECOSYS. The motivation for this development is to underline the independence of the contributions from Chapter 7, and to make the results directly adoptable by aspect languages that wish to retain the traditional deployment mechanisms.

Another motivation for STRONGASPECTJ is the status of type systems in current aspect languages. Since ASPECTJ is designed as a seamless extension of the JAVA programming language, it already offers a type system directly inspired by JAVA. This type system has also been adopted by other aspect languages that employ the pointcut/advice mechanism, such as JASCO and CAESARJ. It may seem a little odd that we present a new type system when there appears to be a *de facto* standard for aspect typing, especially since the STRONGASPECTJ type system is somewhat more complex than the existing solution. However, contrary to the impression given by this situation, aspect typing is not a solved problem. The ASPECTJ type system exhibits significant safety problems that STRONGASPECTJ overcomes, and hence the title of this chapter. The fact that the ASPECTJ type system nonetheless enjoys widespread use indicates that these issues are relatively unknown.

Section 9.1 presents the complete STRONGASPECTJ proposal and discusses how it integrates the typing principles from this dissertation with the ASPECTJ language. Section 9.2 compares the result to the existing ASPECTJ type system and explains the important improvements realized by STRONGASPECTJ. Section 9.3 presents an implementation of the STRONGASPECTJ proposal in the context of the ASPECTBENCH compiler. Finally, Section 9.4 discusses the related work in the area of typed aspect languages.

[1] The "strong" prefix hints at the safety guarantees provided by the extension. The name is also loosely inspired by the STRONGTALK type system for SMALLTALK from Bracha and Griswold (1993).

9.1 StrongAspectJ

StrongAspectJ integrates the typing principles for pointcut and advice bindings with the AspectJ language. We first discuss a number of particular design decisions that are present in the current type system of AspectJ and that we choose to retain. We then present the StrongAspectJ extension with a complete language definition and a discussion of the relation to the typing principles developed in Chapter 7. Afterwards, the proposal is illustrated with some examples.

9.1.1 AspectJ Typing Particulars

Similar to the typing of EcoSys, the safe application of advice is checked in AspectJ through three consecutive steps: (i) the body of an advice method must adhere to its signature, (ii) the join points selected by a pointcut must adhere to the pointcut type, (iii) when advice is bound to a pointcut, the advice signature and pointcut type must be compatible. While the first of these steps is checked in a way that is almost identical to ordinary Java, and the approach for the third step is quite similar to typed EcoSys, the type system of AspectJ has an alternative design for the second step, both with regard to the arguments and the result of join points. This will be described in detail in this section.

StrongAspectJ adopts those aspects of the design of AspectJ described in this section, if for no other reason than to keep the distance between both languages minimal. This does not mean that these design decisions are undisputed; we will discuss both the advantages and the disadvantages of the AspectJ design here.

Typing and Binding of Context Arguments

An AspectJ pointcut may expose context parameters of the selected join points through the use of the primitive pointcut designators **`this`**, **`target`** and **`args`**. These designators bind a pointcut or advice variable to respectively the caller object, the receiver object or the arguments of the join point. The type of the exposed context arguments is an important part of the pointcut type and advice signature. However, rather than checking that the join points matched by the pointcut indeed have context objects that match the declared types and signaling an error when this is not the case, AspectJ will adapt the pointcut matching semantics such that join points with context objects of an incompatible type are not selected.

For example, the pointcut expression **`this`**`(e)`, where `e` is the name of an `Employee` variable from the enclosing declaration, will only match join points where the executing object has this particular type `Employee`. This is a very straightforward design to ensure that the variable can always be bound to an object of the appropriate type. The main disadvantage, however, is that mistakes in pointcut expressions are not detected by the type system. If we want to bind the receiver `Employee` object in the pointcut **`call`**`(* Employee.*(..))`, but by mistake employ a combination with the expression `this(e)` (which binds the caller object instead), no error is reported. Instead, the pointcut will be changed to match only those cases where an `Employee` method is called from within its own class. This seemingly mysterious change in the matching semantics of the original pointcut may be difficult to track down.

We remark that the type comparison for **this**, **target** and **args** pointcut designators involves the *dynamic* type of the to-be-bound object. If necessary, a code *residue* with an **instanceof** test is woven into the static code location that correspond to the join point (the *join point shadow*) to be able to include those join points where the static type of the to-be-bound variable or expression is not sufficient to determine type compatibility. This is in contrast to type patterns that are normally used in the pointcut language, and that match against static types of interfaces and signature declarations. Since a dynamic type test is useful in its own right, **this**, **target** and **args** may alternatively be used without binding a variable. For example, the pointcut **this**(Employee) will only match join points where the caller object is an instance of Employee.

Typing of the Join Point and Advice Result

The other important part of the pointcut type and advice signature is the return type of the intercepted join points. However, **before** advice methods do not have access to the join point return value, simply because the join point has not been executed yet. Ordinary **after** advice cannot access the join point result either, because it is used in both the case of a normal and an abnormal completion of the join point.

In case of an **after returning** advice method, a result *is* always available — at least for a join point that can logically have a return value (*e.g.*, not for a field set or the invocation of a **void** method). To bind the join point result to an argument for this kind of advice method, ASPECTJ does not employ a binding primitive pointcut designator such as **this**, **target** or **args**. The result can be made available to the advice body by simply declaring an (additional) argument in a section of the advice method signature after the **returning** keyword. Similar to the binding with the **this**, **target** and **args** primitives, this implies a condition for the execution of the advice (in addition to the conditions from the pointcut): the advice will only be applied when the result has the type that is declared for this parameter (again, the *dynamic* type of the result is considered, and a dynamic **instanceof** test is used to guard the advice execution when the type relation cannot be statically determined). This design has roughly the same trade-offs as the parameter binding mechanism that we discussed in the previous section.

This leaves only **around** advice methods, which may access the join point result through an invocation of the **proceed** method. Contrary to the case of **after returning** advice, it is not possible to apply the advice only on the condition that it is compatible with the dynamic type of the join point result: since the join point has not been executed yet when the advice is applied, this result is not yet known. In addition, such an advice method returns a new result that is provided to the join point client instead. Obviously, the type system needs to ensure that this new result does not violate the expectations of possible join point clients.

Contrary to the case for join point arguments, ASPECTJ opted for a design where type information about the join point result is not included in the pointcut type. This information would complicate the pointcut types while it is only used for **around** advice and not the other advice kinds. Instead, when the weaver looks up the join point shadows that are matched by the pointcut expression to which the advice is bound, the advice compatibility is verified using the static type information of the join point shadow (*i.e.*, the advice signature and the join point type are compared directly, without the intermediate step of a pointcut type). The major disadvantage of this approach is that the type checking of a pointcut/advice binding

becomes dependent on the base application. The incompatibility of some pointcut and advice combinations may only be detected when the code base includes join points that 'trigger' conflicting situations. In addition, since the weaving of aspects in the base code is increasingly delayed from the compile-time to later stages in the deployment of the application (load-time or run-time), errors may be detected much later than with traditional static type checking.

9.1.2 Language Definition

The StrongAspectJ language extension is defined by means of a syntax definition in Figure 9.1 and a number of (informal) semantic rules in Figure 9.2. The extension incorporates the typing principles for function join points from Section 7.3 into the AspectJ language. We will discuss the definition of the extension in detail in the following sections.

Pointcut type ranges

As explained in Section 7.3.1, a join point in the context of AspectJ may be considered a function that receives a number of arguments and returns a result. Analogously, the signature of the join point corresponds to a function type. We write such a signature $S^o_{jp}(\overline{S^i_{jp}})$, where S^o_{jp} is the return type of the join point and $\overline{S^i_{jp}}$ are zero or more argument types[2].

In Section 7.3.2, it is proposed to attach type information to pointcuts that consists of zero or more type ranges $\overline{U^i_{pc} - S^i_{pc}}$ for each of the join point arguments, and a type range $S^o_{pc} - U^o_{pc}$ for the join point result. The contract of this type information is that the pointcut may only match join points with a signature between lower bound signature $S^o_{pc}(\overline{S^i_{pc}})$ and upper bound signature $U^o_{pc}(\overline{U^i_{pc}})$. Concretely, the following subtype relations should hold for the join point signature $S^o_{jp}(\overline{S^i_{jp}})$:

$$S^o_{pc} <: S^o_{jp} <: U^o_{pc} \qquad \overline{U^i_{pc} <: S^i_{jp} <: S^i_{pc}}$$

(Recall that the lower bound and upper bound for the argument types are exchanged because the subtyping rule for function types is contravariant for the argument types.)

In StrongAspectJ, type ranges are declared for the argument binding pointcut primitives and for the arguments of named pointcuts, as defined in Figure 9.1. (Return types are not included in the pointcut type in StrongAspectJ, the type relations for return types are verified for the join points directly, as described below.) We remark that, contrary to the definition from Gosling et al. (2005, §4.1), the null type (the type of **`null`**) is denotable in StrongAspectJ by means of the reserved name `Null`. The direct supertypes of the null type are all reference types other than the null type itself (Gosling et al., 2005, §4.10.2). `Null` is therefore particularly useful as the lower bound of a type range, to avoid bounding the lower side of the range: *e.g.*, the range `Null-Number` includes all subtypes of `Number`, without restriction.

As explained before, StrongAspectJ will only match join points where the types of the arguments are compatible with the declared type ranges for the binding primitives. This matching behavior is defined in rule *binding primitives* in Figure 9.2. This rule is equivalent to the case of the argument type in rule FMatch in Figure 7.3 on page 116. Remark however that

[2] In Chapter 7, the various types S_{jp} stands for the *most specific* types that may be assigned to the join point, while T_{jp} is any supertype of S_{jp} that the join point may receive through subsumption. In the context of Java however, subsumption is built into the conditions of type rules and expressions are only assigned a single type.

Lexical metavariables

c	*class names (incl.* `Object`, `Null`*)*
p	*pointcut names*
x, y	*term variables (incl.* **`this`***)*
X, Y	*type variables*

Type expressions

S, T, U	::=	$\ldots \mid c\texttt{<}\overline{T}\texttt{>} \mid X$	*(*JAVA *types)*

Term expressions, statements, pointcut expressions

e	::=	$\ldots$	*(*JAVA *terms)*
	$\mid$	$\texttt{proceed(}\overline{e}\texttt{)}$	*proceed invocation*
M	::=	$\ldots$	*(*JAVA *statements)*
pc	::=	$\ldots$	*(non-binding primitives)*
	$\mid$	$p(\overline{x}) \mid \texttt{args(}\overline{U^i_{pc}\texttt{-}S^i_{pc}\ x}\texttt{)} \mid$ $\texttt{this(}U^i_{pc}\texttt{-}S^i_{pc}\ x\texttt{)} \mid \texttt{target(}U^i_{pc}\texttt{-}S^i_{pc}\ x\texttt{)}$	*binding primitives*
	$\mid$	$(pc\ \texttt{\&\&}\ pc) \mid (pc\ \texttt{||}\ pc) \mid \texttt{!}\,pc$	*combinations*

Member declarations

D	::=	$\ldots$	*(*JAVA *members)*
	$\mid$	$\texttt{pointcut}\ p\texttt{(}\overline{U^i_{pc}\texttt{-}S^i_{pc}\ x}\texttt{)}\texttt{:}\ pc\texttt{;}$	*named pointcut*
	$\mid$	$\texttt{before(}\overline{T^i_{adv}\ x}\texttt{)}\texttt{:}\ pc\ \texttt{\{}\ \overline{M}\texttt{;}\ \texttt{\}}$	*advice methods*
	$\mid$	$\texttt{after(}\overline{T^i_{adv}\ x}\texttt{)}\texttt{:}\ pc\ \texttt{\{}\ \overline{M}\texttt{;}\ \texttt{\}}$	
	$\mid$	$\texttt{<}\overline{X\ \texttt{extends}\ U}\texttt{>}\ T^o_{adv}\ \texttt{around(}\overline{T^i_{adv}\ x}\texttt{)}\texttt{:}\ pc\texttt{:}$ $T^o_{pro}\ \texttt{proceed(}\overline{T^i_{pro}}\texttt{)}\ \texttt{\{}\ \overline{M}\texttt{;}\ \texttt{\}}$	

Figure 9.1: STRONGASPECTJ syntax definition (relevant parts). The figure employs the notational conventions from Chapter 6 and Chapter 7. Additionally, we write $\overline{e}$ for a repetition of zero or more element $e_1, \ldots, e_n$, where the element separator may be space, comma or semicolon, depending on the context. The parts omitted from this figure keep the original JAVA definition. Syntactic sugar not shown in the syntax definition, is the removal of the empty angle brackets "<>" when no type variables are declared or no type arguments are provided. Non-binding pointcut primitives are also omitted since they have the same syntax as in ASPECTJ. The declarations from category D may be placed inside top-level class and aspect declarations according to the rules from ASPECTJ.

Advice declarations (*cf.* the last three clauses of D in Figure 9.1)

Type use *(around only)* The identifiers $\overline{X}$ may be used in the entire advice declaration (excluding the type parameter section itself) to refer to variable types that have the declared bounds $\overline{U}$ as their direct supertypes (§4.10.2). All employed types must be primitive types, known variable types or legal parameterized types (§4.5).

Variable signature types *(around only)* A type variable X_k may only appear in the advice signature as exactly one of the type annotations $\{T^o_{adv}, \overline{T^i_{adv}}\}$. In that case, the corresponding annotation from $\{T^o_{pro}, \overline{T^i_{pro}}\}$ should also equal X_k. Similarly, X_k may only appear in the proceed signature when it appears in the corresponding position in the advice signature.

Parameter use The identifiers $\overline{x}$ may be used as simple names in the body statements $\overline{M}$ to refer to parameter variables of the declared types $\overline{T^i_{adv}}$.

Proceed use *(around only)* The fixed identifier **`proceed`** may be used for invocations in the body statements $\overline{M}$. The expressions used as arguments to an invocation must have types assignable (§5.2) to $\overline{T^i_{pro}}$. The invocation itself has type T^o_{pro}.

Body return value For an around advice declared with non-**`void`** return type T^o_{adv}, it is not allowed to drop off the end of the body statements $\overline{M}$, and every **`return`** statement must have an expression of some type that is assignable (§5.2) to T^o_{adv}. For before or after advice, or around advice that is declared **`void`**, the body may only contain **`return`** statements without an expression.

Pointcut usage (*cf.* the last four clauses of D in Figure 9.1)

Type range containment *(auxiliary, cf.* §4.5.1.1*)* A type range S–U is said to contain type range S'–U' when S is assignable to S' and U' is assignable to U. Type range S–U is said to contain type T when S is assignable to T and T is assignable to U. (The assignment conversion is defined in §5.2.)

Parameter binding All declared advice or named pointcut parameters $\overline{x}$ must be bound (*i.e.*, used as an argument) exactly once in each disjunctive branch of the employed pointcut expression *pc*.

Pointcut parameter type *(named pointcut only)* When a named pointcut parameter x_j is used as argument in a pointcut expression, the argument position type range must be contained in type range $U^i_{pc,j}$–$S^i_{pc,j}$.

Advice parameter type *(before and after only)* When an advice parameter x_j is used as argument in a pointcut expression, the upper bound of the argument position type range must be assignable (§5.2) to $T^i_{adv,j}$.

Advice parameter type *(around only)* When an advice parameter x_j is used as argument in a pointcut expression, the argument position type range must be contained in range $T^i_{pro,j}$–$T^i_{adv,j}$, if $T^i_{pro,j}$ and $T^i_{adv,j}$ are non-variable, or must be contained in range `Null`–U_k, if $T^i_{pro,j}$ and $T^i_{adv,j}$ both equal type variable X_k with bound U_k.

Advice return type *(around only)* The return type of join point shadows where an advice is woven, must be contained in range T^o_{adv}–T^o_{pro}, if T^o_{adv} and T^o_{pro} are non-variable, or must be contained in range `Null`–U_k, if T^o_{adv} and T^o_{pro} both equal type variable X_k with bound U_k.

Pointcut matching (*cf.* the clauses of *pc* for **`this`**, **`target`** and **`args`** in Figure 9.1)

Binding primitives Primitive pointcuts **`this`**, **`target`** and **`args`** match when the declared lower bound type U^i_{pc} is assignable (§5.2) to the compile-time type of the to-be-bound variable or expression, and its run-time type is assignable to the upper bound S^i_{pc}.

Figure 9.2: StrongAspectJ typing and matching rules. Together with the Java language specification by Gosling et al. (2005), these rules define the language semantics. The (§) section references refer to specific definitions from that document.

the *dynamic* type of the join point argument (rather than the static type) is employed for the comparison with the upper bound, as is usual in AspectJ.

Advice signature, type variables

In Section 7.3, we model function advice as a function that receives a function join point and returns a new function join point. As can be observed in Figure 9.1 and Figure 9.2, around advice methods in StrongAspectJ function equivalently, but use a more direct formalism. The advice declares a number of ordinary arguments and a return value that are enforced for the advice body by rule *parameter use* and rule *body return value* respectively. The advice also declares a signature for the intercepted join point with the fixed identifier **proceed**, the usage of which is enforced by rule *proceed use*. We can observe that this is indeed equivalent to the model from Section 7.3, since after binding the **proceed** identifier to a join point, the remainder is an ordinary function that may be used in place of the original join point.

Further recall that two kinds of advice functions are described for advising function join points in Section 7.3. Ordinary advice (Section 7.3.3) is typed using subtype polymorphism, while parametric polymorphism and type variables are used for generic advice (Section 7.3.4). A typing using ordinary advice is useful for advice behavior that replaces the join point arguments or results with values of a sub- or supertype, while a typing using generic advice is useful for advice behavior that retains the original arguments or result. It is also explained that it is possible to mix both forms within one advice function: for example, an advice may replace an argument value while it retains the original result value (or vice versa).

As shown in Figure 9.1, StrongAspectJ employs one general form of around advice that integrates both ordinary and generic advice. The advice may declare a number of type variables; their usage is identical to that of type variables declared by standard generic methods in Java (see rule *type use* in Figure 9.2). (Note that in contrast to the type variables in Chapter 6 and Chapter 7, only an upper bound may be defined for type variables.) In particular, type variables may be employed as a type annotation for an argument or a result in the signature defined by the advice method for itself or for **proceed**, which allows to declare generic advice.

For this case, rule *variable signature types* stipulates that a type variable may be used at most once as such a type annotation, and that it must then appear in the same position in both signatures. This allows to treat the advice method as a generic advice with respect to a particular argument or its result, but enforces the proper form for generic advice. The type variable is bound to the type of the particular join point at hand, and the type variable should therefore not be otherwise constrained (this is explained in Section 7.3.4).

Before and after advice kinds in StrongAspectJ do not support the declaration of type variables and cannot be made generic. In general, type variables are only beneficial for advice methods to express type relations between the signature of the advice method and the signature of the intercepted join point. Since before and after advice methods do not have access to **proceed**, there is generally no need to define them as generic advice methods.

Type relations, pointcut/advice binding

Section 7.3.3 and Section 7.3.4 each derive compatibility conditions for the binding between a pointcut and an advice; the first for ordinary advice and the second for generic advice. These

compatibility conditions are specified in the form of a number of subtype relations, that we also apply here. It is additionally explained (in Section 7.3.4) that it is sound for function advice to mix both advice styles on the condition that the compatibility with pointcuts is verified using an appropriate combination of the compatibility conditions.

In StrongAspectJ, the compatibility of join point arguments is ensured through the combination of two measures. On the one hand, rule *parameter binding* requires that every parameter declared in the signature of a named pointcut or an advice method is always bound in the employed pointcut expression. On the other hand, rule *pointcut parameter type* and rule *advice parameter type* ensure that for every such binding, the type range associated with the binding position is contained in the type range declared for the parameter in the enclosing definition.

Specifically, the rule *advice parameter type* mandates different type relations depending on the kind of type annotation declared for the advice parameter being bound: non-variable or variable. In the first case, the rule ensures that an argument to **proceed** can be used by the join point, and that an argument from the join point client can be used by the advice. This correspond directly with the following conditions of rule FBind from Figure 7.3 on page 116:

$$\vdash T^i_{pro} <: U^i_{pc} \qquad \vdash S^i_{pc} <: T^i_{adv}$$

In the second case, the rule ensures that the join point argument type is a valid type value for the type variable, *i.e.*, that it fits within its bounds. This corresponds directly to the following conditions of rule FGBind from the same figure:

$$\vdash U^i_{adv} <: U^i_{pc} \qquad \vdash S^i_{pc} <: S^i_{adv}$$

(Note however that the type variable lower bound U^i_{adv} is necessarily the null type in StrongAspectJ, so the first of these conditions is always met.) In both cases, the range U^i_{pc}–S^i_{pc} must be contained within another type range. The rule *pointcut parameter type* therefore prevents this range to change to "wider" ranges as named pointcuts are used to bind pointcut arguments.

With respect to return types, rule *advice return type* specifies type relations directly for the join point return type (S^o_{jp}). Identical to rule *advice parameter type*, this rule distinguishes between a non-variable and a variable advice return type. In the first case, the employed type relations ensure that the result from the join point is valid for the advice body and that the result from the advice body is valid for the join point client. This corresponds to the remaining conditions from rule FBind combined with the third condition from rule FMatch:

$$\vdash T^o_{adv} <: S^o_{jp} <: T^o_{pro}$$

In the second case, the employed type relations ensure that the result type from the join point lies within the bounds of the corresponding advice type variable. This corresponds to the remaining conditions from rule FGBind, also combined with the third condition from rule FMatch:

$$\vdash S^o_{adv} <: S^o_{jp} <: U^o_{adv}$$

(Again, note that the type variable lower bound S^o_{adv} is necessarily the null type in StrongAspectJ.)

9.1.3 Examples

In order to demonstrate the syntax and typing rules of StrongAspectJ, we consider both an example of ordinary advice and of generic advice.

The first example recreates the ordinary advice behavior from Listing 8.3 on page 125. This advice rounds the argument value to the next integer, executes the intercepted join point and then rounds the result to the previous integer:

```
Integer around(Number n): pc(n): Number proceed(Integer) {
    return proceed(n.intValue() + 1).intValue() - 1;
}
```

In this case, the rule *advice parameter type* stipulates the pointcut must provide an argument n that is contained between `Number` and `Integer`. Similarly, rule *advice return type* stipulates that the advice may only be applied to join points with static return type between `Number` and `Integer`. This corresponds to the earlier conclusions for this example.

As a second example, we consider a generic advice which executes a join point with the original parameter value, provided that this parameter value is higher than 10:

```
<N extends Number> void around(N n): pc(n): void proceed(N) {
    if(n.intValue() > 10)
        proceed(n);
}
```

In this case, rule *advice return type* stipulates that the advice may only be applied to join points of **void** return type. With respect to the argument type, rule *advice parameter type* prescribes an argument type below `Number`.

9.2 Postmortem of Traditional Aspect Typing

Readers with detailed knowledge of the type system of the AspectJ language will have noticed that StrongAspectJ proposes somewhat more complex typing rules. StrongAspectJ may therefore appear unduly restrictive, or at least 'over-engineered'. However, the simplicity of AspectJ typing is deceptive: it is not a consistent abstraction of the capabilities of pointcuts and advice. In this section, we will demonstrate that the two main simplifications by AspectJ directly introduce loopholes in the type system. This is an important problem: as explained in Section 6.1.2, a type system without a safety guarantee is not considered very useful. The deficiencies of the AspectJ type system have sporadically been reported before. Following the discussion, we consider the other cursory reports of safety problems from literature.

9.2.1 Around Advice and Proceed Invocations

A cursory comparison of the StrongAspectJ and AspectJ type systems immediately reveals that the former employs type ranges for the typing of pointcut arguments and join point results, while the latter only tracks a singular type for these values. This corresponds with a difference

in the type relations that are checked for pointcut/advice bindings. For all advice kinds, both type systems will verify that the advice can operate with the context parameters from the join points, by enforcing the advice argument type as an upper bound for the corresponding type in the join points. In case of around advice, both type systems will additionally ensure that any result from the advice meets the expectations of possible join point clients, by enforcing the advice return type as a lower bound for the static join point return types. Thus far, there is no difference between the two approaches.

However, in case of around advice, StrongAspectJ programs declare an additional signature for the **proceed** invocations; the types from this signature are enforced as lower bounds in case of the arguments and as an upper bound in case of the result. AspectJ omits a signature for **proceed** invocations and these additional checks. Instead, it is defined that inside the body of an around advice, a **proceed** method may be invoked with a signature that is identical to the signature of the advice method itself (Kiczales et al., 2001a, sec. 3.6). This is problematic since it declares that intercepted join points may be invoked with whatever argument types the advice method can handle, and that they will provide a result of the same type the advice method can provide, while no such relation between the join points and the advice method is guaranteed. Since an upper bound for the join points argument type is known, it can be verified that the advice can handle the argument of any join point, but without a lower bound for the argument type of the join points, it is impossible to guarantee which argument type the join points can handle. Similarly, it is not possible to guarantee which result join points are guaranteed to produce without tracking a lower bound for their return types.

This treatment of **proceed** invocations causes a loophole in the AspectJ type system. As a concrete example, the following advice method is accepted by AspectJ's typing rules:

```
void around(Person p): execution(void *()) && this(p) {
    proceed(new Person());
}
```

While this advice behavior works correctly with any `Person` as an argument, it may incorrectly assume the same of the intercepted join points. The advice also intercepts invocations of methods of subclasses, e.g. `Employee.promote()`, in which case executing the join point with an argument of the general type `Person` will cause a `ClassCastException`.

An identical situation can also occur with respect to the return types, as demonstrated by the following advice method:

```
Integer around(): call(Number *()) {
    Integer i = proceed();
    // ...
}
```

Because this advice provides an `Integer` result, it can assume that its join points do so as well in AspectJ. However, the advice can also be applied to join points where the returned value is a different kind of `Number`, such as a `Float`, in which case a `ClassCastException` will occur when returning from the **proceed** invocation to the advice body.

9.2.2 Generic Advice and the Object Return Type

The documentation of ASPECTJ makes no mention of any special provision for the case of generic advice. ASPECTJ does not allow to declare type variables at the level of advice methods and it does not support a typing using parametric polymorphism as present in STRONGASPECTJ. Nevertheless, it is a very common practice to apply an around advice method to a heterogeneous set of join points. An advice may be valid for all of these join points because it always invokes the intercepted join point with an original argument from the join point client, or because it always returns a result obtained from the intercepted join point. However, such genericity characteristics are not tracked in the typing. So why don't ASPECTJ users experience a lack of typing expressiveness in such cases?

When an advice method is generic with respect to a join point argument, it turns out ASPECTJ can benefit from the loophole that was described above. For example, the following advice method simply invokes the intercepted join point with the original argument from the join point client:

```
void around(Object o): call(* *(..)) && args(o) {
    proceed(o);
}
```

The advice body may invoke **proceed** with any `Object`. However, in many cases the actual join point will expect an argument of a more specific type. Since we happen to employ the argument from the original join point client this poses no problems *here*, but as long as there is no guarantee that no arbitrary `Object` argument will be used, a sound type system needs to restrict this case. It then becomes necessary to track that the arguments type is retained in the advice behavior (as in STRONGASPECTJ for example) in order to safely support such cases of generic advice.

There is no such (dubious) luck in case of an advice method that is generic with respect to the result. If we consider an advice method that simply returns the result from the intercepted join point to the join point client, then this advice should not be accepted according to the ASPECTJ type relations that we have discussed above:

```
Object around(): call(* *(..)) {
    return proceed();
}
```

Indeed, the advice signature indicates that the advice body may return any `Object` value, which will not be sufficient for join points with a more specific return type, *e.g.*, `Integer`.

Remarkably, the ASPECTJ compiler does not reject the application of this advice method to such join points. The implementation employs an additional type rule: in case of an advice method declared with the `java.lang.Object` return type, the default type relation does not apply and the application to any join point is accepted. For lack of better means to indicate generic advice, this *ad hoc* exception for the `Object` return type is necessary to support advice behavior that is generic with respect to the join point result. It is a very common practice for ASPECTJ users to employ such generic advice and — consciously or unconsciously — depend on this typing behavior.

However, it is clear that the `Object` return type is not a necessary condition for generic advice. Cases of generic advice that do not declare the `Object` return type are still being restricted by the ASPECTJ type system. Consider the following advice method that keeping invoking the intercepted join point until the integer value of the result is smaller than the number 100:

```
Number around(): call((Integer || Float) *(..)) {
    Number n = proceed();
    while(n.intValue() > 100)
        n = proceed();
    return n;
}
```

The advice behavior will always return a result obtained from the intercepted join point and it is therefore valid for any join point with a return type that is a subtype of `Number` However, precisely the opposite (supertypes of `Number`) is enforced by ASPECTJ and the above advice application will be rejected. The workaround to enable this binding consists in declaring the `Object` return type for this advice and including explicit casts to `Number` at every invocation of **proceed**. This is tedious, and of course it does not guarantee static type safety either.

Perhaps even more important than this expressiveness problem, is the fact that such an exception to the standard typing relations introduces new safety problems. Special privileges are being granted to advice methods that have the `Object` return type, but there is no verification that they indeed have generic behavior. This measure to support generic advice therefore introduces a new loophole in the type system as demonstrated by the following advice method:

```
Object around(): call(Number *()) {
    return new Object();
}
```

The advice return a general `Object` value, which is insufficient when the join point client is allowed to expect a more specific `Number` instance. Because of the exception for `Object` return types, this is not rejected by the ASPECTJ compiler and causes a `ClassCastException` at run-time.

9.2.3 Other Accounts of the AspectJ Type System

To our knowledge, the type safety problem involving around advice and **proceed** invocations is first described by Wand, Kiczales, and Dutchyn (2004, Sec. 5.3), in the closing notes of their formal study of the semantics of the advice mechanism. They present ASPECTJ's typing policy for this case and illustrate its lack of soundness. Some time later, in the on-line discussion of an ASPECTJ bug report by Bodden, Isberg, and Colyer (2006), ASPECTJ developers independently reach the conclusion that **proceed** may give rise to a `ClassCastException` at run-time. As a consequence of this discussion, a short note stating this effect is added to the "Language Semantics" and "Implementation Notes" documents of the ASPECTJ distribution. At the time of writing, the bug is still marked unresolved in the bug database. In both of these cases, there is

no discussion of generic advice methods, nor is there any mention of the special treatment of the `Object` return type.

On the other hand, both Tatsuzawa, Masuhara, and Yonezawa (2005) and Jagadeesan, Jeffrey, and Riely (2006) present a loophole example that involves the use of the `Object` return type (and no **`proceed`** invocations). However, in both cases, this is simply to illustrate that typing problems in ASPECTJ exist, and there is no investigation of the root of this problem. It is also not explained that this problem does not exist for return types other than `Object`, and the problems related to **`proceed`** are not considered.

As a conclusion, we may say that while the deficiencies that we report in this section have been identified before, this always occurred in a cursory manner and outside of the main focus of investigation. Our work constitutes the first comprehensive discussion of the ASPECTJ typing rule and the related typing problems. At the same time, the STRONGASPECTJ language provides a complete solution for these issues.

9.3 An Implementation of StrongAspectJ

In order to gain further understanding of the applicability of the STRONGASPECTJ proposal as a mainstream AOP language, a prototype implementation has been realized as a plug-in for the ASPECTBENCH compiler (*abc*) by Avgustinov et al. (2005). Since the STRONGASPECTJ language retains the complete design of ASPECTJ except for some aspects of its type system, it is an obvious implementation path to modify an existing ASPECTJ compiler. The *abc* platform was specifically chosen over the standard ASPECTJ compiler (*ajc*) by Colyer et al. (2002), because *abc* promises easy and modular addition of new aspect language features through plug-ins, without the need to fork the current source tree of the compiler. Following *abc* conventions, the plug-in is named STRONGAJ.

The AspectBench Compiler An overview of the architecture of the ASPECTBENCH compiler is presented in Figure 9.3. *abc* is divided up in a front-end and back-end component. The front-end is managed by the extensible compiler framework POLYGLOT by Nystrom et al. (2003). POLYGLOT parses JAVA programs and performs complete semantic checking using a number of AST visitor passes. An *abc* extension augments POLYGLOT with the ASPECTJ language constructs and includes a number of compiler passes specific to the language. The back-end was designed specifically for *abc* and makes heavy use of the bytecode analysis and transformation library SOOT by Vallée-Rai et al. (1999). The back-end performs weaving at the level of "Jimple", one of SOOT's intermediate representations of JAVA bytecode which is suitable for a number of optimizations.

An important design decision involves the communication between the front-end and back-end. The compilation of an ASPECTJ aspect is realized by translating the body of advice methods and **`if`** pointcut tests to local JAVA placeholder methods, and by weaving invocations of the advice code and inter-type declarations in the other parts of the program. Although POLYGLOT is capable of a number of AST transformations, it is necessary to perform the actual weaving in the back-end, since this may affect the class files (bytecode) for which no AST representations are available. The front-end therefore performs semantic checking of the new source files (based on a class table of the entire program that already takes inter-type declarations into

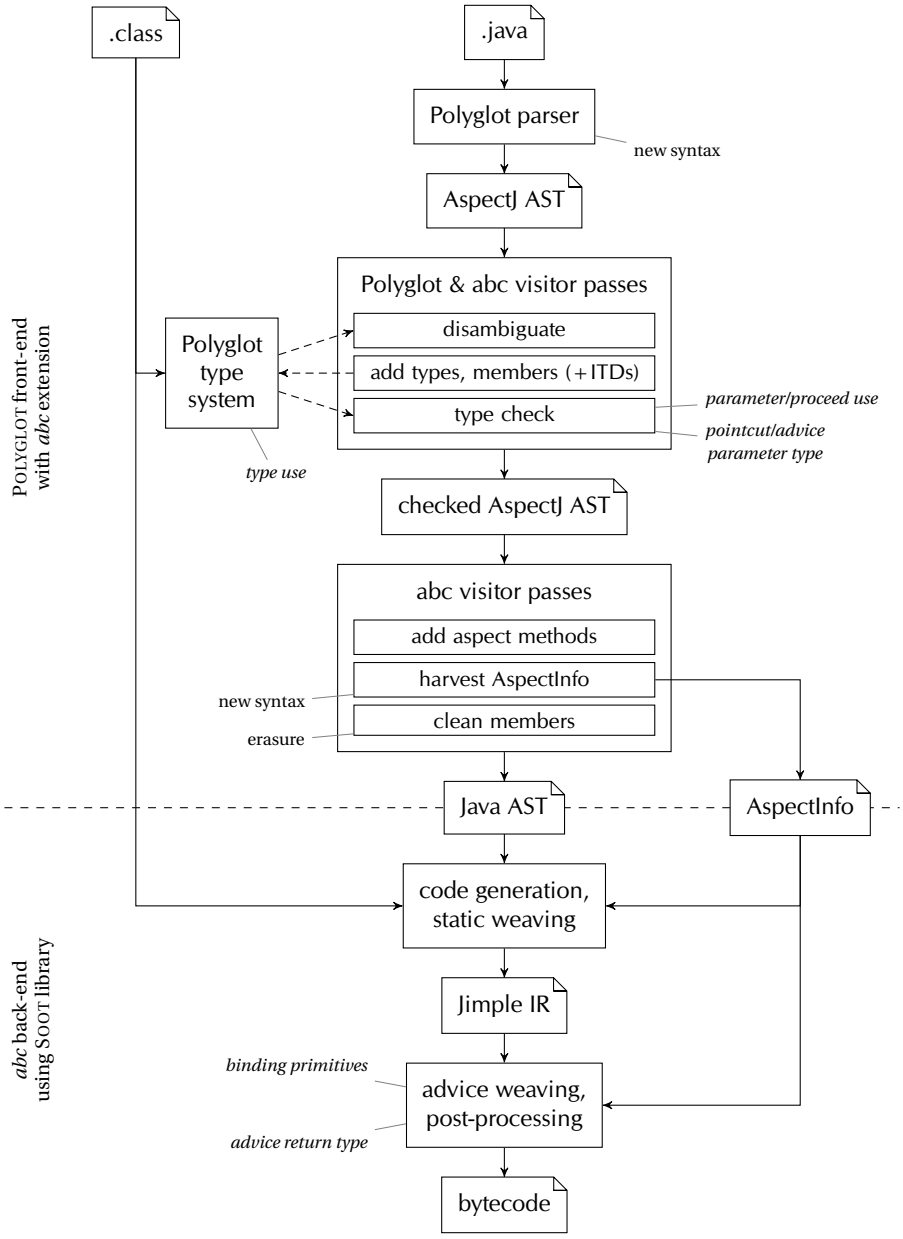


Figure 9.3: Simplified architecture of the ASPECTBENCH compiler. Additionally, the modification by the STRONGAJ plug-in are indicated by 'pin' annotations connected to the relevant boxes. Descriptions of modifications in *italic* refer to the names of semantic rules from Figure 9.2.

account) and prepares the compilation to standard JAVA bytecode by separating the ASPECTJ AST into a pure JAVA AST and an auxiliary "AspectInfo" structure with information about the ASPECTJ language constructs. The back-end performs a conversion of the JAVA AST to Jimple, and employs the AspectInfo to perform the static weaving and advice weaving.

The StrongAJ plug-in The *abc* platform provides a number of extension points that are employed by the STRONGAJ implementation to provide initial support for the STRONGASPECTJ language. This involves an extension of the parser and the addition or subclassing of AST nodes in the front-end to support the new (or changed) syntax elements (in this case, the pointcut type ranges and the new **proceed** signature specification). By making the new nodes reachable to the standard POLYGLOT AST visitors, ambiguities in the type nodes are automatically resolved by the framework. A subsequent visitor will type check nodes against the information from the type context. By installing the new **proceed**-signature in this context when the visitor enters the scope of an advice declaration, the standard visitor will check the advice body according to rule *parameter use* and rule *proceed use* from Figure 9.2.

Verifying the pointcut/advice bindings requires more effort, but in general we can locate the existing ASPECTJ type checks in the original AST nodes and override them in the subclasses to enforce the additional type relations that are present in STRONGASPECTJ (typically by adding a lower bound check in addition to an existing upper bound check). Pointcut arguments are still checked in the front-end, where the `typeCheck` method of the AST nodes of binding pointcuts is overridden to implement the modifications from rule *pointcut parameter type* and rule *advice parameter type*. Return types are checked while weaving, so it is required to transport the additional type information of the advice signatures to the back-end through the AspectInfo classes. The matching of **`this`**, **`target`** and **`args`** is also implemented in the back-end, where, depending on the corresponding static join point type, they either never match, always match, or construct a test residue. The residues are constructed by a number of classes that mirror the AST structure, and by overriding the residue construction in subclasses, the matching behavior from rule *binding primitives* is implemented. In order to employ the new implementations and to provide them with the declared type range of the variable being bound, a number of other back-end classes need to be subclassed as well.

Since the employed *abc* version (1.2.1) provides no support for Java Generics (nor for the other features of the JAVA 5 release from 2004), there is no direct support for type variables in the type system. Nevertheless, such a modification can be realized quite straightforwardly through the introduction of a new class `VariableType` as a subclass of the `ReferenceType` class from the POLYGLOT type system, equipped with a supertype link to the type variable's bound. This is sufficient to verify the usage of type variables according to rule *type use*. However, since the back-end cannot handle these types (type variables are not supported in JAVA bytecode), it is necessary to employ a new front-end visitor pass to erase type variables from the AST after type checking. The type variables are replaced by their respective bounds, identical to the *erasure* procedure for GJ that is discussed at the beginning of Section 6.3.1.

The current version of the StrongAJ plug-in[3] implements the complete STRONGASPECTJ proposal, as verified by a test suite of 62 static and 3 dynamic test cases. The implementation itself consists of a total of 54 classes/interfaces or about 2500 LOC.

[3] Available at `http://ssel.vub.ac.be/strongaj/`

9.4 Related Work: Typed Aspect Languages

9.4.1 Aspectual Caml

ASPECTUAL CAML by Tatsuzawa, Masuhara, and Yonezawa (2005) is an aspect-oriented extension of the functional programming language OBJECTIVE CAML. It includes two aspect mechanisms: a type extension mechanism and a pointcut/advice mechanism. The extension aims to integrate with the specific features of a statically-typed functional language such as OBJECTIVE CAML: type inference, polymorphic types, curried functions, *etc.* The typing of the pointcuts and advice functions is organized as follows. Pointcuts select join points through name and argument patterns, but are typed with type variables whose bindings are inferred from the advice functions to which the pointcuts are bound. The pointcut then only selects join points that match its typing.

Example. To discuss the typing mechanism of ASPECTUAL CAML in detail, we consider the example of an advice function for a simple evaluator of arithmetic terms:

```
advice eval_sub = [around call eval t]
    match t with
    | Sub (t1,t2) -> t1 + t2
    | _ -> proceed t
```

In this code, `call eval t` is a pointcut specification, where `eval` is a name pattern and `t` is a parameter pattern that binds the variable `t`. To determine the matching of this pointcut, ASPECTUAL CAML associates a function type to the pointcut, which prescribes the static type of the function whose invocation should be intercepted. This function type is determined from the advice body using a type inference mechanism that is common in the language: the process starts the polymorphic function type $\alpha \to \beta$ (where α and β are type variables), and based on the advice code, this is specialized to `exp` $\to$ `int` (since a destruction with constructor `Sub` indicates type `exp` and a result from the infix function `(+)` indicates type `int`). Consequently, the pointcut will match invocations of functions with name `eval` and type `exp` $\to$ `int`.

The inference mechanism allows to specify advice behavior entirely without type annotations. However, as a consequence, the matching behavior of the pointcut is dependent on the result of type inference process. This may lead to surprising results where tiny changes in the advice body cause different join points to be matched. In that case, explicit type annotations may still be preferred. In the case where the pointcut is assigned — explicitly or by inference — a polymorphic function type (such as $\alpha \to$ `int`) instead of a concrete type, the pointcut matches any function with a type that is a specialization of this polymorphic type. This may be used to quantify over join points with different types. In such a case, the advice body is guaranteed to handle join points with an unknown argument or result type.

As such, ASPECTUAL CAML is able to handle the definition of pointcuts and generic advice functions in a safe manner, and due to the type inference this may occur without specifying any type annotations. However, we remark that there is no subsumption in the functional context of OBJECTIVE CAML, only explicit coercion between structural subtypes. It is therefore unclear how the results translate to an object-oriented context, where we have indicated that subtype variance is important for advice functions. Also, the type inference algorithms (which are Hindley-Milner based in OBJECTIVE CAML) would not be directly applicable in this context.

9.4.2 PolyAML

Dantas, Walker, Washburn, and Weirich (2005) propose the PolyAML language as an extension of the ML family of languages. This is a statically-typed functional programming context that is very similar to the one of AspectualCaml. However, there are quite some differences between the languages. PolyAML includes only an advice mechanism for function invocations, and this is restricted to either before or after advice. Although this advice may not control the execution of the intercepted function, before advice may manipulate the function argument and after advice may manipulate the function result, providing some of the capabilities of around advice with a proceed facility.

An important aspect of the design of PolyAML is the choice for a first-class pointcut entities: pointcuts are ordinary values that may be passed to and returned by functions, allowing to build pointcut combinator functions. Two forms of pointcuts are supported: **any** (which matches every function invocation) and a list of functions names `{f,g}` (which matches the invocation of any of those functions). Contrary to AspectualCaml, the pointcut is not a pattern, and the referenced function must be in scope. The type of a pointcut expression consists of a pair of *polytypes* (one for the domain of the function and another for the range). Such a polytype is an ordinary type (or *monotype*) that may be quantified with type variables. Concretely, the type of pointcut **any** is `(all a.a,all a.a)` (where a is a type variable), and pointcut `{f,g}` admits a type where the domain (*resp.* range) is more general than the domain (*resp.* range) of each of the designated function. For example, if f has type `string -> string` and g has type `string -> unit`, then the pointcut `{f,g}` admits the type `(string,all a.a)`. In other words, the polytype `all a.a` is used to generalize the difference in range between these functions.

An advice declaration in PolyAML consists of a **before** or **after** marker, a term expression specifying the pointcut, three variable bindings and the advice body:

```
advice before {f,g}:(string,all a.a) (x,y,z) =
    "(" ^ x ^ ")"
```

The first variable is either the function argument (in case of before advice) or the function result (in case of after advice). The second and third variable provide access to a representation of the stack and the intercepted function (which are part of other features of PolyAML which provide a form of run-time reflection). The advice body should produce a new value for either the function argument or result. In the above example, the advice will surround the string argument of the functions f and g with parentheses (note that ^ is the string concatenation operator).

The type checking of the advice declaration occurs as follows. From the pointcut type, the relevant polytype is considered (either domain or range), and this polytype is opened for the typing of the advice body[4]. That is, the type variables of the polytype are added to the context assumptions, and the monotype inside of the polytype is used as the type of x. In addition, the advice body expression should also be of this type.

Similar to the case of AspectualCaml, this provides a way to handle the definition of generic advice functions in a safe manner, with support for some type inference in the definitions. However, identical to the situation there, the language provides no direct support for

[4] Although the authors of PolyAML mark a polytype with the keyword **all**, suggesting a universal quantification, it bears resemblance to an existential type, for which a similar open operation is defined, see Section 6.2.3.

subsumption and subtypes, and it is unclear how the results translate to an object-oriented context.

9.4.3 AspectC++

Lohmann, Blaschke, and Spinczyk (2004) study the combination of AOP and C++ templates in the context of the AspectC++ language. C++ templates provide a compile-time meta-programming facility where different versions of one specification may be derived by providing different type arguments. In contrast to the generics facilities that we describe in Section 6.3, C++ template classes and template functions are not directly supported in the type system; they are expanded and type checked for each set of type arguments.

One dimension of the work by Lohmann et al. focuses on the use of aspects to instrument code in template classes and template functions. To this end, the pointcut language of AspectC++ is extended with patterns to match either specific instantiations of templates, or any instantiation. The other dimension focuses on the usage of generic code in aspects. The authors propose a mechanism where advice code is implicitly placed into a template member function that is called from each instrumented join point. A specifically-generated class that encodes the type information of the join point is passed as a type parameter for this call. As such the advice code may be type-checked by the underlying C++ compiler for each advised join point.

Concretely, advice methods in AspectC++ receive an implicit type argument `JoinPoint`, and an implicit ordinary argument `tjp` of type `*JoinPoint`. The `JoinPoint` type will provide a number of type members that are aliases to the concrete types of the join point at hand. These are named `Target`, `Result`, `Arg<i>`,... In addition, the `JoinPoint` type also provides a number of methods that the advice method may use to interact with the join point. The signatures of these methods are **`void`** `proceed()`, `Result *result()`, `Target *target()`,...

When an ordinary advice method is specified that accesses the join point context, then one may make use of the relation that exists between the type members of `JoinPoint` and the actual types of the join point. However, this relation is a subtype relation rather than an equality, so a cast operation is still required when *reading* join point values (not when *writing* them):

```
advice execution("Vector␣Calc::%(...)") : around() {
    Calc *target = (Calc*) *tjp->target();
    ...
    *tjp->result() = myVector;
}
```

As a consequence, when this advice is applied to a join point where the target is not of the type `Calc`, this is not detected.

On the other hand, one may specify generic advice by typing the advice method using the type members of `JoinPoint`. For example, the following advice method executes the interception join point twice and returns the result from the first invocation:

```
advice execution("%␣Calc::%(...)") : around() {
    typename JoinPoint::Result res;
    // First execution, copy result
```

```
    *tjp->proceed();
    res = *tjp->result();
    // Second execution, write result back
    *tjp->proceed();
    *tjp->result = res;
}
```

However, although this advice method is type safe for any join point, the ASPECTC++ compiler needs to check the body again at every join point where it is applied.

The templates of C++ are very expressive (in fact, Turing-complete) and Lohmann et al. exploit this to provide *generative advice*, which may not only abstract over the types of the join point context, but also over properties such as the number of arguments. However, the trade-off is that less abstraction is possible as type-checking of templates can only be done after their expansion. ASPECTC++ is therefore only capable of type-checking advices for a concrete join point at hand, while ASPECTJ (and our STRONGASPECTJ extension) allows to type-check advices against declared pointcut parameter types, irrespective of a base application.

9.4.4 AspectJ 5

The ASPECTJ 5 update by Colyer et al. (2005) modifies the ASPECTJ language to support JAVA 5 generics. Firstly, this involves coping with the presence of generics in the base language: type patterns can match generic types and their instantiations, and generic members can be defined through inter-type declaration. Secondly, type variables can be declared for aspects, similar to generic classes. When these variables are employed as regular type annotations in the aspect definition, this allows more advanced typing of generic aspect entities, akin to the advantages of generic classes explained in Section 6.3.1.

For example, one may define generic tracing behavior with the following definition:

```
abstract aspect GenericTracer<T> {
    abstract pointcut trace(T t);
    abstract void report(String d, T t);
    before(T t): trace(t) { report("Entering",t); }
    after(T t): trace(t) { report("Leaving",t); }
}
```

Although the aspect will work with a pointcut that exposes a context variable of any single type, we do not have the employ casts to invoke `report` from the advice methods. Also, when a concrete subaspect is defined, it will be verified that the implementation of the named pointcut `trace` and the method `report` involve the same type.

Additionally, type variables can also be employed in the type patterns of pointcuts and inter-type declarations, where they directly influence the semantics of the aspect. As such, a new class of generalizations (as partially proposed by Hanenberg and Unland, 2003) is made possible. For example, consider:

```
abstract aspect Serializer<S> {
    declare parents: S implements Serializable;
}
```

When we define a concrete subaspect of `Serializer<Order>`, then the type argument `Order` is not simply employed for typing; it causes the introduction of the interface `Serializable` in the class `Order` as well.

Although generic aspects can generalize functionality over different deployments, it is not possible to declare type variables for advice methods. It is as such not possible to generalize over different applications of an advice method in one deployment, as generic advice in StrongAspectJ allows. Furthermore, all of the typing problems outlined in Section 9.2 are not addressed by AspectJ 5.

Chapter 10

Formal Evaluation of Pointcut/Advice Bindings

In this chapter, we formally evaluate the soundness of the typing principles for pointcut/advice bindings proposed in this dissertation. This occurs through a formal definition of the semantics of pointcut/advice bindings in the context of the FEATHERWEIGHT JAVA calculus. Since it is a necessary preliminary to our contribution, we discuss this minimal object-oriented calculus in Section 10.1. We also employ the opportunity to present a version of FEATHERWEIGHT JAVA which includes a number of provisions with a small impact on the language, but which facilitate the definition of our extension later on. The extension itself is presented in Section 10.2; it adds a general advice construct and it employs the typing principles from Chapter 7 to type this new construct. Following the naming convention from this part of the dissertation, we refer to the extended language as FEATHERWEIGHT STRONGASPECTJ. We contribute a rigorous proof for the type safety properties of our extension in Section 10.2.2.

It may be surprising that we do start from the generic version of FEATHERWEIGHT JAVA to define our extension. This language already supports variable types, generic classes and generic methods, and after all, we employ similar concepts of parametric polymorphism to provide a generic typing of advice methods. An earlier version of FEATHERWEIGHT STRONGASPECTJ was indeed conceived as an extension of FEATHERWEIGHT GENERIC JAVA (see De Fraine et al., 2007). However, it is our experience that the organization of generic classes and methods (and the related concepts such as parameterized types and F-bounds) heavily complicate the presentation of the aspect extension.

Moreover, it is the case that a generic typing of advice methods is perfectly useful in a language without genericity (this point is also made by Jagadeesan et al., 2006, Sec. 2). A generic language is obviously a better model for recent object-oriented languages such as JAVA 5, but we have not investigated non-straightforward interactions with the concepts of generic classes and generic methods. (There may nevertheless be interesting interactions, for example, if one were to support that advice behavior for generic method invocations intercepts and updates the type arguments of this invocation.) Since the interesting issues with generics are still a part of future work, we decide to employ a non-generic language in this chapter.

10.1 Featherweight Java

FEATHERWEIGHT JAVA is proposed by Igarashi, Pierce, and Wadler (1999, 2001) as a minimal core calculus for JAVA and GJ (GJ is discussed in Section 6.3.1). It is formulated as a subset of JAVA which omits any feature which is not considered essential for a soundness proof; this includes features such as concurrency, inner classes, reflection, interfaces, abstract method declarations, method overloading, field shadowing, **super** invocations, primitive types, access control, **null** references and exceptions. Additionally, FEATHERWEIGHT JAVA omits assignment and mutable state, restricting JAVA to a "purely functional" fragment. The authors make this choice based on the observation that most of the tricky typing issues are independent of assignment. It is also a substantial simplification since the operational semantics of FEATHERWEIGHT JAVA may be formalized entirely within its own syntax, with no additional mechanisms such as object stores to model the heap, and without enforcing a deterministic evaluation order[1]. The calculus is similar in this respect to the systems we discuss in Chapter 6, since it also employs a small-step reduction to describe the evaluation of expressions.

10.1.1 Definition

The definition of FEATHERWEIGHT JAVA is presented over the course of Figure 10.1, Figure 10.2 and Figure 10.3, which define respectively the syntax, dynamics and statics of the calculus. The definition employs the notational conventions from Chapter 6. In particular, the terminals of the language are written in `teletype` and the metavariables in *italic*. Metavariables represent syntactic categories and different metavariables are used for the same syntactic category, in order to distinguish between multiple instances of the same syntactic element in one definition phrase or rule. In addition, we employ a specific notation to concisely describe sequences: $\bar{e}$ represents the ordered sequence of elements $e_1, \ldots, e_n$, where the element separator is determined by the context. This notation is sometimes extended across binary constructs, where the elements of two sequences are appropriately 'zipped', for example, $\bar{x} : \bar{P}$ is a shorthand for $x_1 : P_1, \ldots, x_n : P_n$. This implies that both sequences need to have an equal number of elements. Finally, we employ a bullet ($\bullet$) to represent an empty environment or an empty set of fields.

We first discuss the language definition from a bird's-eye view; below we make some more detailed remarks that refer to our specific definition of the language. In general terms, FEATHERWEIGHT JAVA supports mutually recursive class definitions, object creation, field access, method invocation, method override, method recursion through `this` and subtyping. The class `Object` is predefined with no fields and methods, and other classes are defined with exactly one parent and any number of fields and methods. The auxiliary function fields and meth are employed to lookup field and method declarations for the declared classes in an implicit class table (Figure 10.2, upper part). We assume that the class definitions adhere to the following conditions to ensure a sane definition of these functions: (i) there is at most one definition for a class name in the class table, (ii) there is no definition for `Object` in the class table, (iii) the parent of a class is either `Object` or another defined class, (iv) the inheritance hierarchy contains no cycles.

[1] Clearly, the call-by-value, left-to-right evaluation of JAVA is subsumed by this general relation. Additionally, it is possible to demonstrate *confluence* for the evaluation in FEATHERWEIGHT JAVA: different evaluations of the same program will yield the same result (modulo non-termination).

Lexical metavariables			
c, d		incl. `Object`	*class names*
f, g			*field names*
ℓ			*method names*
x, y, z		incl. `this`	*term variables*
Type and term expressions			
P, Q, R, S, T, U	::=	c	*class type*
K, L, M	::=	x	*variable term*
	|	$M\,.\,f@c$	*field access*
	|	$M\,.\,\ell@c(\bar{N})$	*method invocation*
	|	$\mathbf{new}\ c(\bar{N})$	*object creation*
Evaluation contexts			
$\mathscr{E}[]$	::=	$[]\,.\,f@c$	*field access target*
	|	$[]\,.\,\ell@c(\bar{N})$	*method inv. target*
	|	$M\,.\,\ell@c(\bar{N}, [], \bar{N}')$	*method inv. arg*
	|	$\mathbf{new}\ c(\bar{N}, [], \bar{N}')$	*object creation arg*
Member and top-level declarations			
μ	::=	$R\,\ell(\bar{x} : \bar{P})\{L\}$	*method declaration*
$\mathscr{D}$	::=	$\mathbf{class}\ c\ \mathbf{extends}\ d\ \{\bar{f} : \bar{T}\,;\,\bar{\mu}\}$	*class declaration*
Type and term environments			
Δ	::=	$\bullet$	*empty environment*
Γ	::=	$\bullet$	*empty environment*
	|	$\Gamma, x : T$	*variable term type*

Figure 10.1: FEATHERWEIGHT JAVA syntax

Class fields $\boxed{\vdash \mathsf{fields}(c) = \bar{f} : \bar{T}}$

FIELD-OBJECT
$$\vdash \mathsf{fields}(\texttt{Object}) = \bullet$$

FIELD-THIS-SUPER
$$\frac{\mathbf{class}\ c\ \mathbf{extends}\ d\ \{\bar{f} : \bar{T}\,;\cdots\} \qquad \vdash \mathsf{fields}(d) = \bar{g} : \bar{S}}{\vdash \mathsf{fields}(c) = \bar{g} : \bar{S}, \bar{f} : \bar{T}}$$

Class method $\boxed{\vdash \mathsf{meth}(c.\ell) = R(\bar{x} : \bar{P})\{L\}}$

METHOD-THIS
$$\frac{\mathbf{class}\ c\ \cdots\{\cdots R\,\ell(\bar{x} : \bar{P})\{L\}\cdots\}}{\vdash \mathsf{meth}(c.\ell) = R(\bar{x} : \bar{P})\{L\}}$$

METHOD-SUPER
$$\frac{\mathbf{class}\ c\ \mathbf{extends}\ d\ \{\cdots\,;\,\bar{\mu}\} \qquad \ell \notin \bar{\mu} \qquad \vdash \mathsf{meth}(d.\ell) = R(\bar{x} : \bar{P})\{L\}}{\vdash \mathsf{meth}(c.\ell) = R(\bar{x} : \bar{P})\{L\}}$$

Evaluation $\boxed{M \to M'}$

EVAL-FIELD
$$\frac{\vdash \mathsf{fields}(c) = \bar{f} : \bar{T}}{\mathbf{new}\ c(\bar{N})\,.\,f_i@d \to N_i}$$

EVAL-METHOD
$$\frac{M = \mathbf{new}\ c(\cdots) \qquad \vdash \mathsf{meth}(c.\ell) = (\bar{x})\{L\}}{M\,.\,\ell@d(\bar{N}) \to \left[(\texttt{this}, \bar{x}) \mapsto (M, \bar{N})\right] L}$$

EVAL-CONTEXT
$$\frac{M \to M'}{\mathscr{E}[M] \to \mathscr{E}[M']}$$

Figure 10.2: FEATHERWEIGHT JAVA dynamics and auxiliary definitions

Well-formed types $\boxed{\Delta \vdash T}$

TYPE-CLASS
$$\frac{\textbf{class}\ c\ \{\cdots\}}{\Delta \vdash c}$$

TYPE-OBJECT
$$\Delta \vdash \texttt{Object}$$

Well-formed environments $\boxed{\Delta \vdash \mathsf{ok}}$ $\boxed{\Delta;\Gamma \vdash \mathsf{ok}}$

TENV-EMPTY
$$\bullet \vdash \mathsf{ok}$$

ENV-EMPTY
$$\frac{\Delta \vdash \mathsf{ok}}{\Delta;\bullet \vdash \mathsf{ok}}$$

ENV-TERM-VAR
$$\frac{\Delta;\Gamma \vdash \mathsf{ok} \qquad x \notin \Gamma \qquad \Delta \vdash T}{\Delta;\Gamma, x : T \vdash \mathsf{ok}}$$

Subtyping $\boxed{\Delta \vdash S <: T}$

SUB-REFLEX
$$\Delta \vdash T <: T$$

SUB-TRANS
$$\frac{\Delta \vdash T <: T' \qquad \Delta \vdash T' <: T''}{\Delta \vdash T <: T''}$$

SUB-CLASS
$$\frac{\textbf{class}\ c\ \textbf{extends}\ d\ \{\cdots\}}{\Delta \vdash c <: d}$$

Term typing $\boxed{\Delta;\Gamma \vdash M : T}$

TERM-VAR
$$\frac{\Delta;\Gamma \vdash \mathsf{ok} \qquad x : T \in \Gamma}{\Delta;\Gamma \vdash x : T}$$

TERM-OBJECT
$$\frac{\Delta;\Gamma \vdash \mathsf{ok} \qquad \vdash \mathsf{fields}(c) = \bar{f} : \bar{T} \qquad \Delta;\Gamma \vdash \bar{N} : \bar{S} \qquad \Delta \vdash \bar{S} <: \bar{T}}{\Delta;\Gamma \vdash \textbf{new}\ c(\bar{N}) : c}$$

TERM-FIELD
$$\frac{\vdash \mathsf{fields}(c) = \bar{f} : \bar{T} \qquad \Delta;\Gamma \vdash M : S \qquad \Delta \vdash S <: c}{\Delta;\Gamma \vdash M.\ f_i@c : T_i}$$

TERM-METHOD
$$\frac{\vdash \mathsf{meth}(c.\ell) = R(\bar{P})\{\cdots\} \qquad \Delta;\Gamma \vdash (M, \bar{N}) : (S_0, \bar{S}) \qquad \Delta \vdash (S_0, \bar{S}) <: (c, \bar{P})}{\Delta;\Gamma \vdash M.\ \ell@c(\bar{N}) : R}$$

Member declaration typing $\boxed{\vdash \mu\ \mathsf{ok\ for}\ c, d}$

DEC-METHOD
$$\frac{\vdash R \qquad \bullet;(\texttt{this}, \bar{x}) : (c, \bar{P}) \vdash L : R' \qquad \vdash R' <: R \qquad \text{if } \vdash \mathsf{meth}(d.\ell) = S(\bar{Q})\{\cdots\} \text{ then } \vdash (R, \bar{Q}) <: (S, \bar{P})}{\vdash R\ \ell(\bar{x} : \bar{P})\{L\}\ \mathsf{ok\ for}\ c, d}$$

Declaration typing $\boxed{\vdash \mathscr{D}}$

DEC-CLASS
$$\frac{\vdash \mathsf{fields}(d) = \bar{g} : \bar{S} \qquad \bullet;(\bar{g}, \bar{f}) : (\bar{S}, \bar{T}) \vdash \mathsf{ok} \qquad \vdash \bar{\mu}\ \mathsf{ok\ for}\ c, d}{\vdash \textbf{class}\ c\ \textbf{extends}\ d\ \{\bar{f} : \bar{T}; \bar{\mu}\}}$$

Figure 10.3: FEATHERWEIGHT JAVA statics

A **new** expression is a structure that combines a class name and a number of subexpressions. Recall that in the lambda calculus (or in the functional language from Section 6.2.1) a function application may only be reduced after the function is simplified to a function abstraction. Similarly, field accesses and method invocations may be reduced in FEATHERWEIGHT JAVA when the receiver is simplified to a **new** expression (Figure 10.2, lower part). The field access operation will retrieve one of the subexpressions of a new expression (rule EVAL-FIELD). The method invocation will substitute the **new** expression (and any other arguments) in a function body that is selected according to the class name that is stored in the **new** expression (rule EVAL-METHOD). Since the method implementation that will be used is fixed only after the receiver term has been reduced to a **new** expression, this models the concept of *dynamic binding* in object-oriented languages.

The type checking is organized in a number of different levels (Figure 10.3). In order of increasing granularity these are types, environments, terms, members and finally top-level declarations. Additionally, subtyping is defined as the reflexive and transitive closure of the

parent link in class declarations. We note that, in general, well-formedness on a higher level will require (directly or indirectly) that all involved elements of a lower level are also well-formed, under the assumption that all declared classes are well-formed. For example, all types that appear in a well-formed class are also well-formed, the result of a term typing is always a well-formed type, *etc.* Most typing rules are familiar from JAVA, with some exceptions that we detail below.

Differences w.r.t. Chapter 6. Due to the conventions for FEATHERWEIGHT JAVA, there are three notable differences in organization compared to the systems of Chapter 6. First, the typing context is no longer a single set of assumptions A. Instead, this is divided between a type environment Δ and a term environment Γ. Second, there is no explicit subsumption rule (rule SUB). This means that a term typing always assigns the most specific type for that term. In order to recover the flexibility of subsumption, a requirement for a specific term typing result is always specified by means of a subtype relation in the condition of rules. Third, it is no longer the case that a subtype relation may only exist between well-formed types. Recall that in Chapter 6, the subtyping rules are organized to this effect, for example, rule SREFL in Figure 6.4 on Figure 6.4 on page 94 requires a well-formed type. In FEATHERWEIGHT JAVA, the convention is to organize the definition of subtyping without depending on the definition of well-formed types[2], and there is no corresponding condition in rule SUB-REFL here.

General differences w.r.t. Igarashi et al. FEATHERWEIGHT JAVA is specified by Igarashi et al. (1999) in order to prove the type safety of the GJ language and to evaluate the validity of the erasure procedure employed in the implementation of that language. This context dictates a number of features that are not relevant for our purposes and that we choose to omit. Firstly, Igarashi et al. define FEATHERWEIGHT JAVA as a strict subset of JAVA, *i.e.*, any valid FEATHERWEIGHT JAVA program is also a valid JAVA program. To this end, their version of the calculus includes constructors that have an entirely fixed form, and the method bodies always includes the **`return`** keyword. We do not include these syntactic redundancies here. In addition, we also relax the requirement that overriding methods have the exact same types in their signature as the method which they override. In JAVA, this is required at least for argument types in order to manage method overloading, but otherwise it is safe to allow overriding methods with a covariant return type and contravariant argument types (as stipulated in the last condition of rule DEC-METHOD). The soundness of such variant method overrides is formally proved together with our other extensions in Section 10.2.2. This also corresponds to the criteria for safe substitution of function types, as captured by rule SFUN in Figure 6.2 and Figure 6.4.

Secondly, Igarashi et al. include a cast operation, while we define a language without a general cast operation (nevertheless, we discuss some restricted form of up-casts below). The casts play a central role in the erasure procedure, but provide little insight for our purpose. In addition, their presence introduce non-essential complexity since we have to account for programs which might get stuck due to a failing down-cast. Such a situation is not a loophole in the type system, and the formulation of the safety properties needs to be adapted to exclude this case.

[2] In fact, in the version of FEATHERWEIGHT JAVA with generic classes (Igarashi et al., 1999, Sec. 3), the conditions for a well-formed parameterized class type include subtype relations, so the dependency is the other way around.

Specific provisions in anticipation of our extension. We also include a number of differences with respect to Igarashi et al. which enable us to present our extension in the following section as a simple set of changes to the current language; these provisions avoid a large number of trivial redefinitions later on. The first change is that we explicitly include a type environment in typing and subtyping judgments, although there are no type variables in this version and the type environment is always empty. As a second provision, we include a class name annotation (after the `@` sign) in the field access and method invocation terms. This class name makes explicit the type from which the field or method is accessed in the original source program. The annotation remains constant during the evaluation, while the receiver term of a field access or method invocation may be reduced to a term of a more specific type. This change facilitates the definition of the conditions under which advice behavior may replace the receiver of a method invocation (for field access operations, the change is not strictly necessary, but this is included nonetheless for uniformity).

The vigilant reader may raise the question to what extent this latter change causes a departure from the original FEATHERWEIGHT JAVA language. We address this concern in the following discussion. With respect to evaluation of the field access and method invocation forms, note that the class name annotation is not considered in the evaluation rules rule EVAL-FIELD and rule EVAL-METHOD in Figure 10.2: it is bound to a free metavariable d which is not otherwise constrained in each of these rules. So the evaluation of the terms remains entirely identical. There is an influence, however, with respect to typing in the rules rule TERM-FIELD and rule TERM-METHOD in Figure 10.3: the type of the field or method is retrieved for the class whose name is given in the annotation and the receiver type needs to be a subtype of this class's type. In contrast, in the original FEATHERWEIGHT JAVA, the field or method type is retrieved for the type of the receiver directly. For example, the original rule TERM-METHOD would be, in our notation:

$$\text{OTERM-METHOD}\ \frac{\vdash \mathsf{meth}(c.\ell) = R(\bar{P})\{\cdots\} \qquad \Delta;\Gamma \vdash (M,\bar{N}) : (c,\bar{S}) \qquad \Delta \vdash \bar{S} <: \bar{P}}{\Delta;\Gamma \vdash M.\,\ell(\bar{N}) : R}$$

We may illustrate a form of equivalence between these two variations as follows. When a term $M.\,\ell(\bar{N})$ is well-typed according to the original rule OTERM-METHOD, then it is trivial to demonstrate that M admits a type c such that the term $M.\,\ell@c(\bar{N})$ is well-typed in our version, and that this typing has the same result as the typing of the original term. In the opposite direction, when $M.\,\ell@c(\bar{N})$ is well-typed according to our typing rule, then the term $((c)M).\,\ell(\bar{N})$ is well-typed in the original language, as illustrated by the following derivation, which employs precisely the conditions from our version of rule TERM-METHOD as its assumptions:

$$\text{OTERM-METHOD}\ \frac{\vdash \mathsf{meth}(c.\ell) = R(\bar{P})\{\cdots\} \qquad \Delta;\Gamma \vdash \bar{N} : \bar{S} \qquad \Delta \vdash \bar{S} <: \bar{P} \qquad \text{OTERM-UCAST}\ \dfrac{\Delta;\Gamma \vdash M : S_0 \qquad \Delta \vdash S_0 <: c}{\Delta;\Gamma \vdash (c)M : c}}{\Delta;\Gamma \vdash ((c)M).\,\ell(\bar{N}) : R}$$

The casting to c will always succeed (it is an up-cast), and it is only included to ensure that the typing has the same result. We may additionally show that $M.\,\ell(\bar{N})$ is well-typed, but since M may be of a more specific type than c, say d, the result of the typing may be a subtype of R when method ℓ is overridden with a covariant result type in d.

As a conclusion, we state that the method invocations and field accesses in our version of the FEATHERWEIGHT JAVA language may be interpreted to include implicit up-casts that record the original type of the receiver as it is replaced by a term of a more specific type. The class name annotations remain constant when subterms of the method invocation or field access are reduced, and eventually disappear when the method invocation or field access term itself is reduced. A well-typed source program in the original FEATHERWEIGHT JAVA language may be trivially converted before evaluation to a well-typed program in our version of the language, by inserting class name annotations that correspond to the static types of the receivers.

10.1.2 Safety Properties

We now describe what we mean when we claim that the language definition of FEATHERWEIGHT JAVA is type sound. In Section 6.1.2, type soundness for a rewriting calculus is described as the guarantee that the reduction of a well-typed program will only halt when a primitive value has been obtained. Therefore we first need to provide some definition of a well-typed program and of a primitive value.

For FEATHERWEIGHT JAVA, a complete program specification consists of a collection of top-level declarations (the *class table*) and a starting term. We will explicitly make the assumption that all of the top-level declarations $\mathscr{D}$ are well-formed (written $\vdash \mathscr{D}$). A well-typed program may then simply be equated to a well-typed starting term.

Assumption 10.1.1. *If $\mathscr{D}$ then $\vdash \mathscr{D}$.*

In the context of FEATHERWEIGHT JAVA, a primitive value is a **new** expression with only other primitive values among its subexpressions. An example of such a term is:

$$\textbf{new}\ c(\textbf{new}\ d(),\textbf{new}\ d())$$

It is clear that when the execution ends at such a form, this does not indicate any type error; rather, it is the logical result of a computation. We formally describe which terms are considered primitive values by means of a judgment $\vdash \mathsf{val}\ M$, which is defined using the following rule:

$$\textsc{Value}\ \frac{\vdash \mathsf{val}\ \bar{N}}{\vdash \mathsf{val}\ \textbf{new}\ c(\bar{N})}$$

With these preliminaries in place, we recall from Section 6.1.2 that the classic strategy for the demonstration of the soundness property consists in proving two separate properties: preservation and progress. The formal definition of these properties for FEATHERWEIGHT JAVA is the following:

Theorem 10.1.2 (Preservation)**.** *If $\Delta;\Gamma \vdash M : T$ and $M \to M'$, then $\Delta;\Gamma \vdash M' : S$ for some S such that $\Delta \vdash S <: T$.*

Theorem 10.1.3 (Progress)**.** *If $\bullet;\bullet \vdash M : T$ then either $\vdash \mathsf{val}\ M$ or $M \to M'$ for some M'.*

These theorems provide more powerful results than what is strictly needed for the safety property. For example, preservation states that a reduction step not simply retains the well-formedness of a term, but also retains an equal or more specific type.

Igarashi et al. provide proofs of these theorems for their version of FEATHERWEIGHT JAVA. The proofs proceed by induction on the derivation of the evaluation judgment (in case of preservation) or on the derivation of the typing judgment (in case of progress). The proof-by-induction strategy explains why a more powerful preservation property is formulated: this also implies that more powerful induction hypotheses may be used, which is needed to successfully complete the proof.

The proofs make use of a number of auxiliary lemmas, most importantly, the facts that subtyping preserves field and method types and that the substitution of term values preserves the typing. It is straightforward to adapt the proofs of Igarashi et al. to support our version of the FEATHERWEIGHT JAVA language. Due to the relaxed rules for overriding methods, subtyping will not simply retain the signature of a method but instead provide a method with an equal or more specific return type and equal or more general argument types. These relaxed guarantees are still sufficient for the use of the lemma in the proof of the preservation property. Otherwise, the original proof structure may be retained.

We do not present the proofs for our version of the FEATHERWEIGHT JAVA language in further detail here, since we revisit the question of type safety of the language after the presentation of our complete extension in the next section.

10.2 Featherweight StrongAspectJ

We now extend the FEATHERWEIGHT JAVA language in order to formally prove the soundness of the typing principles from Chapter 7. The goal of this extension, which is called FEATHERWEIGHT STRONGASPECTJ, is to describe the semantics of the essential interaction mechanism of advising join points, as well as our typing principles for this mechanism, in their most general form. Note that we do not aim to faithfully model the semantics of every feature present in concrete aspect languages such as ASPECTJ and STRONGASPECTJ. Similar to Igarashi et al., we consider the effect of a feature on the complexity of the type soundness proof as a "litmus test" for its inclusion: any feature that significantly complicates the development without contributing to the fundamental insights with regards to typing is a candidate for removal. The benefit of this treatment is that we are able to prove the soundness of those features which we *do* model using state-of-the-art approaches for formal rigor in Section 10.2.2.

10.2.1 Definition

The extensions of FEATHERWEIGHT STRONGASPECTJ are presented over the course of Figure 10.4, Figure 10.5 and Figure 10.6. As an overview, we state that the language includes support for around advice methods that is capable of intercepting method invocations. The advice method bodies may invoke the intercepted method (or other advice instances of an advice chain) by means of a `proceed` invocation, which may provide a new receiver and new arguments. The typing of the advice method occurs by means of type variables which have both a lower and upper bound. An advice is compatible with a method invocation join point when the method type is contained with the range between the lower and upper bound. Below, we describe each of these elements in more detail.

Lexical metavariables

$\ldots$			*(previous variables)*
a			*advice names*
X, Y, Z			*type variables*

Type, term and pointcut expressions

P, Q, R, S, T, U	::=	$\ldots$	*(previous form)*
	|	X	*variable type*
	|	`Null`	*bottom type*
K, L, M	::=	$\ldots$	*(previous forms)*
	|	$M\texttt{.}\ell\texttt{@}c\texttt{[}\bar{a}\texttt{](}\bar{N}\texttt{)}$	*advised method invocation*
	|	$\textbf{proceed}\texttt{(}M\texttt{,}\bar{N}\texttt{)}$	*proceed invocation*

Evaluation contexts

$\mathscr{E}[]$	::=	$\ldots$	*(previous forms)*
	|	$[]\texttt{.}\ell\texttt{@}c\texttt{[}\bar{a}\texttt{](}\bar{N}\texttt{)}$	*advised method inv. target*
	|	$M\texttt{.}\ell\texttt{@}c\texttt{[}\bar{a}\texttt{](}\bar{N}\texttt{,}[]\texttt{,}\bar{N}'\texttt{)}$	*advised method inv. arg*
	|	$\textbf{proceed}\texttt{(}[]\texttt{,}\bar{N}\texttt{)}$	*proceed inv. target*
	|	$\textbf{proceed}\texttt{(}M\texttt{,}\bar{N}\texttt{,}[]\texttt{,}\bar{N}'\texttt{)}$	*proceed inv. arg*

Top-level declarations

$\mathscr{D}$	::=	$\ldots$	*(previous forms)*
	|	$\textbf{advice}\ Y\ \textbf{in}\ R\texttt{–}S\ a\texttt{(}x_0\texttt{:}X_0\ \textbf{in}\ P_0\texttt{–}Q_0\texttt{,}$ $\bar{x}\texttt{:}\bar{X}\ \textbf{in}\ \bar{P}\texttt{–}\bar{Q}\texttt{)}\ \{L\}$	*advice declaration*

Type and term environments

Δ	::=	$\ldots \mid \Delta, X\ \textbf{in}\ S\texttt{–}U$	*variable type bounds*
Γ	::=	$\ldots \mid \Gamma, S\ \textbf{proceed}\texttt{(}Q_0\texttt{,}\bar{Q}\texttt{)}$	*proceed type*

Figure 10.4: FEATHERWEIGHT STRONGASPECTJ extension of the FEATHERWEIGHT JAVA syntax

Advice compatibility $\boxed{\vdash a \text{ ok for } c.\ell}$

$$\textsc{AdvCompat-Method}\quad \frac{\textbf{advice}\ R\text{–}S\ a(P_0\text{–}Q_0, \bar{P}\text{–}\bar{Q})\ \{\cdots\} \quad \vdash \mathsf{meth}(c.\ell) = T(\bar{U})\{\cdots\} \qquad \vdash (R, P_0, \bar{P}) <: (T, c, \bar{U}) <: (S, Q_0, \bar{Q})}{\vdash a \text{ ok for } c.\ell}$$

Evaluation $\boxed{M \to M'}$

$$\ldots \quad \textsc{Eval-Method}\quad \frac{M = \textbf{new}\ c(\cdots) \qquad \vdash \mathsf{meth}(c.\ell) = (\bar{x})\{L\}}{M.\ell@d[\,](\bar{N}) \to \big[(\texttt{this}, \bar{x}) \mapsto (M, \bar{N})\big] L}$$

$$\textsc{Eval-Select}\quad \frac{\vdash \bar{a} \text{ ok for } c.\ell}{M.\ell@c(\bar{N}) \to M.\ell@c[\bar{a}](\bar{N})}$$

$$\textsc{Eval-Advise}\quad \frac{\textbf{advice}\ a_0(x_0, \bar{x})\{L\}}{M.\ell@c[a_0, \bar{a}](\bar{N}) \to \big[(x_0, \bar{x}, \textbf{proceed}) \mapsto (M, \bar{N}, \ell@c[\bar{a}])\big] L}$$

Figure 10.5: FEATHERWEIGHT STRONGASPECTJ extension of FEATHERWEIGHT JAVA dynamics and auxiliary advice compatibility judgment

Advice and proceed mechanism

FEATHERWEIGHT STRONGASPECTJ supports the declaration of advice methods as top-level entities, next to classes (Figure 10.4). It is not an uncommon simplification in this context to not consider aspect modules and aspect instances; their addition is normally straightforward. The advice methods define a number of argument names and an advice body which may include invocations of the special `proceed` method. (The advice method declaration also includes some type information which we discuss below.)

The evaluation of method invocations is adapted to include the effect of advice behavior. In Figure 10.5 (bottom part), the original rule EVAL-METHOD is replaced by three rules. rule EVAL-SELECT equips a method invocation expression with an advice chain, which is modeled as a list of remaining advice names (the advised method invocation is a new syntactic form). rule EVAL-ADVISE applies the advice at the head of the list and reduces the method invocation to the advice body, where `proceed` is bound to the method invocation with the remaining advice list. When the advice list is exhausted, rule EVAL-METHOD executes the method lookup and reduces the method invocation to the method body. We note four particular features about the organization of advice application in these rules:

1. We consider the possible effects of arbitrary pointcut expressions in any pointcut language by describing a non-deterministic advice selection process (this explains why the advice declarations do not include pointcut expressions). The condition of rule EVAL-SELECT simply stipulates that each of the selected advice methods must be compatible with the method at hand (this is captured by the judgment "$\vdash a$ ok for $c.\ell$", which is described below). There are no other constraints that further determine the advice list. Consequently, multiple advice lists may satisfy this criterion: if we consider the set of all

advice methods that are compatible with the invoked method, then any advice list which only draws its elements from this set (any number of elements, in any order, possibly with duplication) is accepted.

This non-determinism allows for different reductions of the same term yielding different results, which is obviously a highly undesirable property for a practical language. However, we find it very useful in the context of this theoretical study of the typing principles to abstract over different possible pointcut languages and interaction resolution mechanisms[3]: the evaluation of a concrete pointcut expression and a concrete interaction resolution may narrow the possibilities down to a single advice list and thus recover determinism, as long as the resulting advice list satisfies the criterion from rule EVAL-SELECT, we have considered its possibility in the soundness evaluation and our results will be applicable.

2. In rule EVAL-ADVISE, we note that advice methods receive the arguments directly from the original method invocation (or from a `proceed` invocation by a preceding advice in the chain), without any "routing" by means of a pointcut expression. We consider the routing of context arguments by pointcut expression to be a factor of non-essential complexity for our purposes. The omission of this feature implies that a single advice method may not be used to advise the invocations of multiple methods that have a different structure. For example, in the pointcut language of ASPECTJ, this is possible using a pointcut expression with multiple disjunctive branches that each provide a binding of the same pointcut variable:

```
call(void FigureElement.move(..)) && target(fe) ||
call(void Group.add(FigureElement)) && args(fe)
```

However, such examples may be emulated by means of multiple advice methods, one for each different structure of the advised methods.

3. Similarly to treatment of the arguments of advice, the arguments to the invocation of `proceed` are directly employed for the execution of the method body (or for the execution of the body of a succeeding advice in the chain). This occurs through the mechanism of substituting a method name (ℓ), a class annotation (c) and an advice list ($\bar{a}$) for the `proceed` marker. The proceed substitution will replace the `proceed` invocation by an invocation of the given method, advised by the given advice list. Since this is not a common substitution operation, we provide a formal definition of the substitution behavior. For `proceed` invocation terms, the effect of the substitution will be the method invocation we described:

$$\left[\mathbf{proceed} \mapsto \ell@c[\bar{a}]\right]\mathbf{proceed}(M, \bar{N}) = M.\,\ell@c[\bar{a}](\bar{N})$$

For terms of any other form, the substitution the substitution is simply carried over to any subterms. For example, in case of field access terms there is one such subterm:

$$\left[\mathbf{proceed} \mapsto \ell@c[\bar{a}]\right](M.\,f@c) = (\left[\mathbf{proceed} \mapsto \ell@c[\bar{a}]\right]M).\,f@c$$

[3] Recall from the discussion of interaction resolution in Section 4.1.3 that the resolution of a situation where multiple aspects advise the same join point may be modeled as a manipulation of the list of applicable advice methods.

Well-formed types $\boxed{\Delta \vdash T}$

$$\cdots \qquad \textsc{Type-Var}\ \frac{X \in \Delta}{\Delta \vdash X}$$

$$\textsc{Type-Null}\ \frac{}{\Delta \vdash \texttt{Null}}$$

Well-formed environments $\boxed{\Delta \vdash \mathsf{ok}}$ $\boxed{\Delta;\Gamma \vdash \mathsf{ok}}$

$$\cdots \qquad \textsc{TEnv-Var}\ \frac{\Delta \vdash \mathsf{ok} \quad X \notin \Delta \quad \Delta \vdash S, T}{\Delta, X\ \mathbf{in}\ S\text{–}T \vdash \mathsf{ok}} \qquad \textsc{Env-Proceed}\ \frac{\Delta;\Gamma \vdash \mathsf{ok} \quad \mathbf{proceed} \notin \Gamma \quad \Delta \vdash S, Q_0, \bar{Q}}{\Delta;\Gamma, S\ \mathbf{proceed}(Q_0, \bar{Q}) \vdash \mathsf{ok}}$$

Subtyping $\boxed{\Delta \vdash S <: T}$

$$\cdots \qquad \textsc{Sub-Null}\ \frac{}{\Delta \vdash \texttt{Null} <: T}$$

$$\textsc{Sub-VarU}\ \frac{X\ \mathbf{in}\ S\text{–}U \in \Delta}{\Delta \vdash X <: U} \qquad \textsc{Sub-VarS}\ \frac{X\ \mathbf{in}\ S\text{–}U \in \Delta}{\Delta \vdash S <: X}$$

Term typing $\boxed{\Delta;\Gamma \vdash M : T}$

$$\cdots$$

$$\textsc{Term-Proceed}\ \frac{S\ \mathbf{proceed}(Q_0, \bar{Q}) \in \Gamma \quad \Delta;\Gamma \vdash (M, \bar{N}) : (P_0, \bar{P}) \quad \Delta \vdash (P_0, \bar{P}) <: (Q_0, \bar{Q})}{\Delta;\Gamma \vdash \mathbf{proceed}(M, \bar{N}) : S}$$

$$\textsc{Term-AMethod}\ \frac{\vdash \bar{a}\ \mathsf{ok\ for}\ c.\ell \quad \vdash \mathsf{meth}(c.\ell) = R(\bar{P})\{\cdots\} \quad \Delta;\Gamma \vdash (M, \bar{N}) : (S_0, \bar{S}) \quad \Delta \vdash (S_0, \bar{S}) <: (c, \bar{P})}{\Delta;\Gamma \vdash M.\ell@c[\bar{a}](\bar{N}) : R}$$

Declaration typing $\boxed{\vdash \mathscr{D}}$

$$\cdots \qquad \textsc{Dec-Advice}\ \frac{\Delta = Y\ \mathbf{in}\ R\text{–}S, X_0\ \mathbf{in}\ P_0\text{–}Q_0, \bar{X}\ \mathbf{in}\ \bar{P}\text{–}\bar{Q} \quad \Delta;(x_0, \bar{x}) : (X_0, \bar{X}), Y\ \mathbf{proceed}(X_0, \bar{X}) \vdash L : U \quad \Delta \vdash U <: Y}{\vdash \mathbf{advice}\ Y\ \mathbf{in}\ R\text{–}S\ a(x_0 : X_0\ \mathbf{in}\ P_0\text{–}Q_0, \bar{x} : \bar{X}\ \mathbf{in}\ \bar{P}\text{–}\bar{Q})\ \{L\}}$$

Figure 10.6: FEATHERWEIGHT STRONGASPECTJ extension of FEATHERWEIGHT JAVA statics

4. Lastly, we note that the receiver of the method invocation is included with the arguments that are provided to the advice body in rule EVAL-ADVISE. The receiver may be considered the *implicit* argument of the method invocation and it is passed along together with the explicit arguments (in the position x_0). Similarly, `proceed` invocations require a receiver argument in the first position, and this does not need to be the original receiver of the method invocation. This mimics the possibility in aspect languages of advice behavior that changes the receiver object; as explained by Clifton and Leavens (2006), this is useful for such idioms as introducing proxy objects. Note that for the execution of the method body in rule EVAL-METHOD, method lookup occurs for the final receiver that is obtained after any replacements by any of the advice methods. The semantics of dynamic binding of the method implementation is thus always retained.

Typing of the extensions

FEATHERWEIGHT STRONGASPECTJ includes the following changes to organize the typing of the new constructs:

Type structure We extend the type structure with type variables and a bottom type (`Null`). There are corresponding type well-formedness and subtyping rules for these new type forms (Figure 10.6). The definition of these new rules is straightforward and corresponds with earlier versions in Figure 6.4 on page 94. We choose the reserved name `Null` for the bottom type, since it corresponds to the null type in JAVA, which is also a subtype of any reference type. However, note that in contrast to the situation in JAVA, `Null` does not have any inhabitants, *i.e.*, we do not model **`null`** references (for reasons similar to those given for casts in Section 10.1.1). The type `Null` is still a useful generalization in type comparisons: using `Null` as a lower bound for a type variable means that there is no effective lower bound, any subtype of the upper bound will be a valid type value for that type variable.

Advice typing The typing of advice declarations is the first place where we incorporate our typing principles. Similar to Chapter 8 and Chapter 9, this is based on the typing principles described for advice functions for function join points from Section 7.3, where we propose a typing using subtype polymorphism (ordinary function advice, Section 7.3.3) and another one using parametric polymorphism (generic function advice, Section 7.3.4). In the current context, we are interested in including the *most general* typing principle, and the result from page 113 regarding the incorporation of subtype polymorphism in parametric polymorphism is of particular interest here, since it claims that ordinary advice may be emulated using generic advice, provided that the type variables that are used for the generic typing have an upper *and* lower bound.

We may demonstrate this with an example of an ordinary advice method in the STRONGASPECTJ language from page 141:

```
Integer around(Number n): pc(n): Number proceed(Integer) {
    return proceed(n.intValue() + 1).intValue() - 1;
}
```

If both upper and lower bounds are supported for type variables, this example may be recreated as a generic advice method where the type variables have the concrete types from the advice and proceed signatures as their lower and upper bounds:

```
<Y in Integer-Number, X in Integer-Number>
Y around(X n): pc(n): Y proceed(X) {
    return proceed(n.intValue() + 1).intValue() - 1;
}
```

Note that although the types in the around and proceed signature must be equal, any value of type `X` may be retrieved as a `Number` (due to its upper bound) and may be provided as an `Integer` (due to its lower bound). The same holds for type `Y`.

In the design of FEATHERWEIGHT STRONGASPECTJ, we therefore choose to include a general generic advice typing where the type of the result and all of the argument position are necessarily specified as type variables, which have their bounds specified in-place. In the syntax of FEATHERWEIGHT STRONGASPECTJ, the above example becomes:

```
advice Y in Integer-Number a(n : X in Integer-Number) {
    proceed(n.intValue().succ()).intValue().pred();
}
```

(Since there are no primitive arithmetic operations in the context of FEATHERWEIGHT JAVA, we presume the simple operations for addition and subtraction by one are provided as the methods `succ` and `pred` for the class `Integer`.)

The rule DEC-ADVICE requires a typing of the advice body using a type environment that contains the assumption about the type variables, and a term environment that assigns the variable types to the respective arguments and to the `proceed` marker (the latter assignment allows `proceed` invocations to be used in the advice body because of rule TERM-PROCEED). Also note that the conditions of rule DEC-ADVICE correspond to the conditions for obtaining the type which we assigned to generic function advice in Section 7.3.4:

$$\vdash adv\colon \mathbf{forall}\ I\ \mathbf{in}\ U^i_{adv} - S^i_{adv},\ O\ \mathbf{in}\ S^o_{adv} - U^o_{adv},\ (I \text{->} O) \text{->} (I \text{->} O)$$

Although in the case of rule DEC-ADVICE, this is extended to the case of multiple arguments and the roles of O and I are played by respectively Y and $\bar{X}$, and the roles of S^o_{adv}, U^o_{adv}, U^i_{adv} and S^i_{adv} are played by respectively R, S, $\bar{P}$ and $\bar{Q}$.

Advice compatibility Finally, we define the notion of advice compatibility in Figure 10.5 (upper part), which is employed as both a condition for the introduction of an advised method invocation in rule EVAL-SELECT and the typing of such a term in rule TERM-AMETHOD. The conditions for advice compatibility $\vdash a$ ok for $c.\ell$ need to ensure that the advice method a may be safely applied to the invocation of method $c.\ell$, under the assumption that a is well-typed. Concretely, rule ADVCOMPAT-METHOD stipulates that the method types should be contained within the ranges from the advice declaration. This corresponds to the conditions of rule FGBIND in Figure 7.3 on page 116, although there is no involvement of a pointcut type and the join point type (*i.e.*, the type of the invoked method) is directly used instead. (In other words, the subtype conditions from rule FMATCH in the same figure are directly integrated here.) Once again, this fits in our design motivation of eliminating all of the non-essential complexity.

Class annotations Now that the advice compatibility has been described, we are in a position where we can more concretely explain why we introduce explicit class annotations in the syntax for method invocations (as described at the end of Section 10.1.1). Consider a context without these annotations and consider the advised method invocation $M.\ell[\bar{a}](\bar{N})$. Suppose that this term is correctly typed, because M admits a class type c and the advice methods $\bar{a}$ are compatible with the type of method $c.\ell$. This entails that c is contained in the range of the receiver position of each of the advice methods $\bar{a}$.

Now, we remark that the term $M.\ell[\bar{a}](\bar{N})$ may reduce to the term $M'.\ell[\bar{a}](\bar{N})$ where M' admits a class type d where $d <: c$. In a subject reduction proof, we must show that the

typing of the original term is preserved by the new term. However, it is a problem to show that the advice methods $\bar{a}$ are compatible with the new type of the invoked method, since the invoked method has become $d.\ell$, while d might not be included in the receiver ranges of the advice methods $\bar{a}$. The class annotations provide a simple way to retain the information that $c.\ell$ is the method that is being invoked in the original program.

Alternatively, we may consider a design where we track in which class in the hierarchy the method ℓ is first introduced. As long as the `proceed` invocations in the advice method provide a receiver that is a subtype of this class type, the method invocation would be safe. However, it would be rather restrictive if the advice method could not *expect* a receiver argument of a type more specific of this class type. All in all, we believe the mechanism of class annotations is a more straightforward mechanism since it allows the receiver to be treated similar to the other (explicit) arguments.

10.2.2 Safety Properties and Corresponding Proofs

In this section, we state the progress and preservation theories for FEATHERWEIGHT STRONGASPECTJ. There is a proof for theorems; the proof has been developed using the COQ proof assistant[4], which is described by Bertot and Castéran (2004). A proof assistant of this kind allows the expression of formal assertions and mechanically checks the proofs of these assertions (it is not an automated theorem prover but includes some automated theorem proving tactics and various decision procedures). In the particular case of COQ, this is based on the type theory of the calculus of (inductive) constructions, which is a higher-order lambda calculus initially developed by Coquand and Huet (1988). The core logic of COQ is intuitionistic rather than classical. Compared to the traditional paper-and-pencil proofs, mechanical proofs checked by tools such as COQ rule out the possibility of mistakes by the proof author[5]. They represent the current state-of-the-art in formal rigor.

Below, we present the main theorems of the development, as well as a number of key auxiliary lemmas, in the notation of this chapter. We present the properties in the order in which they can be derived (*i.e.*, the auxiliaries come before the main theorems). We explicitly repeat the assumption that all top-level declarations are well-typed:

Assumption 10.2.1. *If* $\mathscr{D}$ *then* $\vdash \mathscr{D}$.

There is a description of the proof with each statement. Since the details of the proofs have been rigorously verified by COQ, we write the proof descriptions with the intent of providing the reader with some insight in the proof structure, rather than convincing him/her of the correctness of the proof. We therefore permit proof descriptions that are more informal than what is customary for paper-and-pencil proofs.

Subtyping, term and proceed substitution

The first two lemmas provide some properties regarding the subtype relation. Remark that we consider a relation $\vdash c <: d$ between class types under an *empty* type environment. It

[4] The full COQ development is available from `http://ssel.vub.ac.be/fsaj/`. A formatted version of the COQ definitions is included as Appendix A.

[5] There is still a possibility for bugs in the proof checking system or its underlying theory, as is discussed by Castéran et al. (2009, answer 5). However, this is unlikely as it would have implications beyond any single development.

is not possible to draw the same conclusions for an arbitrary type environment: if the type environment contains unsatisfiable assumptions such as in the case of $\Delta = \mathtt{X}\ \mathbf{in}\ \mathtt{Number{-}String}$, then it is allowed to derive subtype relations that do not correspond to the class hierarchy, for example, $\Delta \vdash \mathtt{Number} <: \mathtt{String}$. In contrast, under the empty type environment, subtyping corresponds to the reflexive and transitive closure of the direct subclass relation.

Lemma 10.2.2 (Subtyping preserves fields)**.** *If* $\vdash c <: d$ *and* $\vdash \mathsf{fields}(d) = \bar{g} : \bar{S}$ *then* $\vdash \mathsf{fields}(c) = \bar{g} : \bar{S}; \bar{f} : \bar{T}$ *for some* $\bar{f}$ *and* $\bar{T}$*.*

Proof. By induction on the left transitive step of the derivation of the subtype relation. In the case of rule SUB-CLASS, rule FIELD-THIS-SUPER may be used directly. The cases of rule SUB-VARU and rule SUB-VARS are absurd in case of an empty type environment. The case of rule SUB-NULL is also impossible since `Null` is not a class type. □

Lemma 10.2.3 (Subtyping preserves method types)**.** *If* $\vdash c <: d$ *and* $\vdash \mathsf{meth}(d.\ell) = S(\bar{Q})\{\cdots\}$ *then* $\vdash \mathsf{meth}(c.\ell) = R(\bar{P})\{\cdots\}$ *for some* $R, \bar{P}$ *such that* $\vdash (R, \bar{Q}) <: (S, \bar{P})$*.*

Proof. We employ the same structure as in the previous lemma. In the case of rule SUB-CLASS, we have to distinguish between a method that is inherited with or without overriding. In case of overriding, we employ the last condition from rule DEC-METHOD. □

The following two lemmas are a typical term substitution lemma and a similar one, adapted for the case of proceed substitution.

Lemma 10.2.4 (Term substitutivity)**.** *Consider variable declaration* $\bar{x} : \bar{P}$ *and* $\Gamma_0, \bar{N}$ *such that* $\Delta; \Gamma_0 \vdash \bar{N} : \bar{Q}$ *for some* $\bar{Q}$ *such that* $\Delta \vdash \bar{Q} <: \bar{P}$*. Now, if* $\Delta; \bar{x} : \bar{P} \vdash M : T$ *then* $\Delta; \Gamma_0 \vdash [\bar{x} \mapsto \bar{N}]\, M : T'$ *for some* T' *such that* $\Delta \vdash T' <: T$*.*

Proof. By induction on the derivation of $\Delta; \bar{x} : \bar{P} \vdash M : R$. In the case of rule TERM-VAR, we know the variable in question is among $\bar{x}$, so the goal follows straightforwardly. In the other cases, the substitution is carried through to any subterms, and we employ the induction hypotheses corresponding to those subterms. □

Lemma 10.2.5 (Proceed substitutivity)**.** *Consider proceed declaration* $S\ \mathbf{proceed}(Q_0, \bar{Q})$ *and* $\ell, c, \bar{a}$ *such that* $\vdash \bar{a}$ ok for $c.\ell$ *and* $\vdash \mathsf{meth}(c.\ell) = R(\bar{P})\{\cdots\}$ *where* $\Delta \vdash (R, Q_0, \bar{Q}) <: (S, c, \bar{P})$*. Now, if* $\Delta; S\ \mathbf{proceed}(Q_0, \bar{Q}) \vdash M : T$ *then* $\Delta; \bullet \vdash [\mathbf{proceed} \mapsto c@\ell[\bar{a}]]\, M : T'$ *for some* T' *such that* $\Delta \vdash T' <: T$*.*

Proof. By induction on the derivation of the judgment $\Delta; S\ \mathbf{proceed}(Q_0, \bar{Q}) \vdash M : T$. In the case of rule TERM-PROCEED, the goal follows straightforwardly from the hypotheses. In the other cases, the substitution is carried through to any subterms, and we employ the induction hypotheses corresponding to those subterms. □

Type substitution

The following two lemmas relate to the substitution of types according to the bound declarations from a type environment.

Lemma 10.2.6 (Type substitution preserves subtyping)**.** *Consider the type variable bound declarations* $\bar{X}$ **in** $\bar{S}$–$\bar{U}$ *and types* $\bar{V}$ *such that* $\vdash \bar{V}$ *and* $\vdash \bar{S} <: \bar{V} <: \bar{U}$*. Now, if* $\bar{X}$ **in** $\bar{S}$–$\bar{U} \vdash T <: T'$*, then* $\vdash [\bar{X} \mapsto \bar{V}]\, T <: [\bar{X} \mapsto \bar{V}]\, T'$*.*

Proof. By induction on the derivation of $\bar{X}$ **in** $\bar{S}$–$\bar{U} \vdash T <: T'$. In the cases of rule SUB-VARU and rule SUB-VARS, we know the type variable in question will be among $\bar{X}$ and the goal follows from the hypotheses. The other cases are straightforward or the substitution has no effect. □

Lemma 10.2.7 (Type substitutivity)**.** *Consider the type variable bound declarations* $\bar{X}$ **in** $\bar{S}$–$\bar{U}$ *and types* $\bar{V}$ *such that* $\vdash \bar{V}$ *and* $\vdash \bar{S} <: \bar{V} <: \bar{U}$*. Now, if* $\bar{X}$ **in** $\bar{S}$–$\bar{U}; \Gamma \vdash M : T$*, then* $\bullet; [\bar{X} \mapsto \bar{V}]\, \Gamma \vdash M : T'$ *for some* T' *such that* $\vdash T' <: [\bar{X} \mapsto \bar{V}]\, T$*.*

Proof. By induction on the derivation of $\bar{X}$ **in** $\bar{S}$–$\bar{U}; \Gamma \vdash M : T$. In the cases of rule TERM-VAR and rule TERM-PROCEED, the type $[\bar{X} \mapsto \bar{V}]\, T$ is available in $[\bar{X} \mapsto \bar{V}]\, \Gamma$. In the cases of rule TERM-FIELD, rule TERM-OBJECT and rule TERM-METHOD, we employ the fact that methods and fields have well-formed types in the empty type environment, *i.e.*, that they do not contain any type variables. We additionally employ Lemma 10.2.6 and the induction hypotheses. □

Main properties

The following two theorems provide that preservation and progress properties. The definition of the primitive values is the same as in the case of Section 10.1.2. Note that the preservation property is specified for an empty type and term environment, to match the conditions of Lemma 10.2.2 and Lemma 10.2.3. This does not affect the soundness result, since the evaluation of a program starts with a main expression that is well-typed in an empty environment.

Theorem 10.2.8 (Preservation)**.** *If* $\bullet; \bullet \vdash M : T$ *and* $M \to M'$*, then* $\bullet; \bullet \vdash M' : T'$ *for some* T' *such that* $\vdash T' <: T$

Proof. By induction on the derivation of $M \to M'$. For the case of rule EVAL-FIELD, we employ Lemma 10.2.2. For the case of rule EVAL-METHOD, we employ Lemma 10.2.3 and Lemma 10.2.4 for the substitution of the variables in the method body. The case of rule EVAL-SELECT follows directly. For, the case of rule EVAL-ADVISE, we employ that the advice body is well-typed according to rule DEC-ADVICE. We first employ Lemma 10.2.7 to substitute the types of the involved method for the type variables. Next, we employ Lemma 10.2.4 and Lemma 10.2.5 for the substitution of the term variables and the `proceed` marker. For the case of rule EVAL-CONTEXT, we make use of the induction hypothesis. □

Theorem 10.2.9 (Progress)**.** *If* $\bullet; \bullet \vdash M : T$ *then either* $\vdash \mathrm{val}\ M$ *or* $M \to M'$ *for some* M'*.*

Proof. By induction on the derivation of the typing judgment. We can immediately contradict the cases of rule TERM-VAR and rule TERM-PROCEED because of the empty environment. In case of rule TERM-NEW, the induction hypotheses may be directly applied to the arguments of the **new** expression. For the case of rule TERM-METHOD, progress is trivial using rule EVAL-SELECT. For the cases of rule TERM-FIELD and rule TERM-AMETHOD, the induction hypothesis states that there is either progress for the receiver or the receiver is a value. In the former case, progress is trivial using rule EVAL-CONTEXT. In the latter case, the receiver is a **new** expression, and we may show progress using Lemma 10.2.2 and rule EVAL-FIELD (for field accesses) or using Lemma 10.2.3 and

rule EVAL-METHOD or rule EVAL-ADVISE (for advised method invocations, depending on whether the advice list is empty or not). □

10.3 Related Work: Formal Advice Semantics

10.3.1 Jagadeesan et al.

Jagadeesan, Jeffrey, and Riely (2006) define an aspect extension of FEATHERWEIGHT JAVA that is the main source of inspiration for our formal model in this chapter. Their calculus also extends FEATHERWEIGHT JAVA with advice method declarations that are typed using explicitly-declared type variables, and there is a similar organization of the evaluation of method invocations by means of advised method invocation terms that carry an advice chain.

However, differently from our approach, Jagadeesan et al. do include a specific pointcut language that is directly integrated with the type variables from the advice declaration. We may illustrate this with an example advice method expressed using their formalism:

```
advice a <R extends Number,T> R(T x): exe(R Person.*(T,*)) { proceed(x); }
```

Similar to our proposal, the type variables in an advice declaration are used in the typing of the advice body. Concretely, if L represents the advice body, then the following typing is required in the case of the above declaration (where the type environment Δ contains the assumptions about type variables R and T):

$$\Delta; \mathtt{x} : \mathtt{T}, \mathtt{R}\, \mathbf{proceed}(\mathtt{T}) \vdash L : R' \qquad \Delta \vdash R' <: \mathtt{R}$$

Notice however that the pointcut expression `exe(R Foo.*(T,*))` is not self-contained; it refers to a type variables R and T from the advice declaration. The matching of this pointcut is organized as follows: for an advice declaration $a\langle\bar{X}\ \mathbf{extends}\ \bar{C}\rangle$, the pointcut ϕ will match a method invocation when there exist type values $\bar{V}$ for the variables $\bar{X}$ such that $\bar{V}$ adhere to the bounds $\bar{C}$ and $[\bar{X} \mapsto \bar{V}]\,\phi$ matches the method signature exactly (without subtype variance). The invariant matching is required for type safety, but the binding of the values for the variables $\bar{X}$ allows for a form of quantification: in the case of our example, the pointcut matches any method invocation on class `Person` with a return type that is a subtype of `Number` and any argument type.

We think that this use of the type variables as a quantification mechanism in the pointcuts is conceptually quite distant from actual pointcut languages such as in ASPECTJ. Additionally, it is unclear if the possible values $\bar{V}$ can always be (easily) determined. Worse, while the disjunction and conjunction of pointcuts is supported, their negation is problematic since it will not produce a binding for the type variables Jagadeesan et al. (2006)[as discussed by][p. 16]. In contrast, our model supports arbitrary pointcut languages that correspond with a straightforward compatibility criterion.

Admittedly, the restricted pointcut language also enables the authors to deeply explore the interactions with a generic base language, which is an area that we have not considered here. Jagadeesan et al. define their calculus as an extension of the generic version of FEATHERWEIGHT JAVA, and they consider both the type-carrying and type-erasing implementation of the generics

features. The main difference for an aspect-oriented approach lies in the fact that there is insufficient information for the matching of certain pointcuts in the case of type erasure. The authors describe a system that restricts certain classes of pointcuts for which erasure removes the necessary type information.

Another problem in the model of Jagadeesan et al. is the lacking support for ordinary advice. We have explained how the presence of lower bounds for type variables is crucial in our system to enable this important case. The calculus of Jagadeesan et al. lacks such lower bounds, and the system therefore precludes advice behavior that replaces a join point argument or result (as explained by Jagadeesan et al., 2006, p. 21). Additionally, their type system includes two other important restrictions in comparison to ours: they do not allow the advice methods to change the receiver of a method invocation and they do not support the case of contravariant argument types. Finally, we also contribute a mechanically verified proof of the safety properties of our system.

10.3.2 Clifton and Leavens

Clifton and Leavens (2006) propose MINIMAO$_1$, an imperative core language for studying the semantics of the pointcut/advice mechanism, including around advice and proceed. In comparison to the previous approaches, their emphasis is on a formal description of the dynamic semantics of the advice mechanism, and less on the flexible typing of this mechanism, although a static semantics is provided and proven type sound. The authors consider a context that is somewhat different than ours: MINIMAO$_1$ is specified as an extension of CLASSICJAVA by Flatt et al. (1999) and describes an imperative, reference-based semantics of a language that includes assignment, sequences of statements and null references. It is nevertheless a small-step, operational semantics, and therefore some machinery is required: the evaluation proceeds as rewriting between triples consisting of a term, a representation of a heap and a stack of activation records. Additionally, the syntax of terms is extended with a *loc* form, which is a reference to an object on the heap. While the treatment of an *imperative* language is clearly closer to semantics of concrete aspect-oriented languages such as ASPECTJ, the authors do not report any cases where a model that includes assignment provides insights with respect to typing that are not observable in a purely functional fragment of the language.

The MINIMAO$_1$ language is a fairly complete model of the pointcut/advice mechanism in the ASPECTJ language. It includes aspect modules, around advice methods with proceed facility, advising of method call and execution join points, and a pointcut language with binding primitives (**this**, **target** and **args**) and boolean operators. In addition, it is possible for advice methods to change the receiver object when proceeding (both for call and execution advice). One notable deviation from ASPECTJ, is the structure of such **proceed** invocations. While the pointcut designator of an advice method may freely route the arguments and change the argument order, there is no corresponding routing for **proceed** as in ASPECTJ, and **proceed** is always invoked with values for all the argument positions of the advised method, in their original order. The following example illustrate the syntax for **proceed** employed in MINIMAO$_1$:

```
aspect Asp {
    DSub around(A caller, B arg, C callee): call(D m(..)) &&
        this(A caller) && args(B arg) && target(C callee) {
```

```
        new CSub().proceed(arg);
        ...
    }
}
```

The matching rules of the pointcut language are very strict regarding the type of join points that they match. For example, the pointcut:

$$\textbf{call}(D\ m(..))\ \&\&\ \textbf{args}(B\ x)\ \&\&\ \textbf{target}(C\ y)$$

matches a method call of m on some subtype of C only when C is the class that first introduced m (*i.e.*, m is not inherited nor overridden by C), the return type of m is exactly D and the single argument type of m is exactly B (since variant overriding is not considered by Clifton and Leavens, the result and argument types remain constant). We note that this is quite different from the ASPECTJ matching semantics.

The typing of MINIMAO$_1$ is then organized using a combination of a pointcut typing and a typing of the advice declaration. The pointcut typing assigns types for the positions of the target, the arguments and the result of the matched join points. These types will be exactly equal to the types as they appear in the primitives: in the above example, they will be the types C, B and D respectively. Additionally, the pointcut type tracks the set of variables that a pointcut binds in *all* branches and those that it may bind in *any* of the branches. For the typing of an advice declaration, the following rules are then employed: (i) **proceed** is assigned exactly the pointcut type, (ii) the arguments are assigned exactly the type from the position that they bind, and (iii) the return type of the advice should be a subtype of the pointcut return type. For an advice method bound to the above example pointcut, this means that **proceed** may be invoked using the signature $C \times B \rightarrow D$, which corresponds exactly to the way in which method m may be called (recall that C is the class where m is introduced). This also corresponds with the types assigned to the arguments y and x, which will be C and B respectively. Finally, the advice may return a value which is a subtype of D, which is always valid for a client of this method.

As a conclusion, we may say that MINIMAO$_1$ provides a safe type system but the typing rules are not very flexible: while there is some variance for the return type, we must bind the receiver exactly with the type where a method is first introduced, and we must bind the arguments with the types exactly as they appear in the method definition. This precludes a large number of useful advices that our approach admits. Relaxation of these restrictions is cited as future work by Clifton and Leavens (2006, p.23). Additionally, there is no support for a typing of advice methods using type variables (as a generic advice) which prevents another important class of advice methods.

10.3.3 Lämmel

Lämmel (2002) formally studies a **superimpose** language construct which allows to set up *method-call interception* (MCI) in a class-based object-oriented setting with imperative elements such as assignment and sequences of statements. He defines *dispatch*, *enter* and *exit* join points for method calls, and consecutively considers a system with: (i) basic MCI, where advice behavior is merely triggered at these join points, (ii) interactive MCI, where advice behavior may consult the receiver, arguments and tentative result of the method call, and may

provide a new result to return to the join point client, and (iii) collective MCI, where the advice behavior is triggered by a location pattern that may abstract over multiple method calls (similar to pointcuts).

In the most sophisticated system, we may define advice behavior similar to around advice with a proceed mechanism:

```
superimpose result.intValue() on exit method(getAge) && result(Number)
```

This example will intercept the exit join point of invocations of methods named `getAge` with return type `Number` (exactly). It will apply the `intValue` method on the tentative result returned by the intercepted method (or by the preceding advice behavior) and it will return the new result instead. (Recall that `intValue` is a method defined for class `Number` with return type `Integer`.)

Lämmel defines a safe type system for his approach. A typing of location patterns assigns a bound for the types of the receiver, arguments and the result of the method(s) of which the invocation is matched. This typing is used as an *advice environment* in the typing of the advice expressions. The typing of location patterns allows subtype polymorphism, but only for the case of the result type this is tricky because only results of method invocations may both be consulted *and* updated by advice behavior (whereas the receiver and the arguments may only be consulted, not updated). However, for this case, the typing of the **`result`** pattern assigns a return type that is exactly equal to the return type of the matched methods: recall that a pattern such as **`result`**`(Number)` matches exactly the type `Number`, and not one of its subtypes, such as `Float`. If that were the case, then the new result from the above advice behavior (an `Integer`) might not meet the expectation of the join point client (which may expect a `Float`).

This exact type restriction hinders the flexibility of advice behavior defined for heterogeneous sets of join points, and in comparison to this system, FEATHERWEIGHT STRONGASPECTJ supports type ranges to enable subtype variance for the typing of advice behavior. In addition, we employ type variables to enable advice behavior which always returns the tentative result from the intercepted join point. We also support full around advice behavior, which can update the receiver and argument values in addition to the return value. Finally, Lämmel only informally presents the proofs of the safety properties of his system.

10.3.4 Other Work

Ligatti, Walker, and Zdancewic (2006) define the MINAML system as an extension of simply-typed lambda calculus which models the semantics of pointcuts and advice with two new abstractions: explicitly labeled program points and first-class advice. The labels are used to trigger the advice invocations as well as to mark points that the advice may return to. The advice receives argument values and provides a value when returning, which may be used to simulate some around advice with a proceed mechanism. The authors extend the type theory to include their new concepts, but support neither the subtype nor parametric polymorphism, making the typing rules less flexible than the systems we have considered (basically, the authors only consider advice having exactly the same type as the join point triggering it). Ligatti et al. additionally define type-directed translations from user-friendly external languages. One of these is a core object-oriented language inspired by the object calculus from Abadi and Cardelli

(1996). However, since their object-oriented formalism lacks important concepts such as classes and subtyping, their results are only marginally relevant in our context.

Similarly, Aldrich (2005) defines a typed theory of aspect-oriented programming with a focus on the interaction between aspect and modules, in particular a module sealing operation that hides the internal control-flow points from external advice. The evaluation therefore occurs in a functional context, and without considering subtype nor parametric polymorphism.

Finally, a variety of untyped definitions for the semantics of aspect-oriented mechanisms has also been considered in literature. The approaches by Wand, Kiczales, and Dutchyn (2004) and Douence, Fradet, and Südholt (2002) are only two examples.

Chapter 11

Conclusions

11.1 Summary of the Dissertation

The goal of this dissertation is to improve the language facilities for aspect deployment, in order to enable the advanced reuse of aspect implementations. We focus on the pointcut/advice mechanism and consider both the expressiveness and the safety of the deployment entities.

The deployment mechanisms of current mainstream approaches are analyzed with respect to their deployment expressiveness and three important requirements are identified for the intensive usage of aspects: reuse of deployment code, quantification of deployments, and integration of deployment with dynamic events in the main program. Based on these requirements, the design of first-class deployment procedures is proposed. Such deployment procedures may be parameterized with aspect entities such as pointcuts and advice to abstract and reuse deployment, they integrate control structures to organize deployment quantification, and it is possible to invoke the deployment procedures from the base program with a possible transfer of program values.

A realization of first-class deployment procedures is presented in the form of the EcoSys AOP framework, which organizes the specification of all aspect logic — including the aspect deployment — as code in a general purpose object-oriented programming language, using a number of predefined classes. We define the programming interface of EcoSys and include comprehensive support for the specification of aspect interactions. We demonstrate that EcoSys indeed offers improved deployment expressiveness and a prototype implementation of EcoSys is presented. This prototype employs AspectJ for the atypical purpose of bytecode manipulation. Additionally, we analyze a number of alternative language designs proposed in more recent literature with some impact on the expressiveness of aspect deployment. While these approaches offer a number of advantages similar to EcoSys, they do not provide a complete solution for the requirements we have identified. In some cases, deployment is not an explicit part of investigation, but the motivational examples are caused by deployment issues.

In order to improve the safety of deployment entities, we focus on type systems as a means to verify the compatibility between advice behavior and the context where it is employed. We study the general model of around advice with a proceed mechanism in the context of

a simplified functional language, and we derive two main typing schemes for pointcuts and advice. Ordinary advice is typed using subtype relations, while generic advice is typed using type variables. We also explore the combination of the two typing schemes by considering a quantification of type variables that is bounded by means of subtype relations. We find that generic advice may emulate ordinary advice when type variables may define both a lower and an upper bound may be defined.

We then integrate the proposed typing principles in two practical aspect approaches. We define a typed version of ECOSYS where the typing rules are encoded using the features of JAVA 5 generics such that they are enforced by a standard JAVA compiler. We also propose the STRONGASPECTJ language, an extension of ASPECTJ that incorporates our typing principles using a limited number of modifications. While the typing rules of STRONGASPECTJ are more complex, we demonstrate that any simplification by ASPECTJ is a direct cause for a safety problem or a restriction of valid advice behavior. We have implemented the STRONGASPECTJ language as an extension of the extensible ASPECTBENCH compiler.

Finally, we formally evaluate the proposed typing principles in the context of the FEATHERWEIGHT JAVA calculus. We extend the calculus with a general advice mechanism for method invocations and we model any possible selection of advice through a form of non-determinism in the evaluation. We type the advice construct using type variables with both lower and upper bounds, and we define the conditions for the safe application of an advice to a method type based on inclusion in the range between lower and upper bound. We have developed a proof for the type safety properties of this formal framework using the COQ proof assistant. Since this tool mechanically checks the development according to an established theory, this may be regarded a highly rigorous evaluation of the properties.

11.2 Recapitulation of the Contributions

We summarize the contributions of the dissertation as follows:

- The identification of a set of requirements for the expressive deployment of reusable aspects and a proposal of a design of first-class deployment procedures to meet these requirements. A concrete realization of this design is presented as the ECOSYS AOP framework, with prototype implementation.

  A discussion of the other work in recent literature with an impact on deployment expressiveness confirms the novelty and relevance of this approach.

- The formulation of a set of flexible typing principles for the pointcut/advice mechanism based on the notions of subtype and parametric polymorphism. The typing principles are integrated and implemented in two practical aspect approaches and the soundness of the typing principles is rigorously evaluated using a formal definition and proof of the safety properties.

  An analysis of the safety loopholes and expressiveness restrictions in traditional aspect type systems highlights the improvements of this approach, and a presentation of various examples of practical advice behavior confirms its applicability.

11.3 Future work

11.3.1 Continuations

ECOSYS is developed as a realization of first-class deployment procedures inside a framework-based AOP approach. While this is very beneficial platform for experimentation, and while frameworks are naturally open and extensible systems, there are certain limitations to this approach. We remark that the design of first-class deployment procedures does not necessitate a framework-based approach, and a language-based realization may provide certain benefits such as a more convenient syntax, better tool support, alternative implementation strategies, *etc.* Also, it is currently not obvious from the end result — ECOSYS — which language features are essential for improved deployment expressiveness, since a complete general-purpose programming language is employed for the expression of deployment logic. A language-based version of first-class deployment procedures may render the necessary language elements more explicit and therefore make the arguments from our discourse more apparent.

An open item in the current version of ECOSYS is the typing of its very general interaction resolution mechanism. We have presented a brief, provisional treatment in Section 8.3 that indicates there is some potential, but obviously more work is required in order to obtain a complete proposal. The insights from the discussion of our formal framework in Section 10.2, regarding the compatibility of an advice list and a join point may be of use here.

Another important unresolved issue in Part II of this dissertation is the direct support for the generics features of the base language in the typing of pointcuts and advice. While we provide conclusions that are applicable to parameterized generic types based on the subtype relations that exist between these types, we have otherwise sidestepped the issue. However, it seems that direct support for generics features may allow more expressive advice behavior:

- In case of generic method invocations, it is conceivable for advice behavior of these invocations to provide new type parameter arguments in addition to ordinary value arguments. In the terminology of Chapter 7, this corresponds to the advice function which receives a universal type as its argument, which is the case of first-class polymorphism (Pierce, 2002, Ch.23).

- In case of a join point involving data of specific parameterized class or interface types such as `List<Number>`, `List<Person>` or `List<?>`, it seems beneficial to be able to apply advice behavior that works with `List<X>` for any value of `X`. Allowing the advice based on the observation that the join point have a common structure `List<X>` for some value of `X` corresponds to the case of join points with a special type structure, discussed in Section 7.4.

With respect to the implementation there is also the effect of a type erasure strategy that will make it impossible to match certain pointcuts. This is an important design problem for ASPECTJ 5 (as is reported in Section 9.4.4); the problem is also considered by Jagadeesan et al. (discussed in Section 10.3.1).

11.3.2 Future Research Directions

Aspect interactions beyond shared join points In ECOSYS, the detection and resolution of aspect interactions is restricted to the case where multiple advice methods intercept the same join point. This is the common case that is considered in literature, and as explained by Douence et al. (2002), this involves a non-determinism in the specification of the aspect behavior (without further information, the advice ordering is random or depends on a non-essential property such as the order of declaration). However, aspect interactions may occur in a much wider sense, in any case where the effects of one aspect invalidate the assumptions of another aspect. This problem may be expected to increase as aspects developed by independent teams are combined.

In De Fraine et al. (2008a), we consider the concrete case of control-flow interactions. Control-flow changes introduced by one aspect may modify the flow of control in a system in such a way that it conflicts with other aspects. While these interactions are directly tied to the aspect behavior, they generally do not involve shared join points. This work proposes to tackle this problem by equipping aspect with a formal documentation of its control-flow assumptions. A static control-flow analysis of the bytecode of the composed system allows the automatic verification of certain classes of assumptions.

This technique is quite promising, but the current control-flow policies are conservatively restricted in order to organize the static evaluation. Further research may enable more sophisticated properties to be described. Furthermore, we observe that aspects often assume properties that are not limited to control flow, for example, data flow relations are also relevant.

Intrinsic pointcut semantics The discussion in Section 9.1.1 has illustrated two main strategies that are currently employed to organize the typing of pointcuts. The first strategy is employed when binding variables using primitives such **`this`**, **`target`** and **`args`** in ASPECTJ and STRONGASPECTJ: the pointcut variable is assigned a type based on the use of the variable in the enclosing definition, and the matching behavior is adapted according to this type. This has the downside that the pointcut may match (of fail to match) join points unintentionally, based on the usage in the enclosing definition. The second strategy is employed for the return types of around advice in ASPECTJ and STRONGASPECTJ. Here, the pointcut is expanded to a number of static join point shadows, and for each of these the static type of the join point shadow is checked and violations are reported. This is more intensive to check, and more importantly, it has the downside that type errors in the pointcut become apparent only in the case of certain base programs.

Both of these case have in common that most of the pointcuts are considered as opaque patterns: there is no information for them other than that we may try to match them against each join point (or join point shadow) and this will return a yes/no answer. However, there is typically much more information, consider a pointcut to match the execution of the *setters* of a particular class:

```
execution(void Order+.set*(..))
```

With respect to data types, we know from this definition that the result will always be exactly `void` and the receiver will be a subtype of `Order`. This information is already considered (to a limited extent) for pointcut typing by Lämmel (2002). In addition, Aotani and Masuhara (2007)

propose to include other properties such as the execution join point kind. Also, several other pointcut paradigms have been proposed, which may provide different information. All of this is currently not considered.

The gist of the argument is that pointcuts have a complex semantics and that the proper role of a type system is to provide a simple but meaningful abstraction of these semantics (and to verify that the pointcuts are employed in a manner consistent with the abstraction). By considering pointcuts as opaque entities we do not think this is the case.

Semantics of pointcuts and advice Our formal model of the semantics of the advice mechanism in Chapter 10 focuses on the evaluation of the soundness properties of the proposed typing principles. It is not concerned with developing any particular insight in the fundamental nature of aspect-oriented mechanisms. For example, we model **proceed** invocations in an ad hoc manner, with a custom defined substitution. Although the effect is precisely defined and corresponds to the "real-life" semantics from practical aspect languages, we learn relatively little about this construct in relation to other language constructs, *e.g.*, is **proceed** a method invocation? Some of the other formal evaluations discussed in Section 10.3, but there is no consensus, new models are still being proposed (for example by Schippers et al., 2008) and a number of open questions remain.

Further work on the semantics of pointcuts and advice may help answer these questions and gain a better understanding of the aspect-oriented mechanism. A related topic is the incorporation of aspect deployment in a semantic model in order to capture the core of their effects.

Appendix A

Coq Specification of Featherweight StrongAspectJ

This appendix contains the COQ specification of the FEATHERWEIGHT STRONGASPECTJ calculus. The specification is written in the GALLINA language. The terms of the language form the *calculus of inductive constructions*, which is a higher-order typed lambda calculus. In addition, a number of commands are available to create definitions and declarations. The precise definition of GALLINA is given in the COQ reference manual[1]. The specification employs the List module from the COQ standard library.

A.1 Library Aux

This library provides a number of auxiliary constructs that may be used to study programming languages in Coq. The definition of these constructs is mostly straightforward.

A.1.1 Atoms

We assume there is a set of atoms which have decidable equality, i.e. for any two atoms there is a proof of either their equality or inequality. Several datatypes can easily fulfil this role (for example, the set of natural numbers).

```
Variable atom : Set.
Axiom eq_atom_dec : ∀ (x y : atom), {x = y} + {x ≠ y}.
Notation "x ≐ y" := (eq_atom_dec x y) (at level 67).
```

[1] Available at `http://www.lix.polytechnique.fr/coq/distrib/current/refman/`

A.1.2 Environments

An environment maps atoms to some value of an variable type *A*. We model an environment as a list of pairs: *list* (*atom* × *A*).

```
Notation "x ∈ l" := (List.In x l) (at level 69).
Notation "x ∉ l" := (¬ List.In x l) (at level 69).

Section Environment.

  Variable A : Type.
```

The function *get x E* retrieves the first binding of *x* in environment *E*.

```
  Fixpoint get (x : atom) (E : list (atom × A)) : option A :=
    match E with
    | nil ⇒ None
    | (y, v)::E ⇒ if x ≟ y then Some v else get x E
    end.
```

binds x v E holds when *x* is bound to *v* in *E*. *no_binds x E* holds when there is no binding for *x* in *E*.

```
  Definition binds (x : atom) (v : A) (E : list (atom × A)) : Prop :=
    get x E = Some v.

  Definition no_binds (x : atom) (E : list (atom × A)) : Prop :=
    get x E = None.
```

The functions *keys E* and *dom E* retrieve the atoms that are bound in the environment *E*. The function *imgs E* retrieves the values in *E*.

```
  Definition keys (E : list (atom × A)) : list atom :=
    List.map (@fst atom A) E.

  Definition dom := keys.

  Definition imgs (E : list (atom × A)) : list A :=
    List.map (@snd atom A) E.
```

ok E holds when the environment *E* contains no duplicate bindings.

```
  Inductive ok : list (atom × A) → Prop :=
  | ok_nil: ok nil
  | ok_cons: ∀ E x v,
      ok E → no_binds x E → ok ((x, v) :: E).
```

forall_env P E holds when proposition *P x v* holds for all bindings (*x*, *v*) in environment *E*.

```
  Section forall_env.

    Variable P : atom → A → Prop.

    Inductive forall_env : list (atom × A) → Prop :=
    | fa_nil: forall_env nil
    | fa_cons: ∀ E x v, forall_env E → P x v → forall_env ((x, v) :: E).
```

```
    End forall_env.
End Environment.
```

A.1.3 Zipping and list properties

```
Section Zip.
    Variable B : Type.
```

zip al bl abl will match up atom list *al* with the value list *bl* to produce the environment *abl*.

```
    Inductive zip : list atom → list B → list (atom × B) → Prop :=
    | zip_nil: zip nil nil nil
    | zip_cons: ∀ a al b bl abl,
        zip al bl abl →
        zip (a :: al) (b :: bl) ((a, b) :: abl).
```

zip3 al bl cl abcl will match up atom list *al* with the value lists *bl* and *cl* to produce the environment *abcl*.

```
    Inductive zip3 (C : Type) :
        list atom → list B → list C → list (atom × (B × C)) → Prop :=
    | zip3_nil: zip3 nil nil nil nil
    | zip3_cons: ∀ a al b bl c cl abcl,
        zip3 al bl cl abcl →
        zip3 (a :: al) (b :: bl) (c :: cl) ((a, (b, c)) :: abcl).
```

forall_list P bl holds when proposition *P b* holds for all elements *b* of list *bl*.

```
    Inductive forall_list (P : B → Prop) : list B → Prop :=
    | fal_nil: forall_list P nil
    | fal_cons: ∀ b bl, forall_list P bl → P b → forall_list P (b :: bl).

End Zip.
```

A.2 Library Definitions

This library contains the actual Featherweight StrongAspectJ language definition. The definition is divided up between syntax, auxiliaries, evaluation and typing.

A.2.1 Syntax

Lexical categories

Names of variables, type variables, fields, methods, classes and advice methods are atoms (their equality is decidable).

```
Definition var := atom.
Definition tvar := atom.
Definition fname := atom.
Definition mname := atom.
Definition cname := atom.
Definition aname := atom.
```

The names *this* and *Object* are predefined. We simply assume that these names exist.

```
Parameter this : var.
Parameter Object : cname.
```

Type and term expressions

There are three kinds of types: non-variables types (represented by a class name), variable types (represented by a type variable) and the null type.

```
Inductive typ :=
| typ_nvar : cname → typ
| typ_var : tvar → typ
| typ_null : typ.
```

We define *exp* as a single term expression, and *exps* as a list of terms. We include the usual expression forms from Featherweight Java (variable reference, field get, method invocation and object creation). In addition, there is a method invocation with a list of advice names (written *alist*) and a proceed invocation.

```
Notation alist := (list aname).

Inductive exp : Set :=
| e_var : var → exp
| e_field : exp → fname → cname → exp
| e_meth : exp → mname → cname → exps → exp
| e_meth_adv : exp → mname → cname → alist → exps → exp
| e_new : cname → exps → exp
| e_proceed : exp → exps → exp
with exps : Set :=
| el_nil : exps
| el_cons : exp → exps → exps.
```

Environments, class and advice tables

A *var_env* declares a number of variables and their types. A *pcd_env* optionally declares a proceed type. *env* combines both environments.

```
Notation var_env := (list (var × typ)).
Notation pcd_env := (option (typ × typ × list typ)).
Notation env := (var_env × pcd_env).
```

A *var_benv* binds variables to expressions. A *pcd_benv* optionally binds proceed to an advised method. *benv* combined both bindings.

```
Notation var_benv := (list (var × exp)).
Notation pcd_benv := (option (mname × cname × alist)).
Notation benv := (var_benv × pcd_benv).
```

A *tenv* defines both lower and upper bound types for a number of type variables.

```
Notation tenv := (list (tvar × (typ × typ))).
```

flds and *mths* map the names of fields and methods to their definitions.

```
Notation flds := (list (fname × typ)).
Notation mths := (list (mname × (typ × var_env × exp))).
```

ctable and *atable* map the names of classes and advice methods to their definitions. A class definition consists of a parent class and a number of fields and methods. An advice method definition consists of:

- a type variable with bounds for the result
- a variable and a type variable with bounds for the receiver
- multiple variables and multiple type variables with bounds for the arguments
- a body expression

```
Notation ctable := (list (cname × (cname × flds × mths))).
Notation atable := (list (aname × (tvar × typ × typ ×
    var × tvar × typ × typ ×
    list var × list tvar × list typ × list typ ×
    exp))).
```

We assume a fixed class table *CT* and a fixed advice table *AT*.

```
Parameter CT : ctable.
Parameter AT : atable.
```

A.2.2 Auxiliaries

Field and method lookup

field C f t holds if a field named *f* with type *t* is defined for class *C* in the class hierarchy.

```
Inductive fields : cname → flds → Prop :=
| fields_obj : fields Object nil
| fields_other : ∀ C D fs fs' ms,
    binds C (D, fs, ms) CT →
    fields D fs' →
    fields C (fs' ++ fs).

Definition field (C : cname) (f : fname) (t : typ) : Prop :=
    ∃ fs, fields C fs ∧ binds f t fs.
```

method C m mdecl holds if a method named *m* with declaration *mdecl* is defined for class *C* in the class hierarchy.

```
Inductive method : cname → mname → typ × var_env × exp → Prop :=
```

```
| method_this : ∀ C D fs ms m mdecl,
    binds C (D, fs, ms) CT →
    binds m mdecl ms →
    method C m mdecl
| method_super : ∀ C D fs ms m mdecl,
    binds C (D, fs, ms) CT →
    no_binds m ms →
    method D m mdecl →
    method C m mdecl.
```

Zipping

ezip lenv lexp lzip will match up the atom list *lenv* with the expression list *lexp* in list *lzip*.

```
Inductive ezip : list atom → exps → list (atom × exp) → Prop :=
| ez_nil : ezip nil el_nil nil
| ez_cons : ∀ lenv lexp lzip a e,
    ezip lenv lexp lzip →
    ezip (a :: lenv) (el_cons e lexp) ((a, e) :: lzip).
```

Term substitution

subst_exp E e returns the term expression *e* where any occurrances of variables or proceed have been replaced by their bindings in environment *E*.

```
Fixpoint subst_exp (E : benv) (e : exp) {struct e} : exp :=
    match e with
    | e_var v ⇒
        match get v (fst E) with
        | Some e' ⇒ e'
        | None ⇒ e_var v
        end
    | e_field e0 f C ⇒ e_field (subst_exp E e0) f C
    | e_meth e0 m C es ⇒ e_meth (subst_exp E e0) m C (subst_exps E es)
    | e_meth_adv e0 m C al es ⇒
        e_meth_adv (subst_exp E e0) m C al (subst_exps E es)
    | e_new C es ⇒ e_new C (subst_exps E es)
    | e_proceed e0 es ⇒
        match (snd E) with
        | Some (m, C, al) ⇒
            e_meth_adv (subst_exp E e0) m C al (subst_exps E es)
        | None ⇒ e_proceed (subst_exp E e0) (subst_exps E es)
        end
    end
with subst_exps (E : benv) (es : exps) {struct es} : exps :=
    match es with
```

```
| el_nil ⇒ el_nil
| el_cons e es0 ⇒ el_cons (subst_exp E e) (subst_exps E es0)
end.
```

Well-formed types and environments

ok_type T t holds when *t* is a well-formed type in type environment *T*.

```
Inductive ok_type : tenv → typ → Prop :=
| okt_obj: ∀ T, ok_type T (typ_nvar Object)
| okt_nvar: ∀ T C, C ∈ dom CT → ok_type T (typ_nvar C)
| okt_var: ∀ T X, X ∈ dom T → ok_type T (typ_var X)
| okt_null: ∀ T, ok_type T typ_null.

Definition ok_types (T : tenv) (ts : list typ) :=
    forall_list (ok_type T) ts.
```

ok_tenv T holds if type environment *T* is well-formed. *ok_var_env T E* holds if variable environment *E* is well-formed in *T*. *ok_pcd_env T P* holds if proceed environment *P* is well-formed in *T*. Finally, *ok_env T* (*E*, *P*) holds if the combination of the two holds.

```
Inductive ok_tenv : tenv → Prop :=
| okte_nil : ok_tenv nil
| okte_cons : ∀ T X S U,
    ok_tenv T →
    no_binds X T →
    ok_type T S →
    ok_type T U →
    ok_tenv ((X, (S, U)) :: T).

Inductive ok_var_env : tenv → var_env → Prop :=
| okve_nil : ∀ T, ok_tenv T → ok_var_env T nil
| okve_cons : ∀ T E x t,
    ok_var_env T E →
    no_binds x E →
    ok_type T t →
    ok_var_env T ((x, t) :: E).

Inductive ok_pcd_env : tenv → pcd_env → Prop :=
| okpe_none : ∀ T, ok_tenv T → ok_pcd_env T None
| okpe_some : ∀ T t t0 ts,
    ok_tenv T →
    ok_type T t →
    ok_types T ts →
    ok_pcd_env T (Some (t, t0, ts)).

Definition ok_env (T : tenv) (EE : env) :=
    match EE with (E, P) ⇒ ok_var_env T E ∧ ok_pcd_env T P end.
```

Subtyping

extends C D holds if *C* is a direct subclass of *D*.

```
Definition extends (C D : cname) : Prop :=
  ∃ fs, ∃ ms, binds C (D, fs, ms) CT.
```

sub T s u holds if *s* is a subtype of *u* under type environment *T*. The subtype relation is the reflexive, transitive closure of the following direct subtype relations:

- subclassing, for non-variable types
- the bounds declared in the type environment, for variable types
- any type, for the null type

```
Inductive sub : tenv → typ → typ → Prop :=
| sub_refl : ∀ T t, sub T t t
| sub_trans : ∀ T t1 t2 t3, sub T t1 t2 → sub T t2 t3 → sub T t1 t3
| sub_extends : ∀ T C D, extends C D → sub T (typ_nvar C) (typ_nvar D)
| sub_up : ∀ T X S U, binds X (S, U) T → sub T (typ_var X) U
| sub_low : ∀ T X S U, binds X (S, U) T → sub T S (typ_var X)
| sub_null : ∀ T t, sub T typ_null t.

Inductive subs : tenv → list typ → list typ → Prop :=
| subs_nil : ∀ T, subs T nil nil
| subs_cons : ∀ T t t' ts ts',
    subs T ts ts' → sub T t t' → subs T (t :: ts) (t' :: ts').
```

Advice compatibility

ok_adv_for a c m holds if the advice named *a* is compatible with method *m* of class *c*.

```
Inductive ok_adv_for : aname → cname → mname → Prop :=
| okadv_meth : ∀ a c m Y R S x0 X P0 Q0 xs XS Ps Qs l T UE e,
    binds a (Y, R, S, x0, X, P0, Q0, xs, XS, Ps, Qs, l) AT →
    method c m (T, UE, e) →
    sub nil R T →
    sub nil T S →
    sub nil P0 (typ_nvar c) →
    sub nil (typ_nvar c) Q0 →
    subs nil Ps (imgs UE) →
    subs nil (imgs UE) Qs →
      ok_adv_for a c m.

Inductive ok_advs_for : alist → cname → mname → Prop :=
| okadvs_nil : ∀ c m, ok_advs_for nil c m
| okadvs_cons : ∀ ad ads c m,
    ok_adv_for ad c m →
    ok_advs_for ads c m →
    ok_advs_for (ad :: ads) c m.
```

A.2.3 Evaluation

Evaluation contexts

We model evaluation contexts as functions of type *exp* → *exp*. *exp_context EE* holds if *EE* is an evaluation context. Basically, any subexpression of an expression is an evaluation context.

```
Inductive exps_context : (exp → exps) → Prop :=
| esc_head : ∀ es,
    exps_context (fun e ⇒ el_cons e es)
| esc_tail : ∀ e EE,
    exps_context EE →
    exps_context (fun e0 ⇒ el_cons e (EE e0)).

Inductive exp_context : (exp → exp) → Prop :=
| ec_field_arg0 : ∀ f C,
    exp_context (fun e0 ⇒ e_field e0 f C)
| ec_meth_arg0 : ∀ m C es,
    exp_context (fun e0 ⇒ e_meth e0 m C es)
| ec_meth_args : ∀ m e0 C EE,
    exps_context EE →
    exp_context (fun e ⇒ e_meth e0 m C (EE e))
| ec_meth_adv_arg0 : ∀ m C al es,
    exp_context (fun e0 ⇒ e_meth_adv e0 m C al es)
| ec_meth_adv_args : ∀ m e0 C al EE,
    exps_context EE →
    exp_context (fun e ⇒ e_meth_adv e0 m C al (EE e))
| ec_new_args : ∀ C EE,
    exps_context EE →
    exp_context (fun e ⇒ e_new C (EE e))
| ec_pcd_arg0 : ∀ es,
    exp_context (fun e0 ⇒ e_proceed e0 es)
| ec_pcd_args : ∀ e0 EE,
    exps_context EE →
    exp_context (fun e ⇒ e_proceed e0 (EE e)).
```

Evaluation

eval e e' holds when term expression *e* reduces to *e'* in one step. The evaluation of a method invocation occurs by selecting a number of compatible advice methods (*eval_select*), applying the advice behavior of each with proceed bound to the remaining advice chain (*eval_advise*), and finally executing the method body itself (*eval_method*).

```
Inductive eval : exp → exp → Prop :=
| eval_field : ∀ C D fs es f e fes,
    fields C fs →
    ezip (keys fs) es fes →
```

```
      binds f e fes →
      eval (e_field (e_new C es) f D) e
| eval_select : ∀ e0 m C es ads,
      ok_advs_for ads C m →
      eval (e_meth e0 m C es) (e_meth_adv e0 m C ads es)
| eval_advise : ∀ e0 m C a ads es Y R S x0 X P0 Q0 xs XS Ps Qs l ves,
      binds a (Y, R, S, x0, X, P0, Q0, xs, XS, Ps, Qs, l) AT →
      ezip xs es ves →
      eval
          (e_meth_adv e0 m C (a :: ads) es)
          (subst_exp ((x0, e0) :: ves, Some (m, C, ads)) l)
| eval_meth : ∀ C D m t E e es ves es0,
      method C m (t, E, e) →
      ezip (keys E) es ves →
      eval
          (e_meth_adv (e_new C es0) m D nil es)
          (subst_exp ((this, (e_new C es0)) :: ves, None) e)
| eval_context : ∀ EE e e',
      eval e e' →
      exp_context EE →
      eval (EE e) (EE e').
```

A.2.4 Typing

Term expression typing

typing T E e t holds when expression *e* has type *t* in term environment *E* and type environment *T*. *wide_typing T E e t* holds when *e* has a subtype of *t*.

```
Inductive typing : tenv → env → exp → typ → Prop :=
| t_var : ∀ T E P v t,
      ok_env T (E, P) →
      binds v t E →
      typing T (E, P) (e_var v) t
| t_field : ∀ T E e0 C f t,
      field C f t →
      wide_typing T E e0 (typ_nvar C) →
      typing T E (e_field e0 f C) t
| t_meth : ∀ T E E0 e0 b C t m es,
      method C m (t, E0, b) →
      wide_typing T E e0 (typ_nvar C) →
      wide_typings T E es (imgs E0) →
      typing T E (e_meth e0 m C es) t
| t_meth_adv : ∀ T E E0 e0 b C t m es ads,
      method C m (t, E0, b) →
```

```
      wide_typing T E e0 (typ_nvar C) →
      wide_typings T E es (imgs E0) →
      ok_advs_for ads C m →
      typing T E (e_meth_adv e0 m C ads es) t
| t_new : ∀ T E C fs es,
      fields C fs →
      wide_typings T E es (imgs fs) →
      typing T E (e_new C es) (typ_nvar C)
| t_proceed : ∀ T E P e0 es s q0 qs,
      P = Some (s, q0, qs) →
      wide_typing T (E, P) e0 q0 →
      wide_typings T (E, P) es qs →
      typing T (E, P) (e_proceed e0 es) s

with wide_typing : tenv → env → exp → typ → Prop :=
| wt_sub : ∀ T E e t t',
      typing T E e t → sub T t t' → wide_typing T E e t'

with wide_typings : tenv → env → exps → list typ → Prop :=
| wts_nil : ∀ T E,
      ok_env T E →
      wide_typings T E el_nil nil
| wts_cons : ∀ T E E0 es e t,
      wide_typings T E es E0 →
      wide_typing T E e t →
      wide_typings T E (el_cons e es) (t :: E0).
```

Declaration typing

ok_meth C D m t E e holds when (*t*, *E*, *e*) is a valid method declaration for method *m* in class *C* with parent *D*.

```
Definition can_override (D : cname) (m : mname) (t : typ) (E : var_env) : Prop :=
    ∀ t' E' e, method D m (t', E', e) →
        sub nil t t' ∧ subs nil (imgs E') (imgs E).

Definition ok_meth (C D : cname) (m : mname) (t : typ) (E : var_env) (e : exp) : Prop :=
    can_override D m t E ∧ ok_type nil t ∧
    wide_typing nil ((this, (typ_nvar C)) :: E, None) e t.

Definition ok_meth' (C D : cname) (m : mname) (v : typ × var_env × exp) : Prop :=
    match v with (t, E, e) ⇒ ok_meth C D m t E e end.
```

ok_advice Y R S x0 X0 P0 Q0 xs XS Ps Qs l holds when it is valid to declare an advice with:

- *Y*, *R*, *S* as the type variable and bounds for the result
- *x0*, *X0*, *P0*, *Q0* as the variable, type variable and bounds for the receiver

- *xs*, *XS*, *Ps*, *Qs* as the variables, type variables and bounds for the arguments
- *l* as the body expression

ok_atable atb holds when *atb* is a valid advice table.

```
Inductive ok_advice : tvar → typ → typ →
    var → tvar → typ → typ →
    list var → list tvar → list typ → list typ →
    exp → Prop :=
| decl_adv: ∀ Y R S x0 X0 P0 Q0 xs XS Ps Qs l XPQs xXs,
    zip3 XS Ps Qs XPQs →
    zip xs (List.map typ_var XS) xXs →
    wide_typing
        ((Y, (R, S)) :: (X0, (P0, Q0)) :: XPQs)
        ((x0, typ_var X0) :: xXs,
            Some (typ_var Y, typ_var X0, List.map typ_var XS))
        l (typ_var Y) →
    ok_advice Y R S x0 X0 P0 Q0 xs XS Ps Qs l.

Definition ok_advice' (a : aname) v : Prop :=
    match v with
    | (Y, R, SS, x0, X0, P0, Q0, xs, XS, Ps, Qs, l) ⇒
        ok_advice Y R SS x0 X0 P0 Q0 xs XS Ps Qs l
    end.

Definition ok_atable (atb : atable) :=
    ok atb ∧ forall_env ok_advice' atb.
```

ok_class C D fs ms holds when it is valid to define class *C* with parent *D*, fields *fs* and methods *ms*. *ok_ctable ct* holds when *ct* is a well-formed class table.

```
Definition ok_class (C : cname) (D : cname) (fs : flds) (ms : mths) : Prop :=
    (∀ fs', fields D fs' → ok_var_env nil (fs' ++ fs)) ∧
    (D = Object ∨ D ∈ dom CT) ∧
    forall_env (ok_meth' C D) ms.

Definition ok_class' (C : cname) (v : cname × flds × mths) : Prop :=
    match v with (D, flds, mths) ⇒ ok_class C D flds mths end.

Definition ok_ctable (ct : ctable) :=
    ok ct ∧ forall_env ok_class' ct.
```

A.2.5 Properties

We conclude the definition of the calculus with a definition of the safety properties that are proven in the other parts of the development.

value e holds when the term expression *e* represents a value. Values are terms that consist only of *e_new* expressions.

```
Inductive value : exp → Prop :=
| value_new : ∀ cn es, values es → value (e_new cn es)
with values : exps → Prop :=
| values_nil : values el_nil
| values_cons : ∀ e el, value e → values el → values (el_cons e el).
```

The following module defines the hypotheses of the safety argument. We assume that *Object* is not defined in the class table *CT*, that class table *CT* is well-formed, and that advice table *AT* is well-formed.

```
Module Type HYPS.
    Parameter ct_noobj: Object ∉ dom CT.
    Parameter ok_ct: ok_ctable CT.
    Parameter ok_at: ok_atable AT.
End HYPS.
```

Safety of the language may be demonstrated through an implementation of the following module type: given the above hypotheses, it provides the properties of preservation and progress.

```
Module Type SAFETY (H : HYPS).
    Parameter preservation: ∀ E e e' t,
        typing nil E e t → eval e e' → wide_typing nil E e' t.
    Parameter progress: ∀ e t,
        typing nil (nil, None) e t → value e ∨ (∃ e', eval e e').
End SAFETY.
```

Bibliography

Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, USA, 1996. Cited on pages 80 and 173.

Mehmet Akşit, editor. *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, March 2003. ACM Press. Cited on pages 200, 202, 203, and 204.

Jonathan Aldrich. Open modules: Modular reasoning about advice. In Black (2005), pages 144–168. Cited on pages 41 and 174.

Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proc. 20th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2005)*, pages 345–364. ACM Press, 2005. Cited on pages 40 and 66.

Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 161–168, New York, NY, USA, 1990. ACM. Cited on page 101.

Tomoyuki Aotani and Hidehiko Masuhara. Towards a type system for detecting never-matching pointcut compositions. In William Harrison, editor, *Proceedings of the 6th Workshop on Foundations of Aspect-Oriented Languages, (FOAL 2007)*, pages 23–26. ACM, 2007. Cited on page 178.

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In Awais Rashid and Mehmet Akşit, editors, *Trans. on Aspect-Oriented Software Development I (TAOSD)*, volume 3880 of *LNCS*, pages 135–173. Springer Verlag, 2006. Cited on page 55.

Ken Arnold. Generics considered harmful. Blog post available at `http://weblogs.java.net/blog/arnold/archive/2005/06/generics_consid_1.html`, June 2005. Cited on page 106.

Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In Tarr (2005), pages 87–98. Cited on page 145.

Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North Holland, Amsterdam, The Netherlands, second edition, 1985. Cited on page 80.

Don Batory, Charles Consel, and Walid Taha, editors. *Proc. 1st ACM SIGPLAN/SIGSOFT Conf. on Generative Programming and Component Engineering (GPCE-2002)*, volume 2487 of *LNCS*, October 2002. Springer Verlag. Cited on pages 197 and 199.

Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In Masuhara and Rashid (2006), pages 51–62. Cited on pages 3 and 41.

Lodewijk Bergmans and Mehmet Akşit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001. Cited on page 13.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Texts in Theoretical Computer Science*. Springer, 2004. Cited on page 167.

Andrew P. Black, editor. *Proc. 19th European Conf. on Object-Oriented Programming (ECOOP-2005)*, volume 3586 of *LNCS*, July 2005. Springer Verlag. Cited on pages 195, 201, and 203.

Joshua Bloch. The closures controversy. Presented at JavaPolis 2007, December 2007. Cited on page 106.

Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In Lieberherr (2004), pages 83–92. Cited on pages 31 and 49.

Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting virtual machine techniques for seamless aspect support. In *Proc. of International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA) 2006*. ACM, 2006. Cited on page 49.

Eric Bodden, Wes Isberg, and Adrian Colyer. Typing of around closures can lead to class-cast exceptions. AspectJ bug report #159390, available at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=159390`, September 2006. Cited on page 144.

Ron Bodkin. AOP@Work: Performance monitoring with AspectJ. Technical report, IBM Developer Works, September 2005. Available at `http://www-128.ibm.com/developerworks/java/library/j-aopwork10/`. Cited on page 10.

Jonas Bonér and Alexandre Vasseur. AspectWerkz: simple, high-performant, dynamic, lightweight and powerful AOP for Java. Home page at `http://aspectwerkz.codehaus.org/`, 2004. Cited on page 21.

Jonas Bonér, Alexandre Vasseur, and Joakim Dahlstedt. JRockit JVM support for AOP. Technical report, BEA dev2dev, August 2005. Available at `http://www.oracle.com/technology/pub/articles/dev2arch/2005/08/jvm_aop_1.html`. Cited on pages 31 and 49.

Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications and European Conf. on Object-Oriented Programming (OOPSLA-ECOOP 1990)*, pages 303–311. ACM Press, 1990. Cited on page 17.

Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *Proc. of OOPSLA '93*, pages 215–230. ACM Press, 1993. Cited on pages 101 and 133.

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA '98*, pages 183–200. ACM Press, October 1998. Cited on page 98.

Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In Batory et al. (2002), pages 110–127. Cited on page 45.

Johan Brichau, Shigeru Chiba, David Lorenz, Éric Tanter, and Kris De Volder, editors. *Proc. of Open and Dynamic Aspect Languages Workshop*, March 2006. Cited on pages 198 and 204.

Johan Brichau et al. Survey of aspect-oriented languages and execution models. Deliverable 12, Project IST-2-004349-NOE "AOSD-Europe", May 2005. Cited on page 13.

Bill Burke et al. JBoss Aspect-Oriented Programming. Home page at `http://www.jboss.org/products/aop`, 2004. Cited on page 21.

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, New York, NY, USA, 1989. ACM. Cited on page 100.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985. Cited on pages 4, 77, 78, and 80.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. Cited on page 80.

Pierre Castéran, Hugo Herbelin, Florent Kirchner, Benjamin Monate, and Julien Narboux. Coq version 8.2 for the clueless (frequently asked questions). Available at `http://coq.inria.fr/`, 2009. Cited on page 167.

Shigeru Chiba. Javassist — a reflection-based programming wizard for Java. In *Proc. of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998. Cited on pages 50 and 68.

Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2): 56–68, June 1940. Cited on page 80.

Curtis Clifton and Gary T. Leavens. MiniMAO1: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, December 2006. Cited on pages 164, 171, and 172.

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proceedings of the 15th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 130–145. ACM Press, October 2000. Cited on page 16.

Adrian Colyer. AspectJ. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 123–143. Addison-Wesley, Boston, 2005a. Cited on pages 12 and 18.

Adrian Colyer. Implementing caching with AspectJ. Blog entry: `http://www.aspectprogrammer.org/blogs/adrian/`, June 2004. Cited on pages 10 and 130.

Adrian Colyer. Aspect library discussion at AOSD 2005. Blog entry: `http://www.aspectprogrammer.org/blogs/adrian/`, March 2005b. Cited on page 1.

Adrian Colyer et al. The AspectJ 5 development kit developer's notebook. Available at `http://www.eclipse.org/aspectj/doc/released/adk15notebook/`, December 2005. Cited on pages 21, 122, and 151.

Adrian Colyer et al. The AspectJ development environment guide. Available at `http://www.eclipse.org/aspectj/doc/released/devguide/`, 2002. Cited on page 145.

Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988. Cited on page 167.

Carlos A. Cunha, João L. Sobral, and Miguel P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In Masuhara and Rashid (2006), pages 134–145. Cited on pages 1 and 10.

Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. PolyAML: A polymorphic aspect-oriented functional programmming language. In Pierce (2005), pages 306–319. Cited on page 149.

Bruno De Fraine and Mathieu Braem. Requirements for reusable aspect deployment. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, pages 176–183. Springer Berlin / Heidelberg, December 2007. Cited on page 4.

Bruno De Fraine, Wim Vanderperren, and Davy Suvée. Towards a better modularization of entities in AOP languages. In Kris Gybels, Maja D'Hondt, Istvan Nagy, and Rémi Douence, editors, *Proc. of European Interactive Workshop on Aspects in Software (EIWAS) 2005*, September 2005a. Cited on page 4.

Bruno De Fraine, Wim Vanderperren, Davy Suvée, and Johan Brichau. Jumping aspects revisited. In Robert E. Filman, Michael Haupt, and Robert Hirschfeld, editors, *Dynamic Aspects Workshop*, pages 77–86, March 2005b. Cited on page 31.

Bruno De Fraine, Wim Vanderperren, and Davy Suvée. Motivations for framework-based AOP. In Brichau et al. (2006). Cited on page 4.

Bruno De Fraine, Wim Vanderperren, and Davy Suvée. Eco: A flexible, open and type-safe framework for aspect-oriented programming. Technical Report SSEL 01/2006/a, Vrije Universiteit Brussel, January 2006b. http://ssel.vub.ac.be/files/defraine-eco06a.pdf. Cited on page 4.

Bruno De Fraine, Mario Südholt, and Viviane Jonckers. A formal semantics of flexible and safe pointcut/advice bindings. Technical Report SSEL 02/2007/a, Vrije Universiteit Brussel, October 2007. Available at http://ssel.vub.ac.be/files/formal07a.pdf. Cited on pages 5 and 153.

Bruno De Fraine, Pablo Daniel Quiroga, and Viviane Jonckers. Management of aspect interactions using statically-verified control-flow relations. In *Proceeding of Aspect, Dependencies and Interactions (ADI08)*, July 2008a. Cited on page 178.

Bruno De Fraine, Mario Südholt, and Viviane Jonckers. StrongAspectJ: Flexible and safe pointcut/advice bindings. In Mezini (2008), pages 60–71. Cited on page 5.

Bart De Win, Bart Vanhaute, and Bart De Decker. How aspect-oriented programming can help to build secure software. *Informatica*, 26(2):141–149, 2001. Cited on page 10.

Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp object system: an overview. In *European conference on object-oriented programming (ECOOP '87)*, pages 151–170, London, UK, 1987. Springer-Verlag. ISBN 0-387-18353-1. Cited on page 16.

Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer Verlag, New York, 1982. Originally published as EWD447, August 1974. Cited on page 9.

Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka (2001), pages 170–186. Cited on page 40.

Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Batory et al. (2002), pages 173–188. Cited on pages 29, 45, 66, 174, and 178.

Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr (2004), pages 141–150. Cited on page 66.

Bruce Eckel. Generics. Blog post available at http://www.artima.com/weblogs/viewpost.jsp?thread=117200, June 2005. Cited on page 106.

Erik Ernst. Family polymorphism. In J. L. Knudsen, editor, *Proc. 15th European Conference on Object-Oriented Programming (ECOOP-2001)*, pages 303–326, Berlin, June 2001. Springer-Verlag. Cited on page 58.

Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000. Cited on pages 10 and 34.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, chapter A Programmer's Reduction Semantics for Classes and Mixins, pages 241–270. Springer, 1999. Cited on page 171.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995. Cited on pages 14, 39, 59, 108, and 131.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, June 2005. Cited on pages 40, 97, 102, 104, 136, and 138.

William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006. Cited on page 41.

Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In Akşit (2003), pages 60–69. Cited on page 40.

Richard Hamlet. *Encyclopedia of Software Engineering*, chapter Random testing, pages 970–978. Wiley, 1994. Cited on page 46.

Stefan Hanenberg and Rainer Unland. Parametric introductions. In Akşit (2003), pages 80–89. Cited on page 151.

Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In Matsuoka (2002), pages 161–173. Cited on pages 1 and 131.

Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In Masuhara and Rashid (2006), pages 63–74. Cited on pages 10 and 41.

Michael Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Darmstadt University of Technology, Germany, December 2005. Cited on page 49.

Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Lieberherr (2004), pages 26–35. Cited on page 115.

Robert Hirschfeld. AspectS: Aspect-oriented programming with Squeak. In Mehmet Akşit, Mira Mezini, and Rainer Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World (NODe 2002)*, volume 2591 of *LNCS*, pages 216–232, Berlin, 2003. Springer Verlag. Cited on pages 70 and 72.

Robert Hirschfeld and Ralf Lämmel. Reflective designs. *IEE Proceedings Software*, 2004. Special Issue on Reusable Software Libraries. Cited on pages 70 and 131.

Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In Boris Magnusson, editor, *ECOOP 2002: Proceedings of the 16th European Conference on Object-Oriented Programming*, volume 2374 of *LNCS*, pages 441–469. Springer, 2002. Cited on pages 102 and 104.

Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. of OOPSLA '99*, pages 132–146. ACM Press, October 1999. Cited on pages 80, 154, 157, 158, 159, and 160.

Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems (TOPLAS)*, 23 (3):396–450, May 2001. Cited on page 154.

Wes Isberg. Check out library aspects with AspectJ 5. Technical report, IBM Developer Works, January 2006. URL `http://www.ibm.com/developerworks/java/library/j-aopwork14/`. Cited on page 1.

Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63(3):267–296, December 2006. Cited on pages 145, 153, 170, 171, and 177.

Rod Johnson et al. Spring Java/J2EE Application Framework. Home page at `http://www.springframework.org/`, 2004. Cited on page 21.

Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In Dave Thomas, editor, *Proc. 20th European Conf. on Object-Oriented Programming (ECOOP-2006)*, volume 4067 of *LNCS*, pages 501–525. Springer Verlag, July 2006. Cited on page 41.

Gregor Kiczales, editor. *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, April 2002. ACM Press. Cited on pages 202 and 204.

Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In Black (2005), pages 195–213. Cited on page 41.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox PARC, February 1997a. Cited on page 1.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. 11th European Conf. on Object-Oriented Programming (ECOOP-1997)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997b. Cited on page 10.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen L. Knudsen, editor, *Proc. 15th European Conf. on Object-Oriented Programming (ECOOP-2001)*, volume 2072 of *LNCS*, pages 327–353, Berlin, June 2001a. Springer Verlag. Cited on pages 12, 18, and 142.

Gregor Kiczales et al. Aspect-oriented programming in Java with AspectJ. Presented at O'Reilly EJ Conference, March 2001b. Cited on pages 10 and 11.

Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Syntax directed program modularization. In Pierpaolo Degano and Erik Sandewall, editors, *Integrated Interactive Computing Systems*. North Holland, 1983. Cited on page 16.

Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003. Cited on pages 10 and 132.

Ralf Lämmel. A semantical approach to method-call interception. In Kiczales (2002), pages 41–55. Cited on pages 172, 173, and 178.

Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic programming meets adaptive programming. In Akşit (2003), pages 168–177. Cited on page 14.

Karl Lieberherr, editor. *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, March 2004. ACM Press. Cited on pages 196, 199, 200, and 203.

Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999. Cited on pages 1 and 21.

Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. Cited on page 13.

Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, September 1989. Cited on page 13.

Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 63(3):240–266, December 2006. Cited on page 173.

Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In Gabor Karsai and Eelco Visser, editors, *Proc. 3rd Int' Conf. on Generative Programming and Component Engineering (GPCE-2004)*, volume 3286 of *LNCS*, pages 55–74, Berlin, October 2004. Springer Verlag. Cited on pages 150 and 151.

Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In Luca Cardelli, editor, *Proc. 17th European Conf. on Object-Oriented Programming (ECOOP-2003)*, volume 2743 of *LNCS*, pages 2–28. Springer Verlag, July 2003. Cited on pages 12 and 13.

Hidehiko Masuhara and Awais Rashid, editors. *Proc. 5th Int' Conf. on Aspect-Oriented Software Development (AOSD-2006)*, March 2006. ACM Press. Cited on pages 196, 198, 200, and 203.

Satoshi Matsuoka, editor. *Proc. 17th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2002)*, November 2002. ACM Press. Cited on pages 200 and 202.

Mira Mezini, editor. *Proc. 7th Int' Conf. on Aspect-Oriented Software Development (AOSD-2008)*, March 2008. ACM Press. Cited on pages 199, 204, and 205.

Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In Matsuoka (2002), pages 52–67. Cited on page 59.

Russell Miles. *AspectJ Cookbook*. O'Reilly, December 2004. Cited on pages 19 and 131.

Robin Milner. A proposal for standard ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 184–197, New York, USA, 1984. ACM Press. Cited on page 79.

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988. Cited on page 91.

David A. Moon. Object-oriented programming with flavors. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–8, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. Cited on page 16.

Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., October 2006. Cited on pages 5, 98, 106, and 127.

Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed AOP. In Lieberherr (2004), pages 7–15. Cited on page 41.

Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *12th International Conference on Compiler Construction*, pages 138–152. Springer, 2003. Cited on page 145.

Object Management Group. *Unified Modeling Language (UML) 2.0 Superstructure Specification*, August 2005. URL `http://www.omg.org/docs/formal/05-07-04.pdf`. Cited on page 109.

Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In Yonezawa and Matsuoka (2001), pages 73–80. Cited on page 13.

Klaus Ostermann and Mira Mezini. Conquering aspects with Caesar. In Akşit (2003), pages 90–99. Cited on pages 59 and 131.

Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Black (2005), pages 214–240. Cited on pages 40 and 41.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. Cited on page 9.

Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In Yonezawa and Matsuoka (2001), pages 1–24. Cited on page 21.

Renaud Pawlak, Rod Johnson, Andrei Popovici, et al. AOP Alliance (Java/J2EE AOP standard) version 1.0. Home page at `http://aopalliance.sourceforge.net/`, March 2004. Cited on page 121.

Renaud Pawlak, Jean-Philippe Retaillé, and Lionel Seinturier. *Foundations of AOP for J2EE Development*. APress, 2005. Cited on page 21.

David J. Pearce and James Noble. Relationship aspects. In Masuhara and Rashid (2006), pages 75–86. Cited on page 1.

Benjamin Pierce, editor. *Proc. 10th Int' Conf. on Functional Programming (ICFP-2005)*, September 2005. ACM Press. Cited on pages 198 and 205.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, February 2002. Cited on pages 80, 82, 83, 90, 97, 114, and 177.

Hridesh Rajan. Design pattern implementations in Eos. In *PLoP '07, Conference on Pattern Languages of Programs*, September 2007. Cited on page 131.

Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *Proc. 9th European Conf. on Software Engineering and 11th ACM SIGSOFT Symp. on Foundations of Software Engineering (ESEC/FSE-2003)*, pages 297–306. ACM Press, 2003. Cited on pages 64 and 65.

Hridesh Rajan and Kevin J. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *Proc. 27th Int' Conf. on Software Engineering (ICSE-2005)*, pages 59–68. ACM Press, 2005. Cited on page 64.

Tobias Rho and Günter Kniesel. Uniform genericity for aspect languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn, 2004. Cited on page 40.

Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In Taylor and Dwyer (2004), pages 147–158. Cited on pages 5 and 130.

Kouhei Sakurai and Hidehiko Masuhara. Test-based pointcuts for robust and fine-grained join point specification. In Mezini (2008), pages 96–107. Cited on page 41.

Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *Proceedings of OOPSLA 2008*, pages 525–542. ACM, 2008. Cited on page 179.

Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974. Cited on page 13.

Kevin Sullivan, Lin Gu, and Yuanfang Cai. Non-modularity in aspect-oriented languages: Integration as a crosscutting concern for AspectJ. In Kiczales (2002), pages 19–27. Cited on page 64.

Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: An aspect-oriented approach tailored for component based software development. In Akşit (2003), pages 21–29. Cited on pages 3, 20, 43, and 46.

Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, Reading, Massachusetts, USA, 1st edition, 1998. Cited on pages 1 and 20.

Éric Tanter. An extensible kernel language for AOP: Reflex meets MetaBorg. In Brichau et al. (2006). Cited on page 70.

Éric Tanter. Expressive scoping of dynamically-deployed aspects. In Mezini (2008), pages 168–179. Cited on page 41.

Éric Tanter. *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, chapter Aspects of Composition in the Reflex AOP Kernel, pages 98–113. Springer Berlin / Heidelberg, August 2006b. Cited on page 69.

Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In Robert Glück and Michael R. Lowry, editors, *Proc. 4th Int' Conf. on Generative Programming and Component Engineering (GPCE-2005)*, volume 3676 of *LNCS*, pages 173–188. Springer Verlag, September 2005. Cited on pages 45, 46, and 68.

Peri Tarr, editor. *Proc. 4th Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, March 2005. ACM Press. Cited on pages 195 and 206.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. 21st Int' Conf. on Software Engineering (ICSE-1999)*, pages 107–119. IEEE Computer Society Press, May 1999. Cited on pages 10 and 15.

Peri L. Tarr and Harold Ossher. Hyper/J: Multi-dimensional separation of concerns for Java. In Mary Jean Harrold and Wilhelm Schäfer, editors, *Proc. 23th Int' Conf. on Software Engineering (ICSE-2001)*, pages 729–730. IEEE Computer Society, May 2001. Cited on page 15.

Hideaki Tatsuzawa, Hidehiko Masuhara, and Akinori Yonezawa. Aspectual Caml: An aspect-oriented functional language. In Pierce (2005), pages 320–330. Cited on pages 145 and 148.

Richard N. Taylor and Matthew B. Dwyer, editors. *Proc. 12th ACM SIGSOFT Int' Symp. on Foundations of Software Engineering (FSE-2004)*, November 2004. ACM Press. Cited on pages 204 and 206.

Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In Rachid Guerraoui, editor, *ECOOP'99 - Proceedings of 13th European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 186–204. Springer, 1999. Cited on page 102.

Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal M. Gafter. Adding wildcards to the Java programming language. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *Proc. of the 2004 ACM Symposium on Applied Computing (SAC)*, pages 1289–1296. ACM Press, 2004. Cited on pages 102 and 104.

Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)*, January 2005. Cited on page 104.

Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proc. of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135, November 1999. Cited on page 145.

Wim Vanderperren. *Combining Aspect-Oriented and Component-Based Software Engineering*. PhD thesis, Vrije Universiteit Brussel, May 2004. Cited on pages 1 and 64.

Wim Vanderperren, Davy Suvée, María Agustina Cibrán, and Bruno De Fraine. Stateful aspects in JAsCo. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Software Composition*, volume 3628 of *LNCS*, pages 167–181. Springer Berlin, 2005a. Cited on pages 3 and 40.

Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive programming in JAsCo. In Tarr (2005), pages 75–86. Cited on page 14.

Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In Taylor and Dwyer (2004), pages 159–169. Cited on page 40.

Mitchel Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *Trans. on Programming Languages and Systems (TOPLAS)*, 26(5):890–910, September 2004. Cited on pages 144 and 174.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991. Cited on pages 77 and 79.

Akinori Yonezawa and Satoshi Matsuoka, editors. *Proc. 3rd Int' Conf. on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *LNCS*, September 2001. Springer Verlag. Cited on pages 199 and 203.

Index of Terms