PEDECIBA Informática

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

The Reflex Sandbox: An Experimentation Environment for an Aspect-Oriented Kernel

Leonardo Rodríguez Viacava

Trabajo de tesis para la obtención del grado de Magíster en Informática de la Universidad de la República en el programa de Maestría del área Informática del Pedeciba

Supervisor:	Dr. Gustavo Betarte Instituto de Computación, Universidad de la República.
Orientadores:	Dr. José Miquel Piquer – Dr. Éric Tanter Departamento de Ciencias de la Computación, Universidad de Chile.
Presentación:	13 de marzo de 2006

The Reflex Sandbox: An Experimentation Environment for an Aspect-Oriented Kernel
Leonardo Rodríguez Viacava

ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT06-04
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, Marzo de 2006

Resumen

Reflex es un núcleo versátil para la programación orientada aspectos en Java. Provee de las abstracciones básicas, estructurales y de comportamiento, que permiten implementar una variedad de técnicas orientadas a aspectos. Esta tesis estudia dos tópicos fundamentales. En primer lugar, el desarrollo formal, utilizando el lenguaje Haskell, de las construcciones fundamentales del modelo Reflex para reflexión parcial de comportamiento. Este desarrollo abarca el diseño de un lenguaje, llamado Kernel, el cual es una extensión reflexiva de un lenguaje orientado a objetos simple. La semántica operacional del lenguaje Kernel es presentada mediante una máquina de ejecución abstracta. El otro tópico fundamental que estudia esta tesis es validar que el modelo de reflexión parcial de comportamiento es suficientemente expresivo para proveer de semántica a un subconjunto del lenguaje AspectJ. Con éste fin, se desarrolló el Reflex Sandbox: un ambiente de experimentación en Haskell para el modelo Reflex. Tanto el desarrollo formal del modelo de reflexión parcial de comportamiento como la validación del soporte de AspectJ, son estudiados en el contexto del Reflex Sandbox. La validación abarca la definición de un lenguaje orientado a aspectos que caracteriza el enfoque de AspectJ a la programación orientada a aspectos, así como la definición de su máquina de ejecución abstracta. También se presenta un compilador que transforma programas escritos en este lenguaje al lenguaje Kernel. Este proceso de compilación provee los fundamentos para entender cómo dicha transformación puede ser realizada. El proceso de compilación también fue implementado en Java, pero transformando programas AspectJ a programas Reflex. También se presentan mediciones preliminares del desempeño de un programa compilado y ejecutado en Reflex y un programa compilado, y ejecutado con el compilador AspectJ.

Palabras Clave: Reflexión, Reflexión Parcial, Programación Orientada a Aspectos, Reflex, AspectJ

Abstract

Reflex is a versatile kernel for Aspect-Oriented Programming (AOP) in Java. It provides the core abstractions based on both structural and behavioral reflection to implement a variety of aspect-oriented techniques. This thesis focuses on two main subjects. In the first place, a formal definition, using the language Haskell, of the core constructions of the Reflex model for Partial Behavioral Reflection (PBR) has been developed. This development embodies the design of a language, called Kernel, which is a reflective extension of a simple object-oriented language embedding the core constructions of the PBR model. The operational semantics of the language is given in terms of an abstract execution machine. The other major subject addressed in this work has been the validation that the PBR model has the required expressiveness to provide semantics to a subset of the AspectJ language. To this end, the Reflex Sandbox, an experimentation environment in Haskell for the Reflex model, has been developed. Both the formal development of the PBR model and the validation of the support for AspectJ are studied within the Reflex Sandbox. It embodies the definition of an aspect-oriented language, which characterizes the AspectJ approach to AOP, and the corresponding execution machine. A compiler, transforming programs from this aspect-oriented language to the Kernel language, has also been developed. This compiler provides the foundations for understanding how the transformation is possible. A Java implementation of the compiler, transforming AspectJ programs into Reflex programs, has also been developed. Preliminary benchmarks are also presented for the Java implementation of the compiler.

Keywords: Reflection, Partial Reflection, Aspect Oriented Programming, Reflex, AspectJ

Acknowledgements

When I start this thesis, I was not sure what it means and imply the process of its conception and elaboration. Now that the end has been reached, I have some answers, but still have plenty of questions. For me, this thesis implies hard work and study, academic development and perhaps, the most important thing, the self-evolution as a person. Having reached the end, I feel that I need to say "thank you" to many people that help me during the process. I would like to express my gratitude to my study supervisor Gustavo Betarte for its continuous support and encouragement. I am also thankful with him for the countless fruitful discussion that we had over topics that are not his main matter of interest. I am extremely thankful to Éric Tanter. The chances/causalities of life put us together in the same place at the same time. Our first meetings have inspired the present work. He has made great value comments during the development of the work. And the most important thing, I have gained a new valuable friend. Close related with these causalities was José Miguel Piquer, and Raul Ruggia. José supports me unconditionally during my stay in the Department of Computer Science in the University of Chile. Raúl was a key piece in this puzzle by making the arrangements to meet José, to whom I am also grateful. I am also grateful to Jacques Noyé, which along with Éric were the co-authors of the paper that motivates the present work. I would like to thanks Pablo Cancela, Tali Kimelman, Fernando Rodríguez and Martí Varela for the exhaustive revision that they make over this document and the valuable comments that help to improve my English writing skills. To Pablo, I am also thankful for support me on the difficult times that I live. Alberto Pardo and Alejandro Guti'errez, as well as many other people at the Instituto de Computación, specially my office mates Andrés Vignaga, Daniel Perovich, Marcos Viera, Daniel Calegari, Diego Rivero, Dieter Spangenberg, Jorge Corral and Álvaro Rodríguez, directly or indirectly contributed to this work. I am thankful to Nicolás Kicillof for the numerous valuable comments that he does over this document. Thanks to Laura Bermúdez for its efficiency and help. Also I want to give an special thank to Ana María Pino for offering her home and friendship to me, without knowing me at all, during my internship in Chile. I want to reiterate my gratefulness to Laura and to be thankful to Joseline Cortazzo for contact me with Ana María. Last, but not least, I thanks my family and friends for their love and support.

My particular thanks go to Andrea for her unconditional love, support and exceptional encouragement. She gives me the strength that makes this work to come to a happy end. I love you so much.

Contents

List of Figures

Listings

Chapter 1

Introduction

Over the history, programming language research has been motivated by the construction of languages which allow the programmer to clearly capture the concerns involved in the development of a software system. By concern, here we mean, a functionality that the system must perform and, as a consequence of a design decision, it is treated independently. By clearly capturing we mean, that each concerns should be encapsulated in the appropriate language abstraction, in order to be well-modularized, well-localized and composed as necessary. Achieving those properties increase the reusability of the concerns' implementations and enhance their traceability between the different development phases. The advent of object-oriented languages (OOL) represents a big step in the construction of languages for better concern modularization. They introduce the notion of *object* as an abstraction encapsulating data and behavior. An object communicates with other objects using messages through well-defined interfaces. The object proves to be an attractive abstraction to encapsulate the concerns resulting from the decomposition of the system. Still, the programming language community has detected several limitations of the OOL approach [WY88, HO93, AWB+93, OI94, LSLX94, KLM+97], in particular regarding the development of software in the presence of concerns that tend to crosscut the system basic functionality, the so-called *crosscutting concerns* [KLM+97]. Those concerns are typically non-functional, like among others: security, auditing, distribution and communications. As a consequence of these limitations, the implementation of those concerns results in *tangled* code: this means that the modules of the system get "polluted" with code dedicated to address such concerns. Consequently, neither the "polluted" concerns nor the crosscutting concerns can be clearly modularized. This limitations are not only related to OOLs, but also to procedural and functional languages [KLM+97]. Since this work is concerned with OOLs, the rest of the introduction confines attention to this latter kind of languages.

Aspect-Oriented Programming (AOP) [KLM+97] is a promising technology to modularizing software in the presence of crosscutting concerns. It defines the notion of *aspect* as an abstraction to encapsulate crosscutting concerns. Since its introduction in [KLM+97], several AOP approaches have come out. An AOP approach essentially consists of: a *base language*, one or more *aspect languages* and an *aspect weaver* capable of combining programs written in those languages[1]. The programmer using the abstractions of the base language (e.g. objects in an OOL) implements the basic system functionality, while the aspect language is used to implement the crosscutting concerns. Along with the aspect definition, the programmer must specify how the aspects must be composed with the base level program. Such a specification is used by the weaver to automatically perform the composition of the crosscutting concerns with the rest of the system. Among the great variety of AOP approaches, AspectJ [KHH+01] is a reference: it is a simple, well-designed and production quality extension of the Java programming language. An aspect in AspectJ can be defined using two kinds of mechanisms: *dynamic crosscutting* makes it possible to define additional *behavior* to run at well-defined points in the execution of a program; *static crosscutting* makes it possible to modify the static *structure* of a program (e.g adding new methods, implementing new interfaces, etc).

Multiple aspect-oriented (AO) proposals have emerged and there are more to come. Each of those proposals represents a restricted area in the design space of AOP, and each of them differentiates from the others in terms of the conceptual model they offer, the specification language, its genericity, binding time, expressiveness, etc. For instance, there are different conceptual model proposals: AspectJ [KHH+01] relies on the concepts of join point, pointcuts and advices; Event-Based AOP [DMS01] uses the concepts of crosscuts, monitors, events and aspects; Composition Filters [BA01] rather talks in terms of composable method filtering. Also, some approaches adopt general-purpose aspect languages (GPALs) [KHH+01, BAT01, OT01a], while others rely on domain-specific aspect languages (DSALs) [LK97, MKL97, NCT04]. DSALs provide high-level constructions specialized for a particular domain or aspect (e.g. security). They offer several advantages like: declarative representations, simple analysis and reasoning, domain-level error checking, among others. The price for those advantages is the loss of generality. On the other hand, GPALs provide low-level constructions for aspect construction, allowing more general aspect definitions.

In [TN04a] the authors motivate the need for an *AOP kernel*. The motivation is based on two main observations. The *first* one is that most AO proposals share several commonalities (e.g. weaver base code transformation) that can be factored out by an

[1]Actually, there is a family of AOP approaches, the so-called *symmetrical* ones, which do not distinguish between base and aspect languages. However, in this work we are mainly concerned about *asymmetrical* approaches, which do make the distinction.

AOP kernel. The *second* observation relates to the idea that the most adequate conceptual model and level of genericity for a given application domain actually depends on the particular situation: there is no definitive, omnipotent AO proposal that best suits all needs. Therefore, when several aspects are to be handled in the same piece of software, combining several AO proposals may be the best choice. However, the tools available for each AO proposals are not meant to be compatible with each other, which makes it difficult to combine them in a single piece of software. The tools of the different AO proposals are implemented with a close-world assumption, i.e. each tool affects the base code directly. Consequently, combining different tools without appropriate coordination may have unexpected results. This problem further complicates in the scenarios that aspects defined in two or more different tools affect the same program point, since the interaction is resolved blindly. An AOP kernel enables a wide range of approaches to work together without breaking each other. Through the appropriate structural and behavioral models, such a kernel must provide a core semantics, generic enough to support the different proposals.

From its beginnings, AOP has a deep connection with work in *computational reflection* [Smi82, Smi84, Mae87b], as mentioned in [KLM$^+$97]. A reflective system is aware of its own structure and behavior, and can introspect and alter itself. The computations in a reflective system conceptually occur at the base or meta level. Typically, the former corresponds to the external domain that the system models, while the meta computation controls how the base level computation is performed. The concept of reflection has been extensively studied in the context of programming languages [dRS84, KRB91, Riv96, DS01]. A reflective language offers to the user a back door to the language implementation, providing mechanisms to introspect and/or alter the language inner structures and its semantics. Such a back door provides a view of the program execution that no base level program could ever see, such as the entire execution stack or all calls to objects of a given class. Therefore, the user programs running at the metalevel have crosscutting views of the base level program, which makes them suitable for crosscutting concern implementation. As pointed out in [Kic01], AOP offers a principled way to do reflection, hiding from the user most of the complexity associated to the implementation of a reflective system by providing a high-level language to express the semantics alterations over the base language. In [KLM$^+$97] they said *"AOP is a goal, for which reflection is one powerful tool"*, suggesting that reflection can be seen as a tool to implement AOP rather than an AOP approach. Reflection is too complex for a regular programmer to work with and often, it is too inefficient, specially in the case that behavioral reflection is the goal.

Reflex [TNCC03] is a reflective extension for Java that further extends the primitive reflective mechanisms of that language by providing support for behavioral reflection and a more complete support for structural reflection. Behavioral reflection is provided through a comprehensive model of partial reflection, which allows to specify *where* and *when* reflection must occur. This model is meant to reduce the costs associated to reflection by using it only when it is needed. As presented in [TNCC03, CMT04] Reflex proves to be an effective and efficient approach to implement crosscutting concerns. In [TN04a] the authors put forward partial behavioral reflection (PBR) as an appropriate low-level framework for the construction of an AOP kernel. They have also claimed Reflex as an AOP kernel for the Java language, taking care of the behavior, structure and composition of aspects.

In order to validate such a claim, there is still much work to be done. First, a formal understanding of the low-level constructions of the model is required, since the only available description for the model is the informal semantics presented in [TNCC03, Tan04] and its implementation in Java [RFX]. Second, several consequent case studies should be carried out to validate the aptitude of the Reflex model as an AOP kernel. In [CMT04, TN04b] the authors already studied the support of lightweight domain-specific aspect language for concurrent programming. However, a serious validation requires studying the support for a general-purpose, expressive, and widely-accepted aspect language [RTN04], like AspectJ.

There are two main problems addressed by this thesis:

- The most fundamental portion of Reflex, the core constructions of the PBR model, has been given semantics through the design of a language, which embeds the constructions in the model, and the corresponding abstract execution machine. The language is a reflective extension of a simple OOL called BASE. The machine is used to give operational semantics to the language. Both the language and the corresponding machine have been formally written down in Haskell. This has made it possible to express the semantics of the model in a more abstract and simple way that it is achieved by the informal semantics described in [TNCC03, Tan04] and its implementation in Java [RFX]

- Taking the first and most fundamental step in the validation of the support for the AspectJ language: providing semantics for a subset of its dynamic crosscutting mechanism. Both theoretical and practical validations are performed. The theoretical validation embodies the definition of an extension to the BASE language incorporating the considered AspectJ constructions and the development of a compiler in Haskell, showing how the constructions of the PBR model are used to express the semantics alterations defined (over the base level program) by the AspectJ constructions. The practical validation embodies the implementation of a compiler in Java, transforming an AspectJ program into a Reflex program, along with benchmarks showing that very reasonable levels of efficiency can be achieved. Those benchmarks compare the performance of programs compiled using the AspectJ Compiler [aspb] and programs compiled into the AOP kernel.

In addition, the *Reflex Sandbox* (RSB), an experimentation environment in Haskell for the Reflex model, has been developed. The RSB provides a concise model of Reflex's AOP kernel for theoretical studies. It is a tool that can be used to implement alternative semantics for the low-level constructions of the model. The RSB is inspired in the Aspect Sanbox (ASB) [asb, MKD02, MKD03a, MKD03b, WKD04], sharing the same high-level goals, but the ASB was oriented to the study of different models of AOP.

Three languages and the corresponding execution machines are included in the definition of the RSB. In addition to BASE, the base level language, the RSB includes the *Kernel language* and *Pointcut and Advice* (PA) *language*. The Kernel language is a reflective language that extends BASE by incorporating the constructions of the PBR model. PA is an AO language, also extending BASE, containing the constructions of AspectJ's dynamic crosscutting mechanism. The AO constructions included in the PA language are inspired in those used by the ASB to describe the AspectJ approach to AOP [MKD03b, WKD04]. In addition, the RSB also embodies the compiler that transforms a PA program into one expressed in the Kernel language. This process leaves the BASE portion of the program intact and compiles its AO portion making use of the reflective constructions of the Kernel language.

Since the Kernel machine has been designed to only support part of the Reflex model constructions, not every PA program can be compiled into a Kernel program. This design decision also implies that some interesting features of the AspectJ language (e.g. aspect composition) were left out of the PA language. On the other hand, the Java implementation of the compiler, transforming AspectJ programs into Reflex programs, does consider all the constructions of the Reflex model. Therefore, it supports the compilation of all the AO constructions defined in the PA language and some additional features of the AspectJ language. It was implemented as a plugin for Reflex's plugin architecture. The plugin also implements a reflective API for the AspectJ language, providing introspection facilities, which are similar to those provided by the Java reflection API. The API of the plugin can be used to introspect the aspects (along with its members) defined in the AOP kernel and to access the low-level kernel constructions used to represent them. The main benefit of the API is that it provides access to AOP kernel aspects at the appropriate level of abstraction, which can be used, for instance to define composition rules among aspects of heterogeneous languages.

The RSB provides a testing environment where tests can be done on a per-machine basis or on an inter-machine basis. The former allows to intuitively check that a machine behaves as expected. The latter allows to perform comparative tests between two machines. For instance, allowing to get an initial intuition of the correctness a compiler between two machines. The Kernel and PA machines have been fully implemented using the Haskell language and tested using the RSB testing environment.

The compiler from the PA machine to the Kernel machines also has been tested using the environment.

1.1 Contributions of this Work

The results presented in this work constitute a contribution to the evolution of Reflex as an AOP kernel. In the first place, the RSB environment, whose conception, design and implementation are precisely described in this document, provides a formal setting in which the semantics of the Reflex model can be defined. In particular, this makes it possible to develop theoretical case studies where experiments concerning how the kernel may support different AOP approaches can be carried out. As a concrete experiment, in this work attention has been confined to the study of the dynamic crosscutting mechanism of the AspectJ language. The comparison between the Kernel and PA machines reveals the abstraction gap that exists among them, meanwhile the RSB compiler shows how the lower-level constructions of the Kernel can be used to provide semantics for the higher-level constructions of the PA language. Finally, it is illustrated that the transformation is feasible in an industrial environment, Java, and that it can be done efficiently.

In summary, this work makes the following contributions:

- The formal development in Haskell of the core constructions of Reflex's PBR model, i.e. the Kernel machine. The Kernel language provides a dedicated syntax for the constructions in the PBR model, avoiding thus the unnecessary noise resultant of being limited to use object-oriented syntax to express those constructions. This is in contrast to the API-based approach followed by the Reflex original implementation in Java. The mentioned syntactical enhancement and the concise semantics described by the Kernel machine greatly simplify the understanding of the PBR model. Furthermore, a comprehensive conceptual description of the model, in the context of reflective languages, is also presented.
- The validation of Reflex's PBR model being sufficiently expressive to efficiently represent the semantics of the core constructions of AspectJ's dynamic crosscutting mechanism. Such a validation, represents the first important case study that has been carried out in order to fundament the claim that PBR is an appropriate framework for the construction of an AOP Kernel and that Reflex is a promising candidate for an AOP Kernel for Java.
- The experiment described in the previous item deepens on the understanding of the relation between reflection and AOP. The idea that reflection can support AOP is not new (for instance see [KLLH03, Sul01]) and has been a matter of

(often mundane) discussion since the beginning of AOP. We have gone one step further by actually showing that an essential part of an efficient AOP language, AspectJ, originally implemented without resorting to reflection, can actually be supported by reflection in an efficient way.

- The construction of the RSB experimentation environment. It constitutes a very useful tool for rapidly prototyping new extensions to the Reflex model. It also provides a setting for experimenting with different AOP approaches, avoiding all the complex issues present in production quality environments like Java. The new extensions to the Reflex model can be developed by extending the Kernel or BASE machines. In addition, the BASE language and machine can also be extended to incorporate concise models for AOP approaches being studied.
- Improvements to the Reflex model. The feedback obtained from the AspectJ case study suggests enhancements that can be done to the PBR model. Some of them have already been incorporated into the model, e.g. the ones that we present in [RTN04], while others remain to be further analyzed.

1.2 Structure of the Thesis

This thesis is structured as follows. Chapter 2 includes background information on computational reflection, aspect-oriented programming, AspectJ and the Reflex model for partial behavioral reflection. It also further motivates this work. Chapter 3 is dedicated to the description of the Kernel machine. First, the BASE language and its corresponding execution machine are presented. Secondly, the features of the Reflex model to be included in the Kernel machine are described, along with the design of the Kernel language. Finally, the semantics of the language are exposed through the presentation of the Kernel machine. In addition to the description of the model semantics, a comprehensive conceptual description of the model in the context of reflective languages is also presented. Chapter 4 introduces the PA language, including the most important features of AspectJ's dynamic crosscutting mechanism. In addition, the development of the PA machine is described. Chapter 5 is dedicated to the presentation of how the Reflex model can be used to provide semantics for the high-level constructions of AspectJ. First, this is analyzed within the RSB by presenting a compiler from a PA program to a Kernel program, which provides the foundations for the required translation. Secondly, based on the feedback obtained from the RSB, the compiler is reviewed in the context of Java. Finally, the implementation of a plugin for Reflex, including the compiler and the required infrastructure to execute AspectJ programs in Reflex, is presented. Preliminary benchmarks of the plugin are also presented. Chapter 6 concludes this thesis with a short summary, and presents the related work in the area and an outline of future work.

Chapter 2

Background and Motivation

This chapter presents the background information on two themes surrounding this work, reflection and aspect-oriented programming (AOP). Both of them are research areas that have addressed the design of techniques for crosscutting concern modularization. Herein, the need for an AOP kernel is further motivated, and an explanation of why reflection is a powerful tool for AOP kernel implementation is also provided. Reflex, a reflective extension for the Java language, is presented as well as an initiative to evolve into an AOP kernel. This initiative also motivates the need for a formal understanding of the Reflex model. The AspectJ language, a reference among the great variety of AOP proposals, is presented. Finally, the need for the validation of the aptitudes of Reflex for providing semantics to AspectJ is further motivated.

This chapter is organized as follows. Section 2.1 presents the problem that AOP, and consequently an AOP Kernel, intend to solve: modularization of crosscutting concerns. This problem is motivated in the context of Separation of Concerns, a fundamental principle of software engineering. Section 2.2 introduces the main concepts behind AOP. Also, it briefly describes the great variety of existing AOP proposals and languages. The concept of reflection and its applicability in the context of programming languages is described in Section 2.3. Section 2.4 presents a comprehensive description of Reflex and its model for partial behavioral reflection. Section 2.5 further motivates the need for an AOP kernel and argue why reflection, and in particular partial reflection, offers an adequate model for the construction of such a kernel. In addition, it motivates the present work regarding the construction of the Reflex Sandbox. Section 2.6 presents the AspectJ language, in particular focusing on those constructions of its dynamic crosscutting mechanism. Finally, Section 2.7 makes it clear the importance of validating the Reflex AOP kernel by providing semantics to a subset of the AspectJ dynamic crosscutting mechanism.

2.1 Separation of Concerns

Separation of Concerns (SoC) [Dij68, Par72] is a commonly known and fundamental principle of software engineering. The motto of SoC [HL95] is that concerns must be properly identified and isolated in order to manage system complexity, achieve a system that is easy to understand, allow better reuse of the concerns and enhance system maintenance (among other desirable properties). Separating concerns results in a higher level of abstraction, allowing the developer to reason about individual concerns in isolation. At the same time, concerns are easy to understand since their code is not cluttered with the code of other concerns. The ideal is that this separation should be made in all the phases of the system development (i.e. analysis, design, implementation, etc), in order to achieve the desired properties in each phase and obtain traceability among them.

In particular, during the implementation of the system, the identified concerns are programmed using the constructs offered by the programming language. Therefore, the programming language fulfills a crucial role in achieving good SoC at the implementation level. As a matter of fact, one of the major quests of programming language research is that of being able to develop software systems while preserving a good SoC. Object-Oriented Languages (OOLs) were themselves a step forward in this direction. However, as software systems are applied to more and more complex situations and demanding environments, it becomes difficult to maintain a good separation of all concerns. As it has been pointed out in [HL95], there are many kinds of concerns which OOLs fail to correctly separate (modularize). This kind of concerns are called *crosscutting concerns* [KLM$^+$97]. They are concerns that *crosscut* the system's basic functionality, which are typically associated to non-functional requirements, e.g. synchronization, auditing, distribution, among others.

In order to illustrate those concerns, consider the Java code shown in Listing 2.1[1], which is part of an airline reservation system. The system must allow the user to reserve seats of a given flight and pay for them. In addition, the system must audit all the actions performed by the user.

Listing 2.1 outlines one possible design for this system, which consists of two use-case controllers, one for making a reservation and one for paying the ticket, and an `Audit` class that encapsulates the logic for recording the user actions. Since the audit must be performed for each system operation, each operation must invoke the `Audit` class in order to record its own execution. Note that the audit concern crosscuts the basic system functionality (i.e. reservation and payment) and is not appropriately modularized, because the controllers must *explicitly* invoke the `audit` method. A

[1] As a general convention of the present work, the programming language ($\mathcal{L}$) used to implement the code of a listing shall be specified by an annotation of the form $< \mathcal{L}$ *Code* $>$ in the listing's caption.

```
class ReservationController {
  int reserve(int flight, int seat,..){
     Audit.get().audit(user,tstamp,..);
     [..]
  }
  void cancel(){
     Audit.get().audit(user,tstamp,..);
     [..]
  }
}
class PaymentController {
  void payTicket(int resNumber, ...){
    Audit.get().audit(user,tstamp,..);
    [..]
  }
}
```

```
class Audit {
  Audit itsInstance;
  static Audit get(){..}
  void audit(...){..}
}
```

Listing 2.1: Audit Example <Java Code>

different design may be adopted, in order to enhance the modularity of the solution, for instance, using the Observer design pattern [GHJV95]. However, using design patterns requires adding several artificial classes, which increase the complexity of the design, and usually does not solve the problem completely.

This difficulty for appropriately modularizing crosscutting concerns comes from the fact that OOLs lack an appropriate construct to modularize them [HL95, KLM$^+$97]. This issue along with the design of techniques more suitable for modeling crosscutting concerns is studied in the next section.

2.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [KLM$^+$97] has been proposed as a programming technique for improving separation of concerns in software (see Section 2.1), in particular for modularizing crosscutting concerns. As explained in [Lop02], AOP was conceived based on an extensive amount of prior work, consisting of various techniques trying to solve the same problem, achieving crosscutting modularity. Among those techniques were: Composition Filters [AWB$^+$93], Adaptive Programming [LSLX94], Subject-Oriented Programming [HO93] and Metalevel Programming [OI94, WY88]. The great contribution of the paper [KLM$^+$97], which formally presents AOP, was clarifying the various concepts involving crosscutting modularity and proposing an homogeneous terminology for describing them, which otherwise were treated independently by each different technique.

2.2.1 Motivation and Concepts

As presented in [KLM$^+$97], the motivation behind AOP relies on the observation that, in many existing programming languages, the design decisions involving crosscutting concerns are hard to clearly capture using the existing entities to modularize code. They made the analysis based on a family of languages, which they call *generalized-procedure* (GP) languages, which are characterized by being languages whose dominant abstraction and composition mechanisms are based in some form of generalized procedure. This family of languages includes object-oriented languages, functional languages and procedural languages. For instance, in an object-oriented language[2] the classes and methods are the dominant abstraction (i.e. the generalized procedures) while the method invocation is the dominant composition mechanism (i.e. the generalized procedure call). It is important to note that AOP does not disregard the many important advances that those languages have made in terms of abstraction, genericity, encapsulation, etc; on the contrary, AOP is built over those languages in order to further extend their power.

The design methodologies for the GP languages tend to decompose the system into functional units[3]. Each unit is modeled, at the code level, through its dominant abstraction mechanism (e.g. classes and methods), and all of them are composed using the language composition mechanism (e.g. method call) to form the overall system. The crosscutting concerns are inherently hard to cleanly modularize because of their nature: they, by essence, *crosscut* the basic system functionality. Since those concerns must be coordinated with the system's functional units, the programmer facing the implementation of the system must compose them manually, leading to *tangled code*: meaning that the functional units of the system get "polluted" with code dedicated to address such concerns[4]. For instance, consider the example shown in Listing 2.1 (Section 2.1), where the audit concern is composed with the rest of the system using method invocation, resulting in tangled code. Tangled code is extremely difficult to maintain, since small changes to the functionalities requires manually untangling and then re-tangling it, besides leading to less understandable and reusable code. As argued in [KLM$^+$97], this crosscutting nature can not be appropriately captured with the abstraction and composition mechanisms provided by GP languages, thus new mechanisms for abstracting and composing crosscutting concerns are required.

In [KLM$^+$97] also, they present two important terms: *component* and *aspect*. Considering a system and its implementation using a GP language, a concern that must

[2]Assuming that the language is class-based.

[3]Commonly known as functional decomposition [Par72].

[4]Note that the tangled code notion is defined from the point of view of the "affected" functional units. In addition, the notion of *scattered code* is also commonly used to refer to the same concept seen from the point of view of the crosscutting concern: meaning that the code of the concerns is spread out rather than well-localized.

be implemented is:

A component when it can be cleanly encapsulated in a generalized procedure. The components tend to be units of the system's functional decomposition.

An aspect when it can not be cleanly encapsulated in a generalized procedure. The aspects rather tend to be concerns that affect the performance or semantics of the components in systemic ways.

They also further precise what it means for a concern to be "cleanly" encapsulated: well-localized, easily accessed and composed as necessary.

The goal of AOP is to assist the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system [KLM$^+$97]. AOP, by itself, does not define any mechanisms for abstracting and composing aspects, those are defined by each particular aspect-oriented (AO) technique. In the next section a characterization of the different AO techniques is presented.

2.2.2 Characterizing Aspect-Oriented Techniques

As presented in [KLM$^+$97], an AO technique consists of: a *component language* to program components, one or more *aspect languages* to program aspects, and an *aspect weaver*, that given a component program and one or more aspect programs combines them. The design of an AO technique implies understanding well what should be the component language, what should be the aspect languages, and what must be shared between them. The component languages are typically GP languages. Aspect languages should preferably be high-level and more tailored to the specific aspect they are tackling, in order to be easier and safer for the programmer to use. This implies that special care must be taken in the design of the aspect languages, in order to anticipate anything that the aspect programs need to control, and at the same time, to restrict the control over what it does not need to know. Section 2.2.3 presents an overview of the different styles of aspect languages. Each aspect language defines the mechanism for aspect abstraction, and in coordination with the component language (and also possibly with the other aspect languages), the mechanism for composition [MKD03b].

The different AO techniques can be classified in *symmetric* and *asymmetric* [HOT02a]. Symmetric techniques are those that do not make distinction between components and aspects, both are treated as a single composable element and are implemented using the same language. Conversely, asymmetric techniques make an explicit distinction between components and aspects, providing specific mechanisms for the component-component composition and component-aspect composition. Optionally, they may also provide specific mechanisms for aspect-aspect composition. For instance, AspectJ [KHH$^+$01, MKD03a], is an asymmetric technique supporting aspect-aspect composition.

In [MKD03b] the authors further refine this conception of the elements that conform an AO technique, in order to include symmetrical techniques. For instance, Hyper/J [OT01a, OT01b] consists of: a single language in which components and aspects are implemented, a weaver responsible for combining several programs implemented in that language and a specification of how the combination must be done. They present a general model, that allows to characterize most of the AO techniques that actually exist. Since in this work we are interested in asymmetrical techniques, we shall not enter into further details about the symmetrical ones. The rest of the presentation of AOP is in the context of asymmetrical techniques.

The aspect weavers process the components and aspects, in order to compose them properly to produce the overall system. Essential to that task is the notion of *join points*, which are those elements of the component language semantics that the aspect program coordinate with [KLM$^+$97]. The aspect weaver works by generating a *join point representation* of the component program and integrating the aspect program in the component program at appropriate places, according to the representation. For instance, considering the airline system (see Section 2.1), the join points would be the executions of the system operations (i.e. the methods in the controllers), and the weaver would perform the integration by placing the invocations to the audit concern. Finally, the weavers can be designed to perform the integration at runtime or at compile time.

2.2.2.1 Join Point Model

As explained in [MKD02], the ability of the different AO techniques to modularize crosscutting concerns lies in its *join point model* (JPM). A JPM is defined based on three elements:

- The *join points*, which are the points of reference that aspect programs can affect;
- A *means of identifying* join points;
- A *means of associating a behavior* to the join points.

Join points may be *dynamic join points* if they refer to runtime actions, such as method calls, method executions, object instantiation, etc. They may be *static join points* if they denote locations in the code (e.g. expressions). Also, as explained in [KLM$^+$97], the join points are not necessarily explicit constructs in the component language, they may refer to implicit elements, e.g. elements that can be inferred from a static (resp. dynamic) analysis of the program code (resp. execution) revealing some particular structure or behavior of it. For instance, they may denote data flows in the component program [KLM$^+$97], or elements in the class graph [LSLX94]. But as a matter of fact, most of the AOP techniques are based on a join point model similar to that of AspectJ [KHH$^+$01], characterized as the *dynamic crosscutting mechanism*, where join points are well-defined points in the execution of a program.

The Aspect Sandbox (ASB) [asb] was a project meant to provide an experimentation environment where different AOP techniques can be studied and compared, along with prototyping alternative AOP semantics and implementation techniques. In this context, different join point models have been studied and characterized through the implementation of a interpreter in Scheme, describing its approach to AOP. In [MKD03b] they further refine the notion of join point, as existing in the result of the weaving process rather than being in any of the input programs (component or aspects programs), which allows them to describe a wider range of AO techniques (e.g. symmetrical techniques).

Another dimension of AOP where special attention has been placed is how to identify, or select, join points. In AspectJ [KHH$^+$01], join points are selected through *pointcuts*, which are predicates that declaratively specify a set of join points of interest (see Section 2.6 for an detailed explanation). The introduction of control-flow crosscutting, in which a pointcut depends on control flow relation with respect to another pointcut, is very valuable. It was first introduced in [KHH$^+$01], but in somehow restrictive manner, then [DT04] the authors propose a much more expressive way of reasoning about control flow. In [MK03] the authors have studied the support for dataflow-based pointcuts, which allows to specify where aspects should be applied based on the origins of values. Other approaches to identify join points have been developed, such as in [DD99], where the authors proposed a logic metaprogramming approach to AOP, where logical queries can be used to specify crosscuts, and in [ACK05], where they propose a framework to define semantical pointcuts based on ontologies, which are used to represents different semantical views of a program.

Yet another interesting dimension of AOP is the aspect interaction and composition. Two (or more) aspects are said to interact, when both of them try to affect a single join point occurrence. In those cases, additional semantics must be provided in order to solve the issue. For instance, AspectJ [KHH$^+$01] provides a construct to specify aspect precedence, which allows to specify the order in which aspects must be applied upon an interaction. In [DFS02] the authors have pointed out that such a construct is not sufficient to handle complex interaction between aspects. Composing aspects does not solely refer to specifying the order in which they apply, but to possibly condition their application to the presence and application of other aspects. In [DMS01, DFS02] the authors are concerned by the formal description and analysis of aspect interactions, and they propose several expressive composition operators in order to specify the resolution semantics of the interactions.

2.2.3 General vs. Domain-Specific Languages

There is a wide range of AOLs which have been proposed, some of them are general-purpose aspect languages (GPALs), providing low-level constructions to deal with crosscutting modularity, while others are domain-specific aspect languages (DSALs), providing high-level constructions specialized for a particular domain or aspect. DSALs present various advantages, they offer declarative representation, such as simple analysis and reasoning, domain-level error checking and optimizations. Meanwhile, GPALs provide greater generality for the definition of aspects. As first argued in [KLM$^+$97], aspect languages should preferably be high-level, providing a dedicated set of constructions focusing on an specific aspect domain, in order to be easier and safer for the programmer to use. Actually, the first AOLs used to experiment with AOP were DSALs [MKL97, LK97, ILG$^+$97]. Those experiments have provided elegant and simple languages to effectively modularize specific aspects of a system. However, since their specificity and close-world designs, they were not appropriate to experiment with aspect interaction and composition. Therefore much of the research efforts shifted into GPALs [KHH$^+$01, OT01a, DMS01, DFS02].

The balance of pros and cons between GPALs and DSALs is also valid in the context of programming languages in general, not only aspect languages. Although, in the context of AOLs, the need for languages that provide as much control as possible seems to be particulary evident, because of the power that AOP encloses. As explained in [FF00] AOP allows the programmer to make quantified programmatic assertions over the whole system, consequently a single aspect definition may be affecting the system semantics in many different points. Furthermore, [DGH$^+$04] presents a series of measures of AspectJ showing that the language must be used with care in order to avoid efficiency pitfalls. Therefore, having an appropriate domain-specific control, through DSALs, seems to be the right direction.

2.3 Reflection in Programming Languages

Since its birth in [KLM$^+$97], AOP has been deeply related to reflection. Moreover, reflection has been a mentor in the development of AOP, often used to prototype different AOLs during their conception. In this section we shall introduce the concepts related to reflection (Section 2.3.1) and how they are used in the design of programming languages (Section 2.3.2 in ahead).

2.3.1 Computational Reflection

The concept behind *Computational Reflection* can be grasped by analyzing the meaning of the word *Reflection* in the context of philosophy. Reflection is a human-mind activity: our mind is capable of considering some subject of matter and also has the faculty of considering one's owns ideas or acts as the subject of matters. Therefore, allowing us to reflect about our own ways of thinking or functioning, and possibly changing it. By doing a parallel with this idea, we can say that computational reflection is the activity performed by a computational system when doing computation about its own computation. Those reflective computations may affect the state and computation of the system.

The concept of reflection has been deeply studied in philosophy and it was Brian C. Smith, in his PhD thesis [Smi82], the one that gave the first major step in bringing this concept into the computer science world. He proposed and defined what it means for a system to be reflective. He presented the general architecture of *procedural reflection* and illustrated it through the implementation of a reflective dialect of Lisp, called 3-Lisp.

In order to better understand reflection, we will define the most relevant concepts behind it (see Figure 2.1 for a graphical illustration), based on [Mae87a].

Definition Computational system

A system that acts and reasons about a domain.

The domain is the piece of reality or the abstract problem that the computational system models. From the computational system's point of view, the domain is external to the system. Computational systems are described by *programs*, which are textual descriptions enclosing the definition of the *structures* and *behavior* required to model the external domain. The structures hold data relevant to the domain modeling, while the behavior reasons and acts over them. The computational system is the program being ran by an executor, see Figure 2.1(a). In order to be useful, both sides (i.e. the system and its domain) must be up-to-date with each other, which means that if one side changes, this leads to a corresponding effect in the other. This two-way connection is known as the causal connection.

Definition Causal connection

Property that ensures that changes in the domain are reflected in the computational system, and vice-versa.

Now considering this definitions and the fact that systems can reason and act on other systems, we can define the concept of a *metasystem*, see Figure 2.1(b).

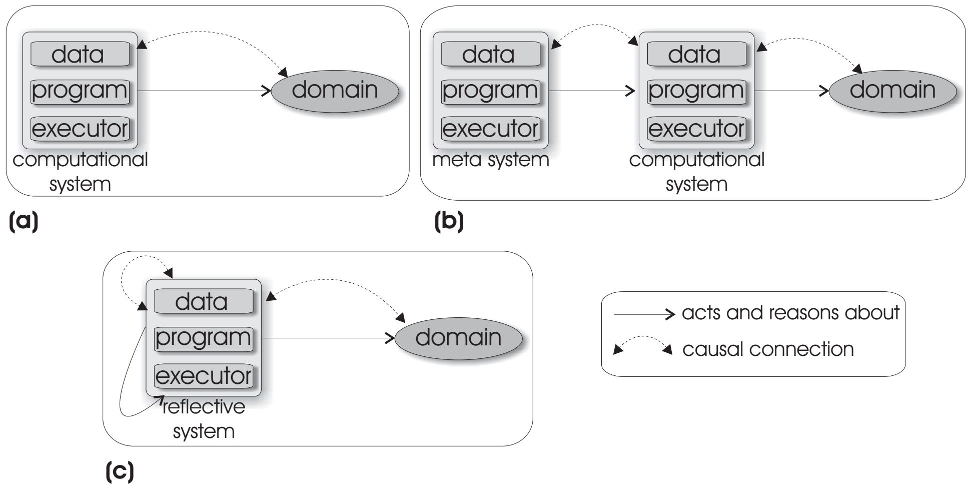


Figure 2.1: Computational Reflection

Definition Metasystem

A computational system whose domain is another computational system.

The domain of a metasystem, a computational system, is called the *base system*. An evaluator is an example of a metasystem that turns a program into a computational system by executing it. The evaluator is causally connected with its base system. The program of the evaluator, or any other metasystem, is a *metaprogram*. Reflective systems appear when considering a metasystem whose domain is itself, see Figure 2.1(c).

Definition Reflective system

A metasystem causally connected to itself.

The idea of a reflective system provides a two-level (actually multi-level) way of thinking. Reflective systems can be conceptually divided in a base-system (or -level) representing the external domain, and a meta-system (or -level), capable of reasoning and acting upon its own computation. The computations occurring at the metalevel provide views of those of the base level that no base program could ever see, such as the entire execution stack, or all calls to objects of a given class. Therefore, they crosscut the base level computations, providing a suitable framework for crosscutting concerns representation.

Now that the concept of reflection has been presented, let's focus on the programming language[5] used to implement the reflective system. The program that describes the reflective system must be capable of manipulating a representation of the system itself, in the same way as it does with its external domain. Therefore, a causally connected representation of the program itself and the executor must be made explicit by the *programming language* for program manipulation. The representation of the executor comprehends the language semantics and the runtime structures used in its implementation. Understanding how this self-representation can be provided, is where most of the research efforts in reflection has been focused, and shall be explained in detail in the next section.

2.3.2 Reflective Programming Languages

A programming language is *"a medium to express computations, which is defined by its syntax and its semantics"* [MJD96]. A illustrative analogy was done in [KRB91][6] between programming language design and theater play, where the producer is the language designer/implementor and the user is the audience. The audience only sees *on-stage* behavior, which is supported by a *backstage* infrastructure, unknown to the audience. The producer is responsible for setting up this infrastructure in order to support the on-stage behavior. Programming language design is similar, language design is an activity where the designer provides a set of high-level constructions to be used by the programmer as a black-box abstraction. During the process, the designer is expected to produce languages with well-defined and fixed semantics. Users are expected to use the language at their convenience, but respecting the designer decisions. Reflective languages come to challenge this tradition, by allowing the user to access and modify some or "all" aspects of the language backstage.

The reflection community still lack of a widely accepted account of what is a reflective language [MJD96, Riv96]. Some of the most relevant characterizations are:

- B. Smith gives two important requirements for a language to be reflective [Smi84]. First, the language needs *"an account of itself embedded within it"*; which means, the language must be able to access its own representation. Second, this self-representation must be causally connected with the language implementation.
- Maes [Mae87a] presents the following requirements for reflective languages: (1) it must recognize reflection as a fundamental programming concept; (2) it must

[5]For simplicity we assume that only one language is used for implementing the system.

[6]Actually the analogy was made between the language implementation and the documented language.

provide support for modular implementation of reflective computation; (3) its implementation must provide an open-ended architecture. Besides, she complements this by saying *"The interpreter of such a language has to give any system that is running access to data representing (aspects of) the system itself [...]"* and *"The interpreter also has to guarantee that the causal connection between these data and the aspects of the system they represent is fulfilled [...]"*[7].

- Malenfant *et al.* say that *"a programming language is said to be reflective when it provides (full) reflection"*. [MJD96].

The first two characterizations differ from the third in the fact that full reflection is not required by a language to be reflective. Full reflection stands for not imposing any limit on what the program can modify or change, therefore ideally any part of the language semantics, syntax, implementation and runtime, should be accessible. However, there are theoretical limits that make it impossible to achieve full reflection as pointed out in [WF88, DS01]. Actually, the precise point where a language that incorporates reflective properties becomes a reflective language is not well defined [Riv96]. In spite of the theoretical limits, the requirement of full reflection is useful to contrast reflective languages from programming languages that may only incorporate some reflective properties. Malenfant [MJD96] explain that the confusion arises because reflective properties already appear in several existing languages. For example, if we contrast standard non-high-order languages with high-order languages, we will note that there are entities that are implicit in the former and that are given the status of first class in the latter. As first class entities, they can be manipulated as standard entities (e.g. passed as parameter, assigned to variables, etc). This is the case of Scheme, where functions and continuations are first class. Clearly, Scheme is not considered a reflective language, the break point came up from the fact that Scheme does not provide a representation (data structures) for those entities that can be inspected [MJD96].

The key concept behind reflective languages seems to be their ability to build such representations for some aspects of itself, and make them available to the user program, which can manipulate (i.e. introspect or change) them as ordinary data. *Reification* is commonly known as the process where those aspects of the language, which were implicit to the user program, are brought to the fore using a representation expressed in the language itself [MJD96]. Those representations are *causally connected* to the related *reified* information such that a modification to one of them affects the other. The concept of reification shall be explained in grater detail in Section 2.3.2.2.

With regard to this thesis, we shall consider as reflective languages those that recognize reflection as a fundamental concept and provide at least one *reflective mechanism*,

[7]Although this characterization is focused in interpreted languages, it is also valid in the context of compiled languages.

which offers to the user program the ability to reify some aspect of the language, for introspection, and optionally, modification. The next section gives a detailed account of what is a reflective mechanism.

2.3.2.1 Reflective Mechanism

Reflective mechanisms is the name given to the facilities provided by a programming language that allow some form of reflective computation. Malenfant defines them as *"any means or tool made available to a program P written in a language L that either reifies the code of P or some aspect of L, or allows P to perform some reflective computation"* [MJD96]. There are two useful distinctions that can be made to better understand reflective mechanisms.

The first comes from a natural distinction between *structural* and *behavioral* reflection [MJD96]:

Definition Structural reflection

The ability of a program to access its own structure, as it is defined in the programming language.

Definition Behavioral reflection

The ability of a program to access its dynamic representation, which means, have access to its operational execution as it is defined by the programming language implementation.

Therefore reflective mechanisms can be structural or behavioral. For example, in an object-oriented langauge, structural mechanisms may give access to the classes in the program as well as their defined members; while behavioral mechanisms may give access to the language method lookup mechanism, as well as the state of the execution stack of the various threads in the program. Behavioral mechanisms may cover any aspect of the semantics of the language, its particular implementation strategy and the runtime infrastructure used for the implementation.

The second distinction is made between *introspection* and *intercession* [BGW93]:

Definition Introspection

The ability of a program to **reason** *about reifications of otherwise implicit aspects of itself or of the programming language implementation.*

Definition Intercession

The ability of a program to actually **act** *upon reifications of otherwise implicit aspects of itself or of the programming language implementation.*

Java language mainly provides a limited introspection mechanism through the reflection API by allowing to introspect class hierarchy and access class members, but it does not provide any mechanism to modify this data. Smalltalk [Riv96] provides a huge set of introspection mechanisms and also provides intercession mechanisms allowing modifications to the class hierarchy and memory management policies, among others.

Note that these two distinctions are orthogonal, the former determines which is the domain of the representation (the program or the processor) and the latter determines the type of access given to that representation.

2.3.2.2 Reflective tower

Behavioral reflection is by far more difficult to achieve than structural reflection. Smith [Smi82] was a pioneer in the field of behavioral reflection and was the one that proposed the reflective towers as a means to model behavioral reflection. He proposes an architecture for 3-Lisp where every program P is interpreted by a continuation-passing metacircular interpreter, which is a program P_1 also written in 3-Lisp [Smi82, Smi84, dRS84]. Theoretically, the interpreter program P_1 is also executed by running an explicit copy of itself, say P_2, and so on up to infinity. This architecture resembles an infinite stack, where at the bottom ($level_0$) is the user program, and on top of it, there are a infinite number of interpreters, where the interpreter at $level_n$ is responsible of interpreting the program at $level_{n-1}$. This stack structure along with the crucial property of causal connection between its levels, is what is called the reflective tower. Actually, since the levels of the tower do not need to be based on interpreter techniques, they use the term *reflective processor program* (RPP) instead of interpreter, in order to avoid confusions [dRS84]. See Figure 2.2 for a graphical illustration. Part of the user program can be executed at the level of RPP_1 (in 3-Lisp, these special programs are called *reflective procedures*). Therefore, gaining access to its dynamic representation as it is modeled by its processor. By executing code at the next higher level, the program is inserting code into its processor, therefore modifying it.

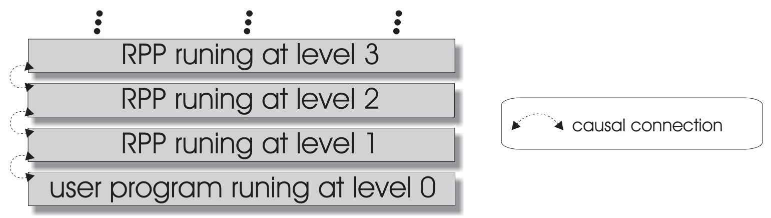


Figure 2.2: Reflective tower

In the reflective tower, there are an infinite number of levels at which a program is processed, each level is simultaneously active and has its own local state. As mentioned in [dRS84], this infinity can be eliminated in practice. They introduce the notion of *degree of introspection* of a program: in any single program P and input i, only a finite number of levels, say Δ, are needed to run the program. This number is the degree of introspection. Therefore, given Δ the level $\Delta + 1$ can be replaced by a non-reflective implementation of the processor G, eliminating the infinite levels.

Since Δ can only be determined at runtime, the implementation of G is proposed to be a *level-shifting processor* (LSP): such a processor is able, when it is determined (dynamically) that a new level of processing is required, to create the explicit state of the LSP on the fly as if it had run from the beginning of the program, and to resume computation from this state. This notion of jumping up in the execution level is called *shift-up*, see Figure 2.3. The optimal strategy is based on never run at a higher level that necessary, therefore they also provide the notion *shift-down* to decrease the level in which the LSP is running. A shift-down requires saving the level state, in order to be reinstalled the next time this level is reached.

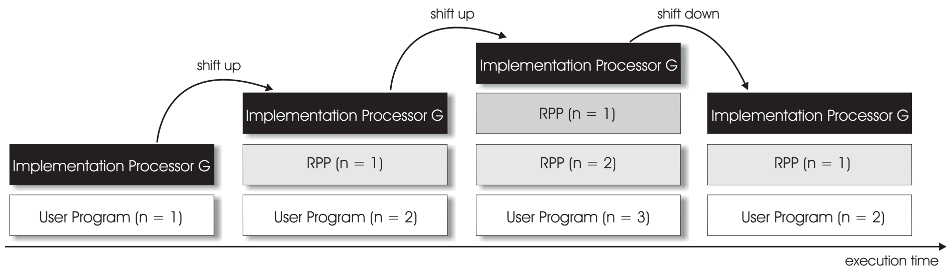


Figure 2.3: The level-shifting processor

As mentioned in [MJD96], reflective towers seems to be an ubiquitous notion in every system supporting behavioral reflection.

Two major papers from Wand and Friedman attempt to give a more formal denotational account of the concepts behind reflection and the reflective tower. In the first paper [FW84], they explain that the concept of reflection, as formulated by Smith, can be decomposed in two process (or operators), called *reification* and *reflection*, which respectively correspond to shift-up and -down.

Definition Reification

The process by which the state of the interpreter is passed to the program itself, suitably packaged (reified) so that the program can manipulate it.

Definition Reflection

The process by which program values are re-installed as the state of the interpreter.

They defined the state of the interpreter as: the interpreter registers holding an expression, an environment and a continuation. As further mentioned, the process of reification can be thought of as converting *program* into *data*. Such a data can be manipulated as values by reflective programs and transformed back into program by the reflection process.

In the second paper [WF88] they give a formal semantic account of the reflective tower, by using meta-continuation semantics. By doing so, they manage to formally explain the reflective tower without using reflection in the formalization.

As a final comment about terminology, the data representing a piece of program is also called *reification*. The process of reflection has been named differently by some authors, *absorption* [Ste94] and *deification* [Yok92, DS01]. From now on we shall refer to it as *absorption*.

2.3.3 Reflective Object-Oriented Languages

The advent of both object-oriented programming (OOP) and reflection was carried on in the 80's, by two different communities. Although OOP's initial steps can be tracked down to the 60's, it was not until the 80's that OOP was adopted by a large number of programmers and organizations [Bru02], and its was not until the 90's, when most of the advances in understanding OOP occurred. Once OOP gained popularity, most of the work in reflection was rapidly formulated in the context of object-orientation. The reason seems to be a good match between both, as explained in [Ibr90].

Object-orientation as a programming paradigm promotes major improvements over the older procedural programming. It shrinks the gap that exists between programs and the domain (or reality) being modeled, by introducing the notion of object [JCJ$^+$92]. In short, an object can be seen as an abstraction of an entity of the modeled domain, that encapsulates the data and behavior required to represent it. Objects are somehow *independent*. They communicate among them using messages, through well-defined interfaces, in order to collaborate to realize more complex tasks. Therefore, OOP allows to build programs that are closer to the real elements they model. Consequently, programs can be understood more easily, and the objects that compose them, are more likely to be reused. Furthermore, they support the notion of sub-typing allowing integrated means of localized extensions.

As mentioned in Section 2.3.2, a fundamental property of every reflective language (or system), is the capability it offers to introspect or intercede-on its self-representation (structural or behavioral). This self-representation must be at the appropriately level of abstraction, in order to improve its usability. This is where object-oriented modeling techniques can be used, providing a flexible, extensible and easy to use paradigm to structure the self-representation. Maes said in [Mae87b], *"the concept of reflection fits most naturally in the spirit of object-oriented programming"* and note the contribution that introduces the notion of object into the reflection world by saying *"an object is free to realize its role in the overall system in whatever way it wants to. Thus, it is natural to think that an object not only performs computation about its domain, but also about how it can realize this (object-) computation"*. Moreover, the communication between objects performing domain computation, so-called *base objects*, and objects performing meta computations, so-called *metaobjects*, can be based on a well-defined interface or protocol. Such an interface is commonly known as *metaobject protocol* (MOP), and shall be explained in grater detail in the next section. It allows to program base objects *independently* of metaobjects. Consequently, responsibility assignment stays in harmony, base objects keep focus on modeling domain entities, meanwhile metaobjects control how base computation is done. Taking some distance, this distinction between base and metaobjects can be seen as a two-layer, or n-level, architecture, where base objects reside at the *base level*, while metaobjects reside at the *metalevel*[8]. See Figure 2.4(a) for an illustration. The dash-line that encloses the reflective processor and the metalevel represents the customized processor, in the sense that its behavior has been modified. In addition, all object-oriented mechanisms and properties are still present: the implementations of base and meta -objects can be changed independently (thanks to the interface between them), metaobjects can be reused in different contexts and extension mechanisms (i.e. through delegation and sub-typing) can be used to extend metaobjects. This opens a great spectrum of possibilities, like developing metaobject libraries.

In addition, OOP also contributes in providing fine-grain control of the *locality* [KAR+93] of the reflective computation. *Locality* stands for the scope that changes done at the metalevel have on the base level computation. The object-based structuring of the self-representation allows to provide access to individual features of it (thanks to the object "*independency*"), which are isolated and can be affected without having to deal with the rest of the features.

The object-oriented community also benefits from reflection [Mae87b]. Initially, the community was facing some difficulties in finding an agreement on which the ideal object oriented language should be, it seems that the only answer was "it depends on the case". This is where reflection can help, by providing a way to build open-ended

[8]Note that from the reflective tower point of view, this conceptual division is present for each pair of adjacent levels.

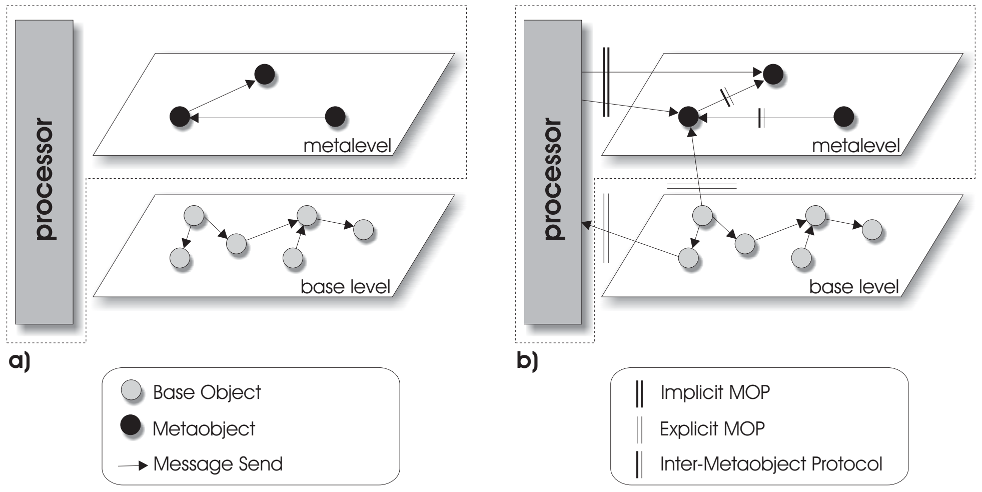


Figure 2.4: Object-Oriented Programming and Reflection.
a) Enhancements in modularization as consequence of applying OO techniques to structuring the metalevel.
b) Different types of MOPs

languages, where aspects of the language that were otherwise implicit are made explicit, through the use of MOPs, allowing the customization of the language behavior based on the user needs. A good example can be seen in the design of the CLOS MOP [KRB91]. Another benefit, which is actually more valuable, is the study of MOPs for SoC, which proves to be an effective tool for modeling non-functional (i.e. crosscutting) concerns. For instance, MOPs have been developed for a wide-rage of domains, including distribution [Str93], mobile objects [LBS00, TVP02], concurrency [WY88, MMY94], only to mention a few.

2.3.3.1 Metaobject Protocols

The key gain of this union between reflection and OOP, is the appearance of the MOPs. From the programming language perspective, a MOP provides a back-door to the language implementation; using the full power of object oriented techniques implicit aspects of the language are exposed through this interface in a proper abstract and encapsulated way. This interface is like any other standard interface between objects. Kiczales *et al.*define it as:

Definition Metaobject Protocol

Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language. [KRB91]

Kiczales *et al.* in the book *The art of the Metaobject Protocols* [KRB91], give a detailed explanation of the design of a MOP for the Common Lisp Object System (CLOS), which is an object-oriented variant of Lisp. In a short paragraph they explain how this concept can be applied to design a programming language which clarifies what is meant by a MOP:

> *"First, the basic elements of the programming language - classes, methods and generic functions - are made accessible as objects. Because these objects represent fragments of a program, they are given the special name of metaobjects. Second, individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects - a metaobject protocol. Third, for each kind of metaobjects, a default class is created, which lays down the behavior of the default language in the form of methods in the protocol."*

Zimmermann in [Zim96] clarifies the notion of MOP by presenting three different types of MOPs, depending on how the reflective computation is triggered (see Figure 2.4(b)):

Explicit MOPs are used by base objects to communicate to the metalevel. For instance, using this type of MOP, a base object may send messages to a given metaobject or may change a metaobject by another one. This usually results in explicit changes in the behavior of the base objects. In Figure 2.4(b), the arrow from the base object to the processor represents the base object using the explicit MOP to query or change some aspect of the metalevel (e.g. changing a metaobject or getting a reference to a metaobject). For example, a base object may change its status from volatile to persistent, by sending a message to the metaobject responsible for handling its persistence.

Implicit MOPs take place transparently: base objects do not know that a jump to the metalevel is occurring. The transparency at the object level comes from the fact that the interpreter itself triggers the meta computation. For example, each time an object is created, its state may be transparently initialized by informing of such an event to the persistence metaobject, which retrieves the object's state form storage. Analogously, when the object is destroyed, its state is updated in the storage.

Inter-metaobject Protocols are used by metaobjects to communicate with each other. This protocol is also explicit (implemented using standard message call) but is transparent for the base objects.

It is interesting to note that explicit and implicit MOPs usually collaborate in order to achieve a given behavior: the base objects use the explicit MOP to specify the desired semantics (e.g. being persistent), which is then implemented through the implicit MOP (e.g. intercepting object creation and destruction).

As reflective approaches matured, attempts to apply them in various domains have brought to the fore the need for further investigating structuring aspects of the metalevel. Most reflective systems are based on the reification of the structural concepts offered by the language (classes, methods, objects, etc), like in [KRB91, DS01]. In [McA96], the author characterizes this approach as a *top-down* one: taking the high-level concepts of the language and breaking them into its constituent pieces. Although this approach presents the advantage of a limited and particular set of concepts that are usually well-understood, it is hard to integrate new concepts or behaviors that have no foundation in the base language. Therefore, compromising the expressiveness and extensibility of the computations that can be implemented at the metalevel, two desirable properties according to him. Therefore, he formulates a *bottom-up* approach, which consists in starting from the basic operations (e.g. message send, field access, object creation, etc) defining the computational behavior of an object. This approach promotes the separation of the description of the computational behavior on an object form that of the base language, thus concentrating on what occurs, not how the description is organized. Object systems are therefore reduced to a set of conceptual operations, whose occurrences can be seen as the events which are required for object execution. This *operational decomposition*, as an approach to structure the metalevel, has shown to be both expressive and extensible [McA96].

The approach that is used to structure the metalevel has a direct impact in the design of its corresponding MOPs. For instance, if the metalevel is organized in terms of operations (e.g. like in Reflex, see Section 2.4) its implicit and explicit MOPs are also structured in terms of operations. Therefore, through the explicit MOP, a base object may configure what operations must be reified for each metaobject, and through the implicit MOP, the processor may trigger the metaobject execution, upon the occurrence of an operation.

2.3.4 Partial Reflection

The basic idea of partial reflection is to make use of reflection only when it is needed, therefore avoiding unnecessary reification. This idea was first motivated in the Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming

[Ibr90]. Among the topics addressed at the workshop were: measuring reflection, identifying which parts must be reified, efficiency and reflection. Brian Smith pointed out that most real world problems only require partial causal connections between the base level and the metalevel. Systems like 3-Lisp, where causal connection is full, are not the common case. He further noted that research efforts should focus on partial connections. They concluded that reflection is usually inefficient because compilation is the embedding of a set of assumptions and reflection is the retracting of some of these assumptions. Having these retractions everywhere, to achieve full reflection, is the cause of inefficiency. Therefore, having the proper mechanism in order to have a fine control of where reflections should take place, would be a great contribution. This mechanism is partial reflection.

In [Tan04] they clarify what it means for a system to be partially reflective by giving a characterization of its execution model. Let ξ be the semantic function representing the compiler or interpreter capable of executing the text of a program P. The application $\xi[\![P]\!]$ denotes an execution function that given an initial environment, executes the program producing the final environment. The execution expression for a standard program is

$$\xi[\![P]\!](D) \quad where\ \xi\ :\ Prog\ \rightarrow\ (Env\ \rightarrow\ Env)$$

where D represents the initial environment. Partial reflection implicitly divides P in two parts, P_r is the subset of the program to be reflected upon and P_{nr} is the complement. P_{nr} will be executed directly by ξ, while P_r will be interpreted by a (localized) metalevel program L. The execution expression for P_r will be

$$\xi[\![L]\!](P_r, D) \quad where\ \xi\ :\ Prog\ \rightarrow\ ((Prog\ x\ Env)\ \rightarrow\ Env)$$

which means that ξ will execute the metaprogram L receiving as a parameter P_r (to be interpreted by L) and the initial environment. Therefore, the partial reflection approach can be characterized by two coexistent expressions $E_{nr} = \xi[\![P_{nr}]\!](D)$ and $E_r = \xi[\![L]\!](P_r, D)$. This expressions shows that partial reflection requires an *hybrid execution model*. In the first model traditional execution can be carried out, while in the second, reflection techniques are required. This double execution model also illustrates that the optimization techniques used in reflection (like the ones mentioned [Tan04, p.44]) can also be used with partial reflection to optimize E_r.

Additionally, considering $|P|$ as a measure of the size of P (e.g. in terms of structures and execution points), ρ can be introduced as the *degree of reflectivity* of a partial reflective program.

$$\rho = \frac{|P_r|}{|P|} \qquad \rho \in [0, 1]$$

Intuitively this arbitrary measure reflects the fraction of structures and execution points reified by a program. As pointed out in [Ibr90], unnecessary use of reflection should be avoided to reduce performance overhead. Therefore, a crucial task to be considered in the design of a reflective system is minimizing ρ.

2.4 Reflex

Reflex is a reflective extension for Java, developed by Éric Tanter in his PhD. thesis [Tan04]. The reflective mechanisms provided by Java are mainly limited to structural/introspection. Reflex further extends those mechanisms by providing support for behavioral reflection (introspection/intercession) and a more complete support for structural reflection (introspection/intercession).

Behavioral reflection is provided through a comprehensive model of partial reflection (see Section 2.3.4), which allows to specify *where* and *when* reflection must occur, thus avoiding the costs of doing unnecessary reifications. In Reflex, the metalevel is structured in terms of *operations* (e.g. message send, object instantiation and Java cast operator), providing a fine-grained decomposition of the language behavior (see Section 2.3.3.1). Reflex is designed as an *open* reflective extension for Java. It is open in the sense that it does not impose any specific MOP [TNCC03, Tan04]. Indeed, Reflex allows metalevel architects to define their own MOP, based on the *Core Reflex* framework, possibly reusing parts of a *standard MOP* library. Architects can define which operations can be reified, and how (including the interface with the metalevel), by defining an operation support [TNCC03].

In addition, Reflex is an implementation of a language-independent conceptual model which was defined in [TNCC03] and later refined in [TN04b]. An overview of the conceptual model is presented in Section 2.4.1, which introduces the basic notions, fairly abstract, behind Reflex. Later in Section 2.4.2 the implementation model defined for Reflex is presented, refining the abstract notions in the context of Java.

2.4.1 Conceptual model

The conceptual model consists of three parts: behavior, structure, and composition. This section reviews the first and briefly mentions the second, since this work is mainly concerned about behavior. The presentation of the composition shall be deferred until Section 2.4.3.

Reflex approach to partial behavioral reflection relies on a *hookset model* [TNCC03], which consists of grouping execution points into composable sets and attaching some metabehavior to these sets through metalinks. The hookset model has three layers (see Figure 2.5 for a graphical illustration):

Hookset Layer is responsible for the selection of the execution points of interest. It relies on the notion of hook, which is an entity responsible for interception and reification of an execution point. A hookset is a composable set of hooks, possibly crosscutting the object decomposition.

Metaobject Layer is composed of metaobjects. The metaobjects are the entities responsible of performing the metacomputation on the reification points (hooksets).

Link Layer is composed of metalinks. The metalinks are the entities responsible for binding hooksets to metaobjects. From now on we shall refer to metalinks simply as links or behavioral links. In addition, they are responsible for assuring the *causal connection* between the base and meta level. Links are explicit in the model and highly configurable. They are characterized by attributes like: *scope* to specify the scope of metaobject; *control* to specify how the control is passed to the metaobject; *activation condition* to establish when the link must be activated or deactivated; among others. The most relevant attributes of links are explained in detail in Section 2.4.2.

The hookset layer is meant to provide the mechanisms to specify hooksets. Those mechanisms are expected to provide fine-grained selectivity of the execution points where reification must occur, so as to avoid unnecessary reifications. To this end, *spacial* and *temporal* selection should be provided: spacial determines where reification must occur, among the base level objects, and temporal determines when the reification must occur. The hooks in a hookset may be disseminated along the objects of the base level, therefore providing support for crosscutting metaobjects.

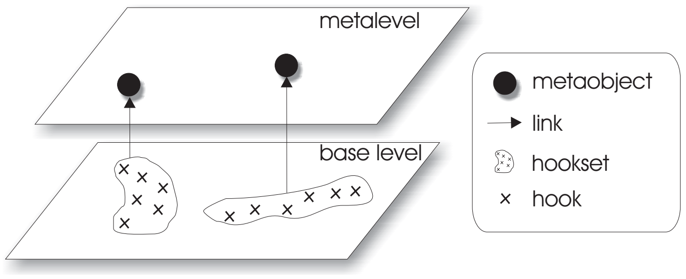


Figure 2.5: The Hookset Model

Structural reflection is achieved by providing a class-object model [TN04b], which in turn provides an intuitive structural description of the program. This is represented by class objects aggregating member objects (e.g. field, methods) and providing the possibility to perform intercession, i.e. structural modifications. The model organization is similar to the behavioral part, it defines the notion of *structural link* which binds a set of classes, so-called *classset*, to a structural metaobject. Structural metaobjects can introspect those classes and modify them.

2.4.2 Implementation Model

Reflex is implemented as a portable Java library [TNCC03]. It does not rely on any runtime support from the Java Virtual Machine (JVM) for implementing the reflective extensions, it is completely based on standard Java. In order to support the extensions it relies on bytecode transformation techniques, performed at *load-time*. Hooks adopt the form of a base level piece of code responsible of reifying the operation and jumping to the metalevel. Thus, during class loading, Reflex transforms Java classes into *reflective classes*, through the insertion of hooks at the appropriate places, along with additional infrastructure code required. By infrastructure code we mean, new fields (e.g. a field holding the reference to the metaobject) and methods (e.g. a method to initialize such a reference) that are inserted in the classes, in order to support behavioral reflection. The process of converting normal classes into reflective classes is explained in grater detail in Section 2.4.3. To make classes reflective, Reflex uses the Javassist framework [CN03] for load-time structural reflection. Javassist relies on a specific class loader to perform the load-time transformations.

There are four roles that participate in the development of a reflective application using Reflex: *(1)* the *metalevel architect* is responsible for designing the particular MOP, based on the requirements of the target application domain, see Section 2.4.2.2; *(2)* the *metaprogrammer* which implements the metaobjects classes, see Section 2.4.2.6; *(3)* the *base programmer* which implements the base application; *(4)* the *assembler* that links both levels by implementing the causal connection, see Section 2.4.2.3 to Section 2.4.2.5. As it shall be presented in the next sections, all the roles interact with Reflex though its API.

2.4.2.1 Example

The presentation of Reflex (and later AspectJ) shall be based on a simple example, a *Shape Editor System* (Figure 2.6). The system manages three kinds of shapes: `Line`, `Point` and `Composite`. A `Line` is represented by two points, one for each extreme. `Composite` is a shape container. All the shape classes are subclasses of `Shape`, which is an abstract class with one method, `moveXY`. This method is meant to perform a vertical and horizontal translation on the concrete shape, therefore, for `Point` it translates the point, for `Line` it translates both extreme points and for `Composite` it translates the overall shape by translating each inner shape.

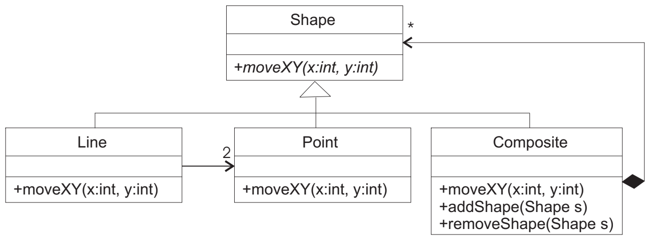


Figure 2.6: Simple shape editor system

In addition, the `Composite` shapes provide two additional methods: `addShape` which receives a `Shape` and adds it to its collection; `removeShape` which receives a `Shape` and removes it from its collection.

2.4.2.2 MOP Definition

The definition of the MOP is performed by the metalevel architect based on the requirements of the intended application. The metalevel architect must analyze those requirements with the purpose of determining the language operations that the MOP must be capable of dealing with. Once the requirements are clear, he/she can define the application's MOP by configuring Reflex to support the required operations.

Adding support for an operation implies specifying: how the language element represented by the operation is identified in the code, the information that must be reified upon an operation occurrence and the interface that metaobjects must implement in order to receive the reification of the operation. Reflex provides a comprehensive framework for specifying an operation support. The metalevel architect must provide several elements (i.e. classes and interfaces) in order to define an operation. Among those elements are:

- *Operation classes*. Operations are represented at load-time by *static operation* classes. An instance of a static operation class represents the static occurrence of an operation at the base level class definition (e.g. the invocation to a method occurring at the program text). The architect is responsible of defining the exact information that must be included in the representation. During hookset definition, the information exposed by the operation class should be enough to decide if a hook should be generated for it. Similarly, *dynamic operation* classes represent language operations during runtime, including state information of the particular operation occurrence (e.g. the *this* reference). The static operation class is mandatory, since it is crucial for the hookset selection process, while the dynamic operation class is optional.

- *Metaobject Interfaces* They are Java interfaces that specify what data is reified and how this data must be passed at runtime to the metaobjects, upon an operation occurrence. The metaobjects must implement the interfaces of the operations that they expect to process[9].

- *Hook Installers.* They are bytecode transformation entities responsible of parsing a class definition to find occurrences of a given operation class, and generating the appropriate hooks to install, if any.

For a detailed description of these elements and the implementation framework see [TNCC03, Tan04].

The specification of the particular MOP is done through the `OperationAPI`. The specification may include operations defined by the metalevel architect or from a comprehensive set of predefined Java operations provided by Reflex. From the point of view of the MOP classification presented in Section 2.3.3.1, such an API is part of the explicit MOP provided by Reflex. It allows to configure the implicit MOP, which is the one that upon the occurrence of a specified operation performs its reification (if it is selected by any hookset) and jumps to the metalevel. Typically, the MOP specification is provided in a special base level class, called the *configuration* class, along with the assembler binding information.

2.4.2.3 Hookset definition

Hooksets are responsible for performing spacial selection. They are meant to perform the selection during load-time, thus their decision is limited to the static properties of the operations. Hooksets are intentionally defined based on a selection predicate, represented by a *class selector* and an *operation selector*. The class selector determines which classes may contain operations of interest, later the operation selector determines which operations of those classes must be hooked (if any). Selectors are implemented as classes realizing some particular interface. The Listing 2.2 illustrates the implementation of a class and an operation selector. The former selects a class using its name, while the latter does the same with the operation name.

[9]The interface of the metalevel can also be defined by the assembler at link definition time. This shall be explained in detail in Section 2.4.2.5.

```
public class NameCS implements ClassSelector{
  private String itsName;
  public NameCS(String aName) {itsName = aName;}
  public boolean accept(RClass aClass) {
    return aClass.getName().equals(itsName);
  }
}
public class NameOS implements OperationSelector{
  private String itsName;
  public NameOS(String aName) {itsName = aName;}
  public boolean accept(Operation aOp, RClass aClass) {
    return aOp.getName().equals(itsName);
  }
}
```

Listing 2.2: Reflex Selector Definition Example <Java Code>

There are two types of hooksets, namely *primitive* and *composite*. The first is an operation-specific set defined by a triple: operation, class selector, operation selector. The primitive hooksets that are related to the same operation can be composed through standard set operators (union, intersection and difference). The second is the union of several hookset, possibly related to different operations. Listing 2.3 illustrates the definition of two primitive hookset that select all invocations to a method named `moveXY` inside `Composite` and `Line` respectively, and a composite hookset as the union of both.

```
PrimitiveHookset compHS =
   new PrimitiveHookset(MsgSend.class,
                        new NameCS("Composite"), new NameOS("moveXY"));

PrimitiveHookset lineHS =
   new PrimitiveHookset(MsgSend.class,
                        new NameCS("Line"), new NameOS("moveXY"));

CompositeHookset bothHS = new CompositeHookset();
bothHS.add(compHS);
bothHS.add(lineHS);
```

Listing 2.3: Reflex Hookset Definition Example <Java Code>

The possibility of composing hooksets through the set operators, makes hookset definitions a powerful reuse mechanism. In the same way, selectors allow the definition of general purpose selection predicates, like the ones in Listing 2.2, thus offering the possibility of reusing them.

2.4.2.4 Link definition

Links are specified during load-time by `BLink`[10] objects, and make it possible to specify and characterize (by their attributes) the association between hooksets and metaobjects. Among the most relevant attributes are:

- **control** specify how the control is passed to the metaobject, it can be passed `BEFORE` or `AFTER` the operation gets executed, or `AROUND` the operation execution (i.e. replacing its execution). In addition, control can also be passed both before and after the operation execution, using control `BEFORE_AFTER`.
- **scope** determines whether, for the associated hookset, there is one single metaobject controlling each and every hook (`GLOBAL` scope), or if each class involved has a particular metaobject handling hooks occurring within its instances (`CLASS` scope), or if each object has a dedicated metaobject (`OBJECT` scope)
- **activation** is a dynamically-evaluated condition to determine if the link is active or not. It is used to specify temporal selection.
- **mintypes** makes it possible to impose type restrictions on the metaobjects associated to the link.

Link definition also comprises the specification of how metaobjects must be instantiated or obtained. There are basically two means of obtaining metaobjects: either by instantiating a metaobject class or by querying a *metaobject factory*, which is a class exposing a method that once invoked, must obtain the metaobject instance (e.g. creating it or obtaining it elsewhere) and return it. Metaobject factories offer the possibility to have a fine-control of how the metaobjects are obtained and initialized, making it a very convenient way to set up an instance-based crosscutting metaobject. The *scope* and *initialization* (it supports eager and lazy initialization) link attributes also affect how and when metaobjects are obtained, but we shall not enter into further details about them, the interested reader may refer to [Tan04] for a complete explanation.

Continuing with the example, Listing 2.4 illustrates how a `BLink` object can be created and defined through the Reflex API. The defined link associates the previously defined hookset `bothHS` with a logging metaobject, obtained by instantiating the class `LogMO`.

The links are created and defined through the `LinkAPI` (i.e. `API.links()`). Note that the link `theCompLink` is defined using `AROUND` control, therefore it will replace the operation occurrence execution.

[10]The 'B' stands for behavioral links, in contrast to structural links also present in the model.

```java
BLink theCompLink =
       API.links().addBLink(bothHS, new MODefinition.MOClass("LogMO"));

theCompLink.setControl(Control.AROUND);
theCompLink.setControl(Scope.GLOBAL);
```

Listing 2.4: Reflex Link Definition Example <Java Code>

The support for link activation is optional. The activation attribute can be *disabled*, meaning that the link can not be activated/deactivated (i.e. it is always active), or it can be *enabled*, in which case an activation condition must be specified. The condition is encapsulated in an object exposing a method to evaluate the condition. The method may receive operation occurrence reified information, static or dynamic, as its parameters (specified through call descriptors, see Section 2.4.2.5) and must return a boolean value. For a detailed description of link activation see [Tan04]. An interesting characteristic of the activation conditions is that they can be changed during runtime, through `RTLink` objects. An `RTLink` object is a runtime representation of a link, which provides the means to access/change metaobjects and activation conditions. The `RTLink` objects can be obtained through the `LinkAPI`.

Note that the `LinkAPI` is also part of the Reflex explicit MOP. As it is mentioned in Section 2.3.3.1, it provides the means for base objects (and metaobjects) to obtain a reference to a metaobject in order to communicate with it.

2.4.2.5 Hookset Restrictions and MOP Descriptors

Hookset restrictions can be attached to a hookset in order to further refine its selection by setting restrictions on the operation's dynamic occurrence, thus complementing the hookset, which only base its decisions on the static occurrence. *MOP descriptors* describe, at the link level (as opposed to the operation level, see Section 2.4.2.2), which information has to be reified, and how, upon occurrences of a given operation. In addition, MOP descriptors allow to specify how the invocation to the metalevel must be done as to pass the reified information. Hookset restrictions (from now on simply restrictions) and MOP descriptors were proposed as extensions to the Reflex model in [RTN04]. The motivation behind those extensions relies on achieving a efficient and clear support of the AspectJ dynamic crosscutting mechanisms as we will see in Chapter 5. They were further refined in [Tan04].

MOP descriptors involve two notions: *Parameters* as objects describing how certain information is reified upon operation occurrence and *Call Descriptor* as a general purpose mechanism describing how to perform a method invocation.

Parameter objects implement a dedicated interface, `Parameter`:

```
interface Parameter {
  public String getCode(Operation aOp);
}
```

The role of a parameter object is to generate the source code that, when executed, results in the reference to the desired information. Standard parameters are provided, such as `CONTEXT` object, which refers to the currently executing object. Operation-specific parameters are provided by hook installers via *Parameter Pools*, e.g. `MsgSend` operation provides parameters for accessing the target object, or a parameter (at a given index) of the invocation. Finally, the user may define custom parameters by providing the source code using the extended Java language supported by Javassist [CN03].

Call descriptors consist of three elements: the name of the method to invoke, the name of the type declaring the method and the parameters (through parameter objects) to be passed in the invocation. Based on this description, Reflex can generate the appropriate hook code to invoke metaobjects. A MOP descriptor basically associates call descriptors to the various controls (`BEFORE`, `AFTER`, `AROUND`), specified either at link or hookset level, thus allowing a fine-grained specification of how reification must be done.

Listing 2.5 illustrates the specification of a MOP descriptor for the logging example. Suppose that the `LogMO` class has the method named `logit` that expects to receive the target `Shape` as a parameter.

```
MsgSendPool thePool =
            API.operations().getParameterPool(MsgSend.class);

theCompLink.setMOCall(Control.AROUND,
                       new CallDescriptor("LogMO", "logit",
                           new Parameter[]{thePool.getTargetObject()}));
```

Listing 2.5: Reflex MOP Descriptor Definition Example <Java Code>

In the example we attach a MOP descriptor at the link level, associating the only control specified for the link (i.e. `AROUND`) to a call descriptor invoking `logit` and passing the target object as a parameter[11]. Note that the parameter object representing the target object is obtained through the `MsgSend` parameter pool, accessible from the `OperationAPI`.

[11]Actually, the parameter object obtained is of type `Object`, while the `logit` method expects an object of type `Shape`. Reflex provides special parameters that cast the object to the appropriate type, but we do not include this with the purpose of simplifying the presentation.

Prior to MOP descriptors, the interface to the metalevel was defined by the metalevel architect during MOP definition. This definition was rather general with not possibility to be customized based on each metaobject needs. MOP descriptors bring this definition closer to the metaobjects, allowing the assembler to specify the interfaces of each metaobject and fine-tune which information must be reified. In addition, metaobjects now do not have to realize standard interfaces, which blur implementation.

A hookset restriction is a dynamically-evaluated condition that must be true in order for a hook to trigger reification and metaobject invocation. Such a restriction is hardwired in the hook code at generation time to improve performance. Hookset restrictions are specified when defining a link or hookset. In case that the hookset bound to a given link is compound, it is possible to set a restriction that applies to all or only some of its sub-hooksets.

A restriction is specified as a static method and hence can be computed based on globally-available information (i.e., static fields or methods), as well as parameters if needed. Parameters are specified when declaring the restriction using the call descriptor mechanism. The list of parameters must be compatible with the signature of the restriction method.

The code in Listing 2.6 continues with the example by restricting the hookset `compHS` to only select the shapes that are instances of `Point`. Note that this kind of selection can not be achieved with the hookset, because the type of the shapes grouped by a `Composite` shape can not be known statically. Thus, we set a restriction that checks that the target object is an instance of class `Point`. Also note that associating a similar restriction to the `lineHS` is not required, since we know in advance that it always calls `moveXY` over `Point` shapes.

```
// Method declared in class Restrictions
public static boolean onlyPoint(Object o){
  return (o instanceof Point);
}

// Restriction declaration, included in the configuration class
compHS.addRestriction("Restrictions", "onlyPoint",
                      new Parameter[]{thePool.getTargetObject()});
```

Listing 2.6: Reflex Restriction Definition Example <Java Code>

The restriction is associated to the hookset and is defined by giving the name of the class and the method that defines its logic, along with the parameters it receives (taken from the parameter pool)[12].

[12]The call descriptor mechanism is also used to specify the invocations to activation conditions,

2.4.2.6 Metaobject definition

Metaobject classes are free to realize the interfaces defined by the metalevel architect (see Section 2.4.2.2) or define their own interface at its convenience. The second alternative is the recommended one, for being more flexible and usually more efficient than the other one, because only the required information is reified.

Reflex provides a marker interface for metaobjects, called `BMetaobject`, which is the only requirement for the implementation of metaobjects. The metacomputation to be performed by the metaobject is implemented as a method of the class. The parameters of the method represent the reified information it expects, which must be compatible with the parameters specified in the corresponding MOP descriptor. The return type of the method must be `void` if the metaobject is associated with control `BEFORE` or `AFTER`, and `Object` for the `AROUND` control. The methods that replace an operation occurrence must return a value compatible with the value returned by the replaced occurrence. If the value is a primitive Java value, it must be boxed into an object; later Reflex takes care of unboxing the value at hook code level.

Reflex provides a mechanism, so-called `proceed`[13], that allows `AROUND` metaobjects to execute the operation occurrence that triggers the metacomputation. Metaobjects that make use of this mechanism must extend class `ProceedMO`, inheriting the method `proceed` that allows to continue with the execution of the operation. The value returned by this method is the one returned by the operation occurrence execution. In addition, the mechanism allows to modify the occurrence (e.g. by changing its arguments or the target object) before proceeding. To this end, the class `ProceedMO` provides the methods to perform such a modifications. For instance, the method `setArg` can be used to change the an argument before proceeding.

Continuing with the example, the code in the Listing 2.7 illustrates how the `LogMO` is implemented, making use of the `proceed` mechanism. The logging metaobject logs the calls to `moveXY` by printing a message before and after proceeding with their execution.

Proceed is deeply related to composition. Actually, one of its most interesting features is that upon proceeding, it may not only execute the replaced operation occurrence, but it may execute nested metaobjects that reify the same operation occurrence. This shall be explained in the next section.

see [Tan04]

[13]The mechanism is inspired by the AspectJ `proceed`, see Section 2.6.

```
public class LogMO extends ProceedMO implements BMetaobject {
  public Object logit(Shape aShape){
    System.out.println("Before " + aShape);
    Object r = proceed();
    System.out.println("After "  + aShape);
    return r;
  }
}
```

Listing 2.7: Reflex Metaobject Implementation Example <Java Code>

2.4.3 Composition

When several `BLinks` happen to affect the same operation occurrence, they *interact* [TN04a]. Usually, those interactions can not be resolved automatically, because they depend on the application semantics. However, interactions can be *detected* automatically. In Reflex, interactions are detected during load-time, in the *B-Link setup* (BLS) phase. See Figure 2.7 for a graphical illustration. Each class being loaded goes through that phase[14]. An initial selection step determines which are the `BLinks` that potentially apply over the class. Then, selection goes at operation level, determining for each operation occurrence inside the class, which are the `BLinks` that apply. If more that one `BLink` applies simultaneously over an operation occurrence, those `BLinks` go through the *Detection-Resolution-Composition* (DRC) step. The DRC step either is able to resolve the interaction and appropriately compose the `BLinks`, or inform the user that an interaction has been detected and not resolved. Reflex provides two mechanisms, *interaction selectors* and *ordering and nesting*, in order to specify the interaction semantics. Once the interactions are resolved, the hooks are generated and setup into the class code. If there is at least one `BLink` that applies, the result of the BLS phase is the *reflective class* otherwise the class stays intact.

The *interaction selectors* mechanism allows the user to specify if a link must apply in the presence of other links. The interaction selectors are attached to links. The interaction selector of a link is queried whenever the link is involved in an interaction, in order to determine whether it actually applies or not, which depends on the other links present in the interaction. An interaction selector is implemented by realizing an interface containing a single method, `boolean accept(LinkInteraction li)`, which receives a collection with the links that are interacting, and must return `False` when the link must not apply or `True` otherwise.

In addition to the interaction selectors, the users may specify the ordering and nesting relations among links. Reflex provides two composition operators, `seq` and `wrap`,

[14]Actually, each class before going into the BLS phase, goes through the `SLink` application phase, which sets up structural links. For more information see [Tan04].

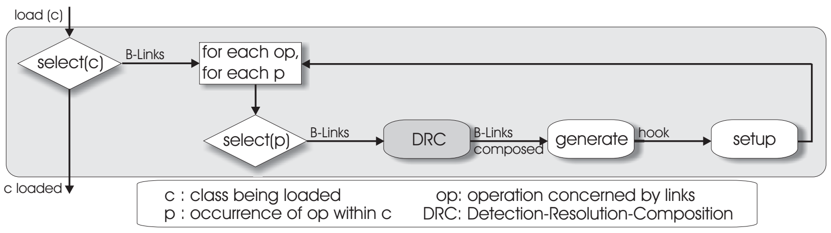


Figure 2.7: B-Link Setup Phase

dealing with sequencing and wrapping, that the user may use to specify the composition rules. The rule `seq(`l_1`, `l_2`)` means that l_1 must be applied before l_2. The rule `wrap(`l_1`, `l_2`)` means that l_2 must be applied within l_1. These rules are defined in terms of primitive operators, `ord` and `nest`, not dealing with links but with *link elements*: a link element is a pair *(link, control)*, where *control* is a control attribute value. Thus, in general, a link l_1 has three link elements: be_1 is the link element for `BEFORE` control, ar_1 for the `AROUND` control and af_1 for the `AFTER` control. The `ord` operator expresses sequencing between link elements of the same control. The `nest` operator expresses nesting between an `AROUND` link element and other link elements. For instance, `ord(`be_1`, `be_2`)` express that be_1 must be applied before be_2 and `nest(`ar_1`, `af_2`)` means that the application of the link element af_2 is nested within ar_1. Using this primitive operators, the rules are defined as:

```
seq(l₁, l₂) = {ord(be₁, be₂), ord(ar₁, ar₂), ord(af₁, af₂)}

wrap(l₁, l₂) = {ord(be₁, be₂), ord(af₂, af₁),
                nest(ar₁, be₂), nest(ar₁, ar₂), nest(ar₁, af₂)}
```

Note that the rules specify all the possible relations between the link elements of two links. Once the rules are applied over two concrete links, only the appropriate `ord` and `nest` relations apply. In order to illustrate this and the use of the rules, Figure 2.8 shows two examples of using both rules to specify the composition between two links. The result of the application of the rules to the links is illustrated graphically as a tree. The black circle represents the root of the tree.

- The first example composes one link with `AROUND` control and another with `BEFORE_AFTER`, thus the only valid link elements are: ar_1, be_2 and af_2. The result of composing them with `seq` is that the reifications are performed in the natural order: first be_2, then ar_1 and finally af_2. The result of using the `wrap` rule is that the link elements of l_2 are nested within the ar_1.

- The second example composes two links with `BEFORE_AFTER` control, thus the only valid link elements are: $\texttt{be}_1$, $\texttt{af}_1$, $\texttt{be}_2$, $\texttt{af}_2$. The result of using the `seq` operator is that the link elements of $\texttt{l}_1$ always are applied (i.e. the operation is reified) before the elements of $\texttt{l}_2$, respecting the control. The result of using the `wrap` operator is that the link elements of the second link are enclosed within the elements of the first.

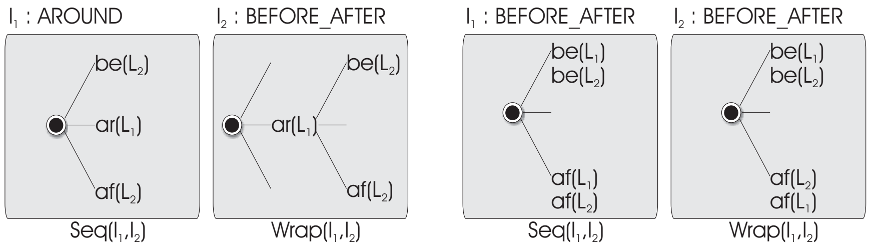


Figure 2.8: Composition operators

Finally, note that in the first example, when the links are composed with the `wrap` operator, link $\texttt{l}_2$ gets executed only if the metaobject associated to link $\texttt{l}_1$ invokes `proceed`.

2.5 Motivation for an Aspect-Oriented Kernel

In [TN04a, Tan04] the authors have made various observations about the current state of the art of aspect-oriented technologies that suggest the necessity for the development of an aspect-oriented kernel. Among those observations are:

Combining several aspect-oriented proposals Since the advent of AOP [KLM$^+$97], numerous AOLs have been proposed, along with different models for achieving a more effective modularization. The space of AOP is under exploration, and each proposal is a fixed point or a restricted region in the space. Each proposal differentiates from the others in terms of its specification language, its genericity, its binding time, its expressiveness, etc. Since the tools available for each proposal are not meant to be compatible with each other, the programmer facing the development of a system, is usually forced to choose the proposal that best suits his needs, even though the best option may be combining two or more of those proposals. The tools available are implemented with a close-world assumption: each tool eventually affects the base code directly, thus the combination of those tools, without appropriate coordination, may have unexpected results. The key idea is that providing the most adequate conceptual model and level of genericity for a given application domain actually depends on the specific situation: there is no definitive, omnipotent proposal that best suits all needs. Therefore, combining several aspect-oriented proposals seems promising[15], which in turn requires coordinating them. Having an aspect-oriented kernel, over which each different proposal could be implemented and coordinated, offers an attractive solution for this issue.

Aspect Interaction Aspects interaction further complicates the previous observation, since it occurs when one or more aspects (possibly implemented with different tools) are affecting the same program point. Providing the adequate semantics for performing the composition may be very difficult (if not impossible) by using heterogeneous tools. Even worst, since the tools perform the composition silently and blindly, the user may never become aware that an interaction has occurred and how it was resolved, thus introducing bugs which are hard to track. Implementing those approaches on top of a kernel that provides appropriate support for the detection an resolution of aspect interaction would offer a safer and correct way of resolving the interactions.

Easy Tool Implementation Each new proposal usually has to "reinvent the wheel", Since the program transformation work done by each AOP proposal is very similar, and can be factored out in a kernel for AOP.

An AOP kernel enables a wide range of approaches, from well-established to experimental, to coexist and collaborate without breaking each other. Such a kernel must provide core semantics, including structural and behavioral models, along with aspect interaction detection and resolution, generic enough to provide semantics for those approaches. In addition, the AOP kernel allows the designers of AOLs to experiment more comfortably and rapidly by using the kernel as a back-end, focusing on the best way for programmers to express aspects, rather that handling low level details.

As presented in [KLLH03], AOP is in essence a computational reflection mechanism, where the dynamic-crosscutting mechanism reflects a program's behavior: a join point provides the ability to introspect (see Section 2.3.2.1); advices provide the intercession (see Section 2.3.2.1) capability, meanwhile the static-crosscutting mechanism reflects the program's structure. This comes from the fact that, in the end, AOP provides high-level constructions to specify semantic alterations of applications written in the base language. A model that has some convincing history in describing semantic alterations is the model for structural and behavioral reflection. Therefore, it seems

[15]For instance, in [Ras01, SLS03] they experiment with the support of several aspect-oriented proposals, obtaining positive results.

to be a promising model for the core semantics of an AOP kernel. Furthermore, the techniques for partial reflection (see Section 2.3.4), promise reducing the performance issues typically associated to reflection, hence achieving an efficient kernel implementation.

2.5.1 Reflex an AOP Kernel for Java

In [TN04a] the authors have presented a detailed description of the requirements for an AOP Kernel and they put forward PBR as an appropriate low-level framework for the behavioral model of such a kernel. Also, they have claimed Reflex as an AOP kernel for the Java language, taking care of behavior, structure and aspect composition. The hooksets model and the class-object model take care of the behavioral and structural part, respectively (see Section 2.4.1). The generic layer for Detection-Resolution-Composition of link interactions provides a powerful framework for aspect composition (see Section 2.4.3). They further note that Reflex AOP kernel is centered in the study of AOP Kernels for a unique base language: they do not aim at multi-language support, or even more ambitiously at any software representation like IBM's CME [HOT$^+$02b, CME02], which attempts to address similarly UML diagrams for instance.

Such a complex artifact as an AOP kernel, into which different AO techniques can be translated, demands a formal understanding of its model's constructions. Furthermore, it demands having a reduced environment where theoretical case studies on the support for those AO techniques can be carried out. Previous works in the AOP area [asb, MKD02, MKD03b, WKD04, DFS02, DT04] have brought to the fore the need for satisfying these two demands in the context of AO techniques. For instance, the ASB project [asb, MKD02] (briefly presented in Section 2.2.2.1) has provided concise models of AOP for theoretical studies and a tool for prototyping alternative AOP semantics. In order to avoid difficulties to develop formal semantics directly from complex artifacts like AspectJ, the ASB provides a reduced environment consisting of a suite of interpreters of simplified AOP languages that allows to characterize the existing AOP approaches.

The motivation of the present work, regarding the construction of the Reflex Sandbox, relies on the following two observations. The first is that the only available description of the Reflex model is the informal semantics presented in [TNCC03, TN04a, Tan04] and its implementation in Java [RFX]. Therefore, a more abstract and precise description of the model semantics is need. Such a semantics would enable, for instance to carry out experiments on the support for AO techniques where the required transformations can be formally studied. The second observation is that the only available environment to experiment on the support of different AO techniques is Java, where its inherent complexity makes it harder to implement the required translations and often deviates attention from the central aspects of them. Therefore, having a reduced environment with an executable Reflex model, where the translation strategies to that model can be rapidly prototyped and tested, would ease the process of studying how an AO technique can be supported by the Reflex model.

Haskell is the programming language used to model the Reflex Sandbox. It has been chosen as the modeling language because it has a precise and clear semantics, and it provides high-level programming language constructs, suitable for rapid prototyping. Therefore, a formal development of the Reflex model in Haskell makes it possible both an abstract and precise semantics for the model, and an executable model. The high-level constructs provided by Haskell also make it possible to prototype translators form an AO technique to the executable Reflex mode. There are several works in the AOP community that have used Haskell as the modeling language [DMS01, DFS02, DS02, DT04]. The Haskell community also offers a set of techniques that simplify the process of building formal proofs.

2.6 AspectJ

Among the great variety of AOP proposals, AspectJ [KHH$^+$01] is a reference: it is a simple, well-designed and production-quality extension to the Java programming language, which allows a modular implementation of crosscutting concerns. AspectJ is a GPAL (see Section 2.2.3). It extends the Java language with a new unit of modularity, the *aspect*. In AspectJ, the core concerns of a system are implemented using Java, the base-level language, while the crosscutting concerns are implemented as aspects. The core concerns and the crosscutting ones are statically *woven* (at the byte-code level) by the AspectJ Compiler, producing class files that conform to the Java byte-code specification [HH04].

AspectJ supports two kinds of crosscutting, namely *dynamic* and *static*. Dynamic crosscutting makes it possible to define additional behavior to run at certain well-defined points in the execution of the program. Static crosscutting makes it possible to modify the static structure of a program (e.g., adding new methods, implementing new interfaces, modifying the class hierarchy). Since this work is concerned about the behavioral part of AspectJ, we restrict the presentation to the dynamic crosscutting mechanism.

2.6.1 Dynamic crosscutting Elements

AspectJ follows the Pointcut and Advice join point model for AOP [MKD03a]. To understand it, we need to introduce the AspectJ's notion of join point and two crucial elements for the dynamic crosscutting definition: pointcuts and advices.

In AspectJ, a *joint point* represents a well-defined point in the execution of a program, where program behavior can be extended with a crosscutting behavior. AspectJ supports different kinds of join points, which correspond to different operations of the underlying language, AspectJ: method-call, method-execution, field-get, advice-execution, among others.

The AspectJ language provides the means to group join points of interest into a *pointcut*, in order to specify the places where an aspect actually affects a base application[16]. A pointcut definition may also specify the *context information* that should be passed to the aspect (e.g., the arguments of the current join point). Pointcuts are specified using several primitive *pointcut designators* (PCDs) which can be combined using the standard logic operators. For instance, based on the example described in Section 2.4.2.1, the following AspectJ code:

```
pointcut move(Shape shape, int x, int y):
  call(* *.moveXY(..)) && target(shape) && args(x,y);
```

Listing 2.8: Pointcut Definition Example <AspectJ Code>

defines a pointcut named `move` that combines three primitive PCDs in order to select all calls to method `moveXY` made over an object of type `Shape`, and exposes both method parameters as context information.

Finally, the crosscutting behavior that should be applied upon occurrences of join points matched by a given pointcut definition is called an *advice*. An advice is a method-like construct that defines the additional behavior to execute at certain join points. When defining an advice, one must explicitly bind it to a pointcut. There are five *kinds* of advice, which differentiate the moment at which the advice is executed with respect to the join point execution: before (before the join point execution), after (after the join point execution), after throwing (after the join point execution, returning with an exception), after returning (after the join point execution, returning normally), around (replace the join point execution). An around advice can include a special `proceed` statement to trigger the execution of the captured join point, that is to say, the join point being replaced by the advice. Advices may have parameters, in which case they must be bound to the pointcut context exposure parameters. For instance, the following AspectJ advice:

simply ensures that if a shape has been locked, it does not move. For the moment, assume that `itsLockedShapes` is a collection that contains all the locked shapes and is accessible form the advice, in Section 2.6.2 this shall be clarified. The `proceed` statement takes as its arguments the arguments of the underlying around advice, and

[16]Actually, AspectJ supports *aspects on aspects*, which allows an aspect to affect other aspects in the same way it does to the base application.

```
void around(Shape aShape, int x,int y): move(aShape,x,y){
 if (itsLockedShapes.contains(aShape)){
   proceed(aShape,0,0);
 }else{
   proceed(aShape,x,y);
 }
}
```

Listing 2.9: Advice Definition Example <AspectJ Code>

returns whatever the around advice is declared to return. The values passed to the `proceed` statement can differ from those received by the advice, in that case the new values are replaced in the context of the captured join point before executing it. For instance, if the target shape has been locked, the around advice would forbid its movement by passing two zeros to the `proceed` statement. Therefore, the `moveXY` method would be executed with both `x` and `y` as zero, resulting in a null movement. Note that another implementation strategy may avoid performing such a null execution of `moveXY` method by not calling `proceed` when the shape is locked (i.e. simply returning). If the `proceed` statement is not invoked from the underlying around advice, the captured join point never gets executed.

2.6.2 The Aspect Unit

The aspect is the central unit of AspectJ, in the same way that the class is the central unit in Java. An aspect contains both dynamic (i.e. pointcuts and advices definitions) and static crosscutting elements, which together describe the implementation of a crosscutting concern. In addition to the AspectJ elements, aspects can contain fields, methods, and nested class members, just like a normal Java class. Also, an aspect may implement an interface or extend a standard Java class. For instance, we can merge the code in Listings 2.8 and 2.9 together in an aspect as shown in Listing 2.10.

This simple aspect definition encapsulates the crosscutting behavior of ensuring that every shape that has been locked, is not moved. Note that the aspect implements the `ILockShape` interface, which encloses the definition of the methods `lockShape` and `unlockShape`. These two methods can be used to lock or unlock a shape, respectively. Every aspect in AspectJ provides an implicit static method, named `aspectOf`, that can be used to get the aspect instance. Therefore, in the example a shape can be locked by doing: `ShapeLocker.aspectOf().lockShape(shape)`. Note that the responsibility of specifying which shapes must be locked is left outside the aspect.

By default, only one instance of an aspect exists in a virtual machine (VM), much like the singleton class. Those type of aspects are called *singleton aspects*. In addition,

```
aspect ShapeLocker implements ILockShape {
 Collection itsLockedShapes = new HashSet();

 void lockShape(Shape aShape){ itsLockedShapes.add(aShape); }
 void unlockShape(Shape aShape){ itsLockedShapes.remove(aShape); }

 pointcut move(Shape shape, int x, int y):
   call(* *.moveXY(..)) && target(shape) && args(x,y);

 void around(Shape aShape, int x,int y): move(aShape,x,y){
  if (itsLockedShapes.contains(aShape)){
    proceed(aShape,0,0);
  }else{
    proceed(aShape,x,y);
  }
 }
}
```

Listing 2.10: Aspect Definition Example <AspectJ Code>

AspectJ supports *per-object aspects* and *per-control-flow aspects*. The former allow to associate an aspect instance to each object of a particular group of objects of interest. The latter are somehow an extension of the per-object, where a separate instance is kept for each thread of execution encompassing a given join point. This work is concerned about singleton aspects, the interested reader may refer to [Lad03] for a detailed description of the per-object and per-control-flow aspects.

The aspect unit may include a special construct–`declare precedence`–to specify *aspect precedence*. Such a precedence declaration is used by AspectJ to resolve aspect interactions (see Section 2.2.2.1). The `declare precedence` construct is followed by a list of aspect names. The aspects on the left of the list dominate the ones on the right. For instance, consider two aspects, `A` and `B`. Both aspects declare a *before* advice defined over pointcut `P`. A declaration like `declare precedence: A, B;` would specify that the `A`'s advice must be executed before the `B`'s advice, because `A` has higher precedence. See [Lad03] for a complete description the semantics of the `declare precedence` construct.

2.6.3 Pointcut Designator

The PCDs in AspectJ are either user-defined or primitive. The user-defined PCDs are those that result from a pointcut definition[17]. For instance, the pointcut defined in Listing 2.8 results in a user-defined PCD, named `move`, which can be reused in the

[17]Note that the terms *pointcut* and *pointcut designator* are often used interchangeably.

definition of other pointcuts. The primitive PCDs are those provided by the AspectJ language.

The primitive PCDs can be divided into the following categories[18]:

- Kinded PCDs are those that capture a specific kind of join point. For instance, `call(MethodSignature)` captures method-call join points, `aexecution()` captures the advice-execution join points, `get(FieldSignature)` captures field-get join points, among others. The signatures embedded in the PCDs definitions allow to further restrict join points of a certain kind by specifying restrictions over its signature. AspectJ offers wildcards in signature restrictions for convenience. For example, `call(* *.moveXY(..))` (used in Listing 2.8) restricts joinpoints of kind method-call to calls to a method named `moveXY`, with an arbitrary number of arguments (through the `..` wildcard), having any return type (specified by the first `*` wildcard), and being defined over any class (specified by the second `*` wildcard).

- Control-flow based PCDs are those that capture join points based on the control flow of join points captured by another pointcut. There are two control-flow PCDs. The first is `cflow(Pointcut)`, and it captures all the join points in the control flow of the specified pointcut, including the join points matching the pointcut itself. The second is `cflowbelow(Pointcut)`, and it excludes the join points in the specified pointcut. For instance, consider the following pointcut definition

```
pointcut move(): execution(void *.moveXY(..))

pointcut moveControlledPoint():
  move() && within(Point) && cflowbelow(move())
```

  The pointcut `moveControlledPoint` selects those executions of the method `moveXY` of the class Point, and that are in the control flow of other `moveXY` method execution. In other words, it selects the `Points` that are moved as a result of moving a `Line` or `Composite` shape. Note that the `cflowbelow` is used in order to exclude the execution of `Point.moveXY` from the control flow restriction.

- Lexical-structure based PCDs are those that capture join points occurring inside a lexical scope of specified classes, aspects, and methods. There are two PCDs in this category: `within` and `withincode`. The former is used to capture join points *within* the body of a specified class or aspect. The latter is used to capture all the join points inside the body of an specified constructor or method.

[18]A more exhaustive presentation of pointcut designators can be found on the AspectJ website [aspb].

- Context Exposure PCDs are those that allow to expose some value in the context of the join point by binding it to a pointcut parameter. There are three PCDs in this category: `this` to bind the current executing object, `target` to bind the target object and `args` to bind the arguments of the join point. As shown in Listing 2.8, the binding is done by name matching, that is to say, `target(shape)` binds the pointcut parameter `shape` to the target object and `args(x,y)` binds both `x` and `y` to the arguments of the join point. As a matter of fact, binding a context value to a parameter also implies restrictions to be imposed. For instance, in the previous example the target object must be an *instance of* class `Shape` (because `shape` is declared as being of type `Shape` in the pointcut header) and the join point must have two `int` arguments.

- Conditional check is a PCD that captures join points based on some arbitrary conditional check. It is specified as `if(BooleanExpression)`. For instance, `if(System.currentTimeMillis() > triggerTime)` captures all the join points occurring after the current time has crossed the `triggerTime` value.

2.6.3.1 Pointcut Designators and Weaving

In the realm of the weaving process both join points and PCDs can be further described. A join point is an execution point thus it has a static correspondent at the code level, called the join point *shadow* [MKD02]. A join point may further be discriminated by a dynamically-evaluated condition, called the join point *residue* [HH04], in order to determine whether a runtime occurrence of the join point shadow actually is an expected join point.

The PCDs can be characterized based on the possibility of matching them statically or dynamically. *Statically matched* PCDs are those that can be resolved completely by looking at the program text. For instance, kinded PCDs match only certain kinds of join points (possibly imposing some signature restriction), which can be matched against each join point shadow in the code being processed. Also, lexical-structure based PCDs can be matched against the shadows, since they impose source-code location restrictions. *Dynamically matched* PCDs are those that require runtime information to determine whether they match a candidate join point or not. Such checks are implemented as *residues*. AspectJ supports three types of residues [HH04]: *control flow* residues for expressing control-flow based crosscutting, *instanceOf* residues to do runtime type checking, and *if* residues to evaluate arbitrary (though `static`) boolean expressions. The *instanceOf* residues are generated as a consequence of the `this`, `target`, and `args` PCDs, which define matching based on the dynamic type of the values exposed at a join point.

2.7 Modeling AspectJ using the Reflex AOP Kernel

There are two main reasons that motivate the present work for developing a case study on the support of AspectJ dynamic crosscutting mechanism semantics on top of the Reflex AOP Kernel. On the one hand, AspectJ provides an expressive language which includes features like control-flow based join point matching, context exposure, support for aspects on aspects, aspect precedence specification, join point reflective information, nested advices (through *around* advices). Therefore, providing semantics to AspectJ would put to the test the expressiveness of the Reflex model. On the other hand, the AspectJ Compiler [aspb] delivers an efficient language implementation. Therefore, the comparison on the performance of programs compiled using the AspectJ Compiler and programs compiled into Reflex would put to the test the promise of the PBR model of offering an efficient way to do reflection. These two reasons, in addition to the fact that AspectJ is a reference amongst the existing AOLs, make it clear that supporting it represents a big step in the validation of Reflex as an AOP Kernel.

Chapter 3

Modeling the Kernel Machine

This chapter describes the conception, design and implementation of the heart of the Reflex Sandbox, the Kernel machine. This machine is used to give an operational semantics to the Kernel language, which is a reflective extension of a simple object-oriented language called BASE. The Kernel language in turn embeds the core constructs of Reflex's partial behavioral reflection (PBR) model. The Kernel language and the corresponding machine have been formally written down in Haskell. This, we claim, has made it possible to express the semantics of the PBR model in a more abstract and simple way that it is achieved by the informal semantics described in [TNCC03, Tan04] and the current implementation in Java [RFX]. The Kernel machine, in addition, can be used to develop theoretical case studies concerning how the Reflex AOP kernel may support different AOP approaches.

This chapter is organized as follows. Section 3.1 presents the Kernel machine conception and the general aspects of its design. Section 3.2 presents BASE, the base level language of the Kernel machine, and the development of its corresponding execution machine. Finally, Section 3.3 presents the design of the Kernel language, which extends BASE with the constructions of the PBR model. The definition of the Kernel machine is also presented. In addition, this section provides a comprehensive conceptual description of the PBR model in the context of reflective languages.

3.1 Introduction

The Kernel language and its corresponding execution machine are conceived to provide an abstract, simple and precise way of expressing the semantics of the PBR model's core constructs. The only available description of the model is its implementation in Java [RFX] and the informal semantics in [TNCC03, Tan04], which is also presented based on the Java language. To avoid difficulties to develop formal semantics directly from an artifact as complex as Java, we use a simplified base-level language (BASE) on which the semantics of the reflective constructs of the model are defined. BASE is a simple yet powerful object-oriented language, which can be seen as a subset of Java.

The Kernel language, which extends BASE, provides a dedicated syntax for those constructs of the PBR model. This language-based approach used to express those constructs, as opposed to the API-based approach followed the Reflex implementation in Java (see Section 2.4), avoids the unnecessary noise resultant of being limited to use object-oriented syntax to express them. Therefore, simplifying the presentation of the model's constructs.

The Kernel language and its corresponding machine, which is used to give an operational semantics to the language, have been formally written down in Haskell. This has made it possible to express the semantics of the PBR model in a more abstract and precise way that is achieved by the informal semantics in [TNCC03, Tan04]. In addition, since the Reflex implementation in Java is based on code-transformation techniques[1] from which is difficult to grasp the semantics of the PBR model, the interpreter like implementation of the Kernel machine also offers a simpler description of the semantics.

The development of the Kernel machine shall be presented in two stages. The first involves the definition of the BASE language and the development of its corresponding execution machine, called the *BASE machine*. Then the Kernel language shall be defined, via an extension to the BASE language, and Kernel machine shall be developed by extending the BASE machine. The BASE language and BASE machine shall also be extended in Chapter 4 in order to build the *Pointcut and Advice* language and machine, respectively.

The architecture of the machines follows the *Pipe and Filter* architectural design pattern [BMR+96], commonly used in compiler design, see the Figure 3.1. It is structured in two main parts:

The Front End is responsible for performing the lexical and syntactic analysis, both of them implemented using monadic parser combinators [HM96].

The Back End is responsible for the semantic analysis and the program interpretation. The semantic analysis mainly consists on type-checking the program

[1] Actually, it is based on Javassist [CN03], a load-time metaobject protocol. Nonetheless, this tool can be seen as a code-transformation tool.

before it is executed by the interpreter. In order to simplify the implementation of the machines, the static type checker shall not be implemented. The interpreter assumes that the program checks.

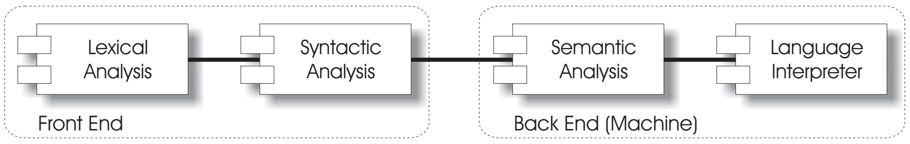


Figure 3.1: Machine Architecture

The interpreters are implemented using monads. Appendix A provides a comprehensive introduction to the Haskell language and to monadic programming. In addition, it also introduces the *state and exception* monad (`StE`), which models the computations that maintain a state and that may throw exceptions. The monad `StE` is used in the interpreter implementation to transparently propagate its state and the exceptions that may be raised, among the functions that compose the interpreter.

3.2 The BASE Machine

This section gives an overview of the design of the BASE language and its corresponding machine. The BASE language is inspired in the equally named language presented in the Aspect Sandbox [MKD03b, MKD02]. Basically, the language can be seen as a subset of Java; it is statically typed, it uses by value parameter passing style and is statically scoped. It is a class-based object-oriented language supporting single inheritance, overriding, sub-typing polymorphism [Bru02] and dynamic method dispatching, and without interfaces and overloading. The next section presents the syntax of the language. Section 3.2.2 briefly presents the implementation of its interpreter.

3.2.1 Language Syntax

This section exposes the syntax of the BASE language by first giving a short example of its concrete syntax and then presenting its abstract syntax in terms of Haskell's algebraic types. Appendix B presents a complete formulation of the concrete syntax of the language.

Listing 3.1 illustrates the implementation in BASE of the `Shape` and `Line` classes of the example introduced in Section 2.4.2.1. Ideally, the `Shape` class should be implemented as an abstract class or an interface. However, since neither abstract classes nor interfaces are supported by the language, it is implemented as a simple class where the body of the method `moveXY` is empty. Note that the syntax used for declaring classes is very similar to that of the Java language. A class is declared through the keyword `class` and its members (i.e. fields and methods) are lexically defined inside it. Note that the method's parameter names must begin with an `#`.

```
class Shape {
  void moveXY(int #aX, int #aY){}
}

class Line extends Shape {
  Point itsPoint1;
  Point itsPoint2;

  void init(int #aX1, int #aY1, int #aX2, int #aY2){
    itsPoint1 := new Point;
    itsPoint1.init(#aX1, #aY1);
    itsPoint2 := new Point;
    itsPoint2.init(#aX2, #aY2);
  }

  void moveXY(int #aX1, int #aY1){
    itsPoint1.moveXY(#aX1, #aY1);
    itsPoint2.moveXY(#aX1, #aY1);
  }
}
```

Listing 3.1: BASE Concrete Syntax Example <BASE Code>

The class `Line` is declared as a subclass of `Shape`, via the keyword `extends`. It has two fields that represent the points at the extremes of the line. Since the language does not support constructors, an initialization method, called `init`, is defined. This method instantiates and initializes its two fields based on the coordinates received as parameters. A class is instantiated through the operator `new` followed by the name of the class. Methods are invoked via the '`.`' operator, just as in Java. BASE does not make the distinction between statements and expression, everything is an expression. Although, an expression can be used as an statement by using the '`;`' symbol to delimit them. For instance, the body of the method `init` consists of four *statement-like* expressions which are executed sequentially. In the cases where a method returns a value, the value returned by the last statement-like expression in its body is the value that the method returns. The method `moveXY` is overridden. The new implementation invokes the `moveXY` method of the two `Point` fields, consequently translating the line.

The presentation of the abstract syntax is divided in two parts. Firstly, Section 3.2.1.1 presents the declaration language, showing the algebraic types use to represent class declarations. Then, Section 3.2.1.2 presents the action language, showing the algebraic types use to represent the valid expressions in the language.

3.2.1.1 Declaration Language

A BASE `Program` is a collection of class declarations, see Listing 3.2. A class declaration, `ClassDecl`, consists of the name of the class, the name of the super class and the list of members (i.e. fields and methods) which are declared lexically inside the class. Every class in BASE must, directly or indirectly, extend the class `Object`. The programmer may specify a super class or the class `Object` is assumed by default.

```
type Program = [ClassDecl]

data ClassDecl = Class ClassName SuperClassName [MemberDecl]

data MemberDecl = MthMember MethodDecl |
                  FldMember FieldDecl

data FieldDecl = Field Type Name

data MethodDecl =
        Method ReturnType MethodName [ParameterDecl] ExprBlock

data ParameterDecl = Parameter Type Name

type ExprBlock = [Expr]
```

Listing 3.2: BASE Declaration Language <Haskell Code>

The fields are declared by giving its type and name. Initialization expressions are not supported, during class instantiation fields are initialized with default values. A `Type` can be a primitive type or a reference type. The supported primitive types are `void`, `bool`, `int` and `string`. There are two kinds of reference type: class types and array types. Class types comprise user-defined types and predefined types (e.g. `Object`). An array is an indexed collection of elements. All the elements have the same `Type`, usually called the *component type* [GJSB00]. The array structure, like in Java [GJSB00], is encapsulated inside an array class. The interpreter automatically defines a class for each array component type used in the program (see Sections 3.2.2.1 and 3.2.2.2).

The declaration of a method consists of a name, a list of formal parameters, a return type and a body. The name of the method fully identifies the method inside a class, since overloading is not supported. The body is a sequence of expressions (i.e. `ExprBlock`), where the value returned by the last expression is the value returned by the method.

3.2.1.2 Action Language

The expression language, as shown in Listing 3.3, includes the following categories of expressions:

- **Literal expressions.** They include literals for string, integer and boolean values. In addition, the literal `NULL` is included, which represents the null reference value. Note that these literals are defined by the data type `LiteralExpr` not shown here.
- **Primitive expressions.** They include arithmetic, boolean, string manipulation and output expressions. Note that these primitive expressions are defined by the data type `PrimitiveExpr` not shown here.
- **Control expressions.** The only control expression supported in the language is the `if` expression. The concrete syntax supports both `if-then` and `if-then-else` styles.
- **Variable expressions.** The `LetE` expression declares and initializes a list of variables. The scope of those variables is the associated expression block. The expressions `VarGetE` and `VarSetE`, allow to get and set the value of a variable, respectively.
- **Object expressions.** The `NewObjectE` expression creates an instance of a class and initializes it with the default fields values. The `MethodCallE` expression performs a method invocation using dynamic method dispatching. The `SuperMethodCallE` expression invokes a super class method. The `FieldGetE` and `FieldSetE` expressions provide access to object fields. The `InstanceOfE` expression returns `true` if an object is an instance of certain class (including array classes). The `CastE` expression allows to check at runtime if an object is compatible with the specified type. Usually used to bypass the type system in order to perform a downcast from a reference of general type to a reference of a more concrete type.
- **Array related expressions.** They include expressions to create fixed length arrays, using default initialization (i.e. `NewArrayE`) or providing the initialization expressions (i.e. `NewArrayInitE`). In addition, it provides expressions to access (`GetArrayByIndexE`) or modify (`SetArrayByIndexE`) an array element

at certain index. Also, the array class provides methods to obtain the length of the array and to concatenate two arrays.

The language provides access to the current executing object through the keyword `this`. For simplicity, and as an aid to abstraction, object fields are not accessible from outside of that object's methods. We will assume that methods are publicly accessible from outside the object.

3.2.2 Interpreter Implementation

The BASE interpreter shall be presented in two parts. In first place, Section 3.2.2.1 presents the state structure used by the interpreter. Then, Sections 3.2.2.2 and 3.2.2.3 present a brief overview of the program evaluation and expression evaluation functions, respectively.

3.2.2.1 Interpreter State

The data type `State`, shown in Listing 3.4, represents the state of the interpreter. It is composed of a variable environment, a class environment, an object store and an output stream. The variable environment is represented by a stack of activation records, which binds variable names to values. The values are represented by the `Value` data type. A `Value` can be an object reference (i.e. `ValueRef`, where `ObjectID`

```
data Expr = LiteralE          LiteralExpr |
            PrimitiveE        PrimitiveExpr |
            IfE               Expr ExprBlock ExprBlock |
            LetE              [VariableDecl] ExprBlock |
            VarGetE           Name |
            VarSetE           Name Expr |
            FieldGetE         FieldName |
            FieldSetE         FieldName Expr |
            MethodCallE       ObjectExpr MethodName [ArgumentExpr] |
            SuperMethodCallE  MethodName [ArgumentExpr] |
            NewObjectE        ClassName |
            InstanceOfE       Type ObjectExpr |
            CastE             Type ObjectExpr |
            NewArrayE         Type Int |
            NewArrayInitE     Type Int [Expr] |
            GetArrayByIndexE  ObjectExpr Expr |
            SetArrayByIndexE  ObjectExpr Expr Expr
```

Listing 3.3: BASE Action Language <Haskell Code>

is the object identifier), a string, an integer, a boolean, the null value or the void value. The output stream is represented by a `String`. Each time a program writes a string to the output, the string is concatenated at the end of the output `String`.

```
data State = CState Environments Stores Output

data Environments = CEnv VariableEnv ClassEnv

data Stores = CSto ObjectStore

type Output = String

data Value = ValueRef ObjectID |
             ValueStr String |
             ValueInt Int |
             ValueBool Bool |
             ValueNull |
             ValueVoid
```

Listing 3.4: BASE Interpreter State - `State` Data Type <Haskell Code>

The class environment is where the definition of the classes in the machine resides. It is represented by the type `ClassEnv`, shown in Listing 3.5, which is defined as a list of `ClassDescriptor`. A `ClassDescriptor` models both array and user-defined classes. An array class, represented by the constructor `AClass`, consists of: a class name, the component type, a list of methods for the array and one field. The name of an array class is obtained by concatenating the array element type name with the string `[]`. For example, if we have an array of arrays of integers, the name of the array class would be `int[][]`[2]. The class includes a set of hardwired methods like `getLength`, `concat` and `toString` (overridden from `Object`). Those methods are implemented using the BASE language. The only field that an array class may have corresponds to the array length, which is used by the array methods. All array classes are direct subclasses of the `Object` class.

```
type ClassEnv = [ClassDescriptor]

data ClassDescriptor =
          CClass ClassName SuperClassName [MethodDecl] [FieldDecl]
          AClass ClassName Type [MethodDecl] FieldDecl
```

Listing 3.5: BASE Interpreter State - `ClassEnv` Data Type <Haskell Code>

[2]The names of user-defined classes cannot contain the symbol `[]`. Therefore, the names of an user-defined class and array classes cannot crash. Also, a user-defined class cannot extend an array class.

The user-defined classes, represented with the constructor CClass, are like the ClassDecl type used in the AST, but, while ClassDecl only includes methods and fields lexically defined inside the class, CClass also includes those methods and fields inherited. During class loading, the interpreter collects the fields and methods declared in the class and its super classes, in order to build the ClassDescriptor[3]. In the case of method overriding, it keeps the most specific method. The class environment defines the *class namespace*, binding each class name to a unique ClassDescriptor. Note that the structures used in the ClassDescriptor to model methods and fields, are those defined for the AST.

The object store is where instantiated objects reside. The ObjectStore, shown in Listing 3.6 is defined using the MapGenerator. This is a polymorphic type that provides an implementation of a mapping structure, which automatically generates the identifiers for the elements, i.e. the objects identifiers. The polymorphic type is parameterized with the identifier type (ObjectID for the object store) and the element type (ObjectObj for the object store). The ObjectObj data type contains the instance information. An instance of an array class consists of: its concrete class name, a value corresponding to the length field and a list with the array elements' values. An instance of a user-defined class consists of: its concrete class name and a list with the fields' values. The list of values holds one value for each field in its ClassDescriptor, respecting the order defined in the ClassDescriptor.

```
type ObjectStore = MapGenerator ObjectID ObjectObj

type Object = MapGeneratorPair ObjectID ObjectObj

data ObjectObj = CInstance ClassName [Value] |
                 AInstance ClassName Value [Value]
```

Listing 3.6: BASE Interpreter State - ObjectStore datatype <Haskell Code>

The Object type is defined using the MapGeneratorPair data type, which represents an element of the mapping. An Object contains both its identifier (used to reference it) and the instance information.

3.2.2.2 Program evaluation

The function runExc, shown in Listing 3.7, receives a Program as input and evaluates it. The evaluation may end normally or with an error. In the case where the evaluation ends normally, runExc returns the Value, output and State resultant of the program evaluation. Otherwise, it returns an exception. There are two activities that are

[3]This also applies to array classes.

performed before a program is actually evaluated, namely class loading and array classes definition. Both activities are done by the function `loadProgramStEState` (line 2). Loading the classes requires creating a `ClassDescriptor` for each class, which in turn requires collecting the inherited fields and methods from its super classes (see Section 3.2.2.1). Since BASE does not impose any restriction in the order that classes must be declared, the classes are first ordered and then loaded. In addition, `loadProgramStEState` also transverses the program, collecting all the array classes that the program uses[4] and defines the appropriate `ClassDescriptor` for them.

```
 1  runExc :: Program -> Exc ((Value, String), State)
 2  runExc p = applyStE (do loadProgramStEState p
 3                          val <- evalExprStEState initExpr
 4                          out <- getOutputStEState
 5                          return (val, out))
 6                     initialState
 7            where
 8             initExpr = (MethodCallE (NewObjectE "Main") "main" [])
 9
10  loadProgramStEState :: Program -> StEState ()
11    ...
```

Listing 3.7: BASE Program Evaluation Function <Haskell Code>

The type `StEState`, used for instance in the function `loadProgramStEState`, is actually a synonym of `StE State`, which represents the monads `StE` (see Appendix A) that carry a state of type `State`. As a general naming convention of the present work, the function names that end with the postfix `Exc` or `StExxx` are monadic functions working with the `Exc` or `StE` monads, respectively. The `xxx` portion of the name, in the cases where the `StE` monad is used, is the name of the state's type propagated by the monad. For instance, the function `loadProgramStEState` is a monadic function defined over the `State` data type. This naming convention shall also be used for the Kernel machine and the *Pointcut and Advice* machine.

The function `runExc`, besides invoking `loadProgramStEState`, evaluates the program using the expression evaluator function `evalExprStEState`, described in the next section. The default entry point for a BASE program is the method `main` of the class `Main`, thus, once the program has been loaded, the `runExc` function invokes the expression evaluator passing the `initExpr` expression. `initExpr` creates an instance of the `Main` class and invokes the `main` method.

The `applyStE` is the function that actually executes the monadic computation specified between brackets (lines 2 to 5). Remember that the monad `StE` is a function with the following signature `s -> Exc (v,s)`, so `applyStE` evaluates the monad by

[4]Note that this activity can be performed statically.

passing the initial state (`initialState`) from which the monad must be evaluated. `initialState` basically contains an empty variable environment, the class environment containing the class `Object`, the empty object store and an empty output.

3.2.2.3 Expression evaluation

The evaluation of the expressions is performed by a monadic evaluator function, called `evalExprStEState`, see Listing 3.8. This function is a standard recursive evaluator. It receives an expression as input (see Listing 3.3) and returns the value resultant of its evaluation. The function is defined by doing pattern matching over the expression to evaluate.

```
evalExprStEState :: Expr -> StEState Value
evalExprStEState e
    case e of
      (LiteralE l)             -> ...
      (PrimitiveE p)           -> ...
      (IfE e b1 b2)            -> ...
      (LetE vd b)              -> ...
      (MethodCallE oe n args) -> ...
      (NewArrayE t s)          -> ...
      ...
```

Listing 3.8: BASE Expression Evaluator <Haskell Code>

In order to illustrate the implementation of the evaluator, we will present the evaluation of the `MethodCallE` expression, see Listing 3.9. In general, the implementation of the evaluator makes use of several helper functions. For instance, `lookupObjectStEState` is used to obtain the `Object` structure from its reference and `lookupClassStEState` is used to obtain the `ClassDescriptor` structure from a class name.

```
case e of
 (MethodCallE oe n args) ->
    do ref <- evalExprStEState oe
       obj <- lookupObjectStEState ref
       cls <- lookupClassStEState (getObjectClass obj)
       (Method _ _ _ xp eb) <- lookupClassMethodStEState cls n
       vals <- mapM (\e -> evalExprStEState e) args
       createInstanceCallEnvStEState xp vals ref
       val <- evalExprBlockStEState eb
       dropEnvStEState
       return val
```

Listing 3.9: BASE Expression Evaluator (continued)

In order to evaluate a method invocation the `evalExprStEState` function must perform the following activities:

- *Line 3 to 6.* It evaluates the target object expression, which returns a reference to the target object. Using that reference, it first obtains the corresponding object, then the class of the object, and finally, using the `lookupClassMethodStEState`, it obtains the `MethodDecl` of the invoked method. The `MethodDecl` is needed to get access to the parameter list and the body of the method.

- *Line 7 to 8.* It evaluates the argument expressions, hence obtaining the argument values. Then, the function `createInstanceCallEnvStEState` creates a new environment for the invoked method, binding the formal parameter names to the argument values and the `this` object reference to the target object.

- *Line 9.* It evaluates the body of the method. `evalExprBlockStEState` evaluates all the expressions in the body sequentially and returns the value of the last expression.

- *Line 10 and 11.* It drops the environment created for the invoked method and return the value returned from the evaluation of the expression block.

The `lookupClassMethodStEState` function looks up the method in the `ClassDescriptor` of the target object concrete class. Note that since the `ClassDescriptor` already contains the most specific methods, see Section 3.2.2.1, all the method invocations through the `MethodCallE` expression are performed using dynamic method dispatching[5].

3.3 The Kernel Machine

This section presents the design of the Kernel language and its corresponding execution machine. The Kernel language embeds the core constructs of the Reflex's PBR model. We choose to include, within the language, those PBR model's constructs that together are interesting enough to explain the main concepts of the PBR model, and at the same time expressive enough to provide semantics to a subset of the AspectJ dynamic crosscutting mechanism. We encourage the eager reader to see Chapter 4 to know which are the constructs of AspectJ to be implemented on top of the Kernel language, and Chapter 5 for a full explanation of how they are actually implemented.

[5]We say that the method dispatching mechanism is dynamic because the invoked method is always determined based on the *dynamic type* of the target object.

Next section presents the requirements for the Kernel language by enumerating the PBR model's constructs that it must embed. Then based on those requirements Section 3.3.2 describes the design of the Kernel language, including the presentation of its concrete and abstract syntax. Section 3.3.3 presents a conceptual description of the machine operation. Finally, from Section 3.3.4 to the end of the chapter the implementation of the Kernel machine is described.

3.3.1 Language Requirements

The PBR model's constructs that must be included in the Kernel language are presented based on the Reflex implementation model, introduced on Section 2.4.2. Those constructs are:

Hookset Layer

- Hooksets. Support for primitive and composite hooksets must be provided, along with the operators for hookset composition.
- Selectors. Both class and operation selectors are required in order to specify the selection predicates.
- Restrictions. Support for dynamically evaluated restrictions must be provided, in order to further discriminate operation occurrences of interest.

Link Layer

- Links. Support for links must be included, specifying the binding between hookset and metaobjects. The only required link attribute is the *control* attribute, including support for `BEFORE`, `AFTER`, `BEFORE_AFTER` and `AROUND` controls. All links shall be implicitly defined using *hookset scope*. The runtime representation of a link is not required, however it must provide a mechanism to access the metaobject associated to a link.
- Call Descriptors. Support for the specification of the metalevel interface must be provided, along with a mechanism to specify the information that must be reified.[6]

[6]Note that the open characteristic of the Reflex MOP shall be partially supported, because the user can specify the metalevel interface, but he cannot specify the operations that the MOP is capable of reifying.

Metaobject Layer

- Metaobject. Support for metaobjects must be included, specifying the meta level behavior.
- Proceed. The proceed mechanism must be supported within metaobjects associated with the `AROUND` control, in order to resume execution of the current operation occurrence. It must also allow to modify of the operation occurrence.

In addition to those constructs just mentioned, the Kernel language shall be designed to support *operations* as the unit of decomposition of the base-level language. User defined operations are not required, only a fixed collection of operations are required. The operations that must be reified are[7]: method invocation, method execution, field get/set and object instantiation.

Finally, link interactions detection and resolution have been left as further work. Therefore, the language shall not provide any means for the user to specify how to resolve a link interaction. As a consequence of this decision, there are some constructions of AspectJ that shall not be fully supported by the Kernel language (see Chapter 5).

3.3.2 Language Design

Although the language requirements are presented in terms of the Reflex implementation model, the language design shall not be limited by it, in the sense that not every feature shall be modeled strictly as the implementation model is. The implementation model is actually a realization of the conceptual model of Reflex for the Java language. Therefore, it embeds some decisions that are most relevant in the context of Java. Consequently, the designed language shall stand in the middle between those models, incorporating some concepts of the implementation model but always with the goal of achieving the most faithful representations of the notions in the conceptual model in mind. As a result, some of the concepts of the implementation model will be adapted and others will be discarded. The following sections present the design of each of the language elements, which together fulfill the established requirements.

3.3.2.1 Operation Support

In the Kernel language, like in Reflex (see Section 2.4), the metalevel is structured in terms of metaobject reasoning and acting upon reification of base-level language's

[7]Note that this list of operations is not arbitrary. Each operation shall reify one or more of the join point kinds supported by the subset of the AspectJ language, see Chapter 5.

mechanisms described in terms of operations. An operation takes the form of a class. The fields of that class hold the representation of the language mechanism described by the operation. Whereas the methods provide the means to access (i.e. perform introspection on) and modify (i.e. perform intercession on) such a representation. The instances of an operation class represent the occurrences of the operation within a program. Unlike Reflex, the Kernel language shall provide a fixed set of supported operations, user-defined operations are left out. The operation classes are organized in a class hierarchy, where at the top of the hierarchy is the `Operation` class, conceptually abstract, and the concrete operations are subclasses of it, see Figure 3.2. As a direct consequence of the hierarchical organization of the operation classes, the language provides better support for the construction of general programs. For instance, it shall allow to define selection predicates over an `Operation` rather than over a concrete operation class, see Section 3.3.2.2.

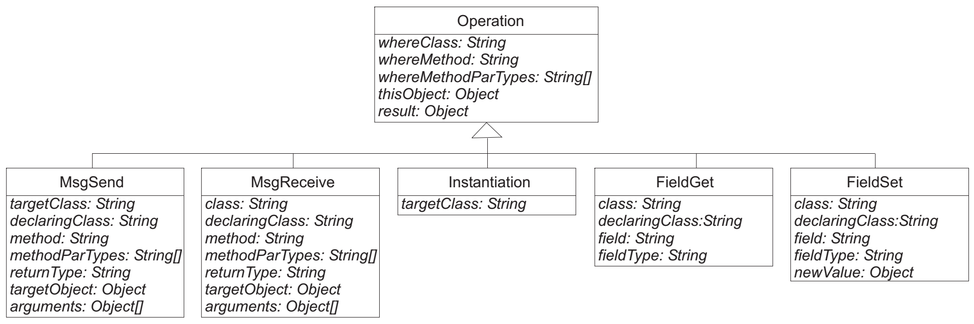


Figure 3.2: Operation Support

The reified information encapsulated inside an operation class is conceptually divided in static and dynamic information. Static information refers to the information directly available from the program text, which is mainly structural information. For example, consider the `Operation` class on Figure 3.2. The class where the operation is located (`whereClass`), the method where the operation is located (`whereMethod`), and its parameter types (`whereMethodParTypes`), are static information. The dynamic information refers to the information that is only available at program evaluation, usually associated to interpreter *state values*, like the `this` object (`thisObject`) and result value (`result`)[8] for the `Operation` class. Unlike Reflex, which provides for each supported operation a static and a dynamic operation class (see Section 2.4.2.2), the Kernel language provides only one operation class holding both static and dynamic

[8]The result value is the value returned by the evaluation of the operation occurrence, which clearly, is only available once the operation has been evaluated.

information, indistinctly. The reason is that, since the Kernel machine is implemented as an interpreter, there is no need to differentiate compilation time from runtime, as Reflex does. Therefore, from now on when we refer to an *operation occurrence* in the context of the Kernel language, we always refer to its runtime occurrence, which encompasses both static and dynamic information.

The reification of an operation occurrence also requires defining how the *structural information* shall be reified, e.g. the classes and methods. Reflex relies on the Java Reflection API in order to provide the reification of the structural elements, for instance, the `java.lang.Class` class represents the reification of a class and the `java.lang.reflect.Method` represents the reification of a method. Conversely, the BASE language does not provide any facility to reify any structural element. Since structural reflection is out of the scope of this work, we will represent structural information using strings. Consequently, introspection capabilities will be limited, but will be enough for our purpose. As shown in Figure 3.2, the structural elements in the `Operation` class and the other operations are declared with the `String` type.

In addition, the reification of an operation occurrence also requires to analyze how *values* shall be reified. The reification of a value, as any other reification, requires to build a representation of it. Since values are first-class elements of the language, they can be reified as themselves, unlike the structural elements which are not. However, in order to be able to treat all values uniformly, the values are reified as objects. This uniform treatment of the values allows, for instance, to represents the arguments of a method invocation using an simple `Object` array (see Figure 3.2). Thus, object reference values are reified as themselves, meanwhile the primitive values are wrapped into an object during reification. Therefore, the language provides one predefined wrapper class for each of its primitive types.

As shown in Figure 3.2 the supported operations are: `MsgSend` reifying the invocation to a method, `MsgReceive` reifying the execution of a method, `Instantiation` reifying the creation of an object (it does not reifies array instantiation), `FieldGet` reifying the access to a field and `FieldSet`, reifying the modification of a field. Note that `MsgSend` and `MsgReceive` reify the same conceptual language behavior from two different perspectives, *caller* and *callee* side, respectively. Just for illustration purposes, we present a description of the fields of the `MsgSend` operation (excluding the ones inherited from the `Operation` class):

- `targetClass` is the name of the target class. It is the name of the dynamic type of the `target` object, as opposed to the static type i.e. the type of the reference used to perform the invocation.
- `method` is the name of the invoked method
- `declaringClass` is the name of the class that declares the method.

- `methodParTypes` is an array containing the names of the method's formal parameters' types.
- `returnType` is the name of the type returned by the method.
- `targetObject` is the reference to the invoked object.
- `arguments` is an array containing the values of the arguments of the invocation.

3.3.2.2 Selector Declaration

In the Reflex conceptual model, hooksets stand for an abstraction of a set of operation occurrences of interest, while selectors appear in the implementation model as the mechanism to specify the set of hooks by intension. Actually, the implementation model supports several mechanisms for selecting operation occurrences of interest, which differ in terms of the time at which they are bound and evaluated [Tan04]. There are four different mechanisms: class and operation selectors which are statically bound and evaluated (selection can only be based on static properties of the operation occurrences), restrictions which are statically bound and dynamically evaluated (selection can also be based on dynamic properties), and activation conditions which are dynamically bound and evaluated[9]. This decomposition is required to achieve an efficient and yet flexible implementation of the selection process for Java. Typically, the later a mechanism is bound and evaluated, the more flexible and the less efficient it is. In Reflex, the combination of these four mechanisms allows to fully specify hooksets.

The Kernel machine is meant to formalize the notions behind the Reflex model, rather than achieve an efficient and flexible implementation of them. Consequently, we adopt a wider notion of selector, as a construction that allows to fully specify a hookset. A selector is a parameterized abstraction of a selection predicate, which can be based on static and dynamic properties of the operation occurrences.

In order to introduce them, let's see an example of a selector that matches all the operation occurrences within certain class, see Listing 3.10.

```
selector %inClassSel(String #aName){
  on Operation #op,
  when {#op.getWhereClass() == #aName; }
}
```

Listing 3.10: Selector Declaration Example <Kernel Code>

[9] Metaobjects can also be considered as a selection mechanism, dynamically bound and evaluated, but since they are not meant to perform selection, we leave them out.

The selectors are class-level declarations, declared using the keyword `selector`, along with the following elements:

Selector name is the unique identifier assigned the selector, in the example `inClass-Sel`[10]. Selectors have their own namespace.

List of parameters specifies the parameters that the selector expects to receive. Those parameters are accessible within the scope of the selection predicate. The selectors parameters enhance the reutilization of a selector. In the example, there is only one parameter, the class name.

Operation occurrence specifies a name for the operation occurrence, accessible within the scope of the selection predicate (i.e. `op`), and a type restriction establishing the operations admitted by the selector. The subtyping relation among operation types gives place to the definition of both *generic selectors* admitting any type of operation, like the one in the example, and *specialized selectors* admitting just one type of operation, for example replacing `Operation` by `MsgSend` in the example, admitting only method invocations.

Selection Predicate is specified by an expression block whose result must be a boolean value, true if the selector matches the operation occurrence and false otherwise. The expressions in the expression block can be defined using the full power of the expression language.

A selector essentially defines a function of the form

$$sel: \ (p_1, .., p_n) \longrightarrow op_{occur} \longrightarrow Bool$$

where $p_1, .., p_n$ and op_{occur} represent the selector parameters and operation occurrence respectively. Evaluating a selector requires two steps, binding the parameters and providing the operation occurrence. The first step is performed by the user in order to fix the meaning of the selection predicate during hookset declaration; to that end an application operator, so-called *apply*, is provided by the expression language in order to bind them (see next section). Selectors with bound parameters are first-class values of the language, we will call them *selector functions*. The second step is triggered by the interpreter in order to check if an operation occurrence is matched by the selection predicate.

This definition of a selector, proposes a cleaner and more simple mechanism to specify selection predicates, because overall selection predicate is defined in one place. In

[10]The identifiers of the constructions introduced by the Kernel machine, i.e. `selector`, `hookset`, `link` and `call`, must be preceded by the symbol `%`.

contrast with the definition of a selection predicate in Reflex, which usually ends up spread over several specialized constructions (i.e. restrictions, class selectors, operator selectors and activation conditions).

It is worth noting that selection predicates, just like selectors - restrictions - activation conditions in Reflex, are defined using the full power of the expression language. Therefore, their evaluation may not be decidable, because they may not terminate. Considering that selector evaluation is triggered by the interpreter during normal base program execution, decidability would be a nice property to achieve, avoiding bugs which may be difficult to catch. Furthermore, the fact that most selection predicates only require checking that the operation occurrence fulfills simple properties (where no recursion is required), suggest that the design of a decidable language for the definition of selector predicates should be further studied. However, it is out of the scope of this work.

3.3.2.3 Hookset Declaration

The language supports the declaration of hooksets in two forms: using a selector in order to express the set intentionally, so-called *primitive hooksets*, or by using standard set composition operators (union, intersection, complement) to compose already defined hooksets, so-called *composite hooksets*. Listing 3.11 illustrates them using an example.

```
// Occurrences inside Class Composite
hookset %hsInClsComp using {%inClassSel("Composite")}

// Occurrences inside method moveXY
hookset %hsInMthMoveXY using {%inMethodSel("moveXY")}

// Occurrences inside the method moveXY of Composite
hookset %hsInFooBar combine {%hsInClsComp && %hsInMthMoveXY}

// Occurrences outside Composite
hookset %hsNotFoo combine {!%hsInClsComp}

// Occurrences inside Composite and inside every method moveXY
hookset %hsInFoo combine {%hsInClsComp || %hsInMthMoveXY}
```

Listing 3.11: Hookset Declaration Example <Kernel Code>

The first two are primitive hooksets, defined using the keywords `hookset` and `using`. They consist of a name that identifies the hookset (i.e. `hsInClsComp` and `hsInMthMoveXY`) and a body that contains a selector function used to define the hookset. The syntax for the *apply operator* is similar to a method call. It consists of the name of the selector followed by a list of parameters between brackets. Composite hooksets are defined using the keywords `hookset` and `combine`, as in the last three declarations of the example, and consist of a name and a body containing a hookset composition expression. The available set operators are: complement (`!`), intersession (`&&`) and union (`||`). The composition expression specifies the new hookset by combining other hooksets (specified through their identifiers) with the set operators. Hooksets identifiers have its own namespace.

The notion of primitive and composite hooksets differs from the ones originally used in the Reflex implementation model. In Reflex, primitive hooksets select operation occurrences of the same type, while composite hooksets are the union of various hooksets. In addition, set combination operators (union, intersection and difference) are supported among primitive hooksets of the same operation type. In the Kernel language, we adopt a more uniform approach, where primitive hooksets are hooksets defined intensionally (possibly selecting operation occurrences of different types), whereas composite hooksets are constructed by composing other hooksets. Composition operators can be applied to compose any of those hooksets.

Quite frequently, primitive hooksets are defined using specific selectors, i.e. selectors that only exist to specify one hookset (are not meant to be reused). In order to avoid defining both the selector and the primitive hookset, syntactic sugar is provided for primitive hookset declaration, which allows defining the selection predicate and the hookset all at once. The Listing 3.12 shows the definition of the primitive hookset `hsMoveXY` that selects all the invocations to the method `moveXY`.

```
hookset %hsMoveXY using {
  on MsgSend #op,
  when {#op.getMethod() == "moveXY";}
}
```

Listing 3.12: Embedded Declaration <Kernel Code>

3.3.2.4 Link Declaration

The link is incorporated as a class-level langauge declaration, which binds a hookset to a metaobject. As shown in Listing 3.13, a link is declared using the keyword `link` followed by its name and a body containing the binding specification. The specification comprises the means to specify: which is the metaobject, which is the hookset and how the binding must be done. The metaobject is specified by giving an expression block that, once executed, returns the reference to the metaobject. This block, the so-called metaobject initialization block, is specified using the keyword `to`, see Listing 3.13. The binding hookset-metaobject is characterized by a *control* value, specifying how the control is passed to the metaobject (see Section 2.4.2.4), and a *call descriptor* which specifies how to invoke the metaobject and what information must be reified (see Section 2.4.2.5). There are three keywords, BEFORE, AROUND and AFTER, that correspond to the three supported control values. The hookset and the binding design are specified by giving the name of the hookset, the control value and the name of the call descriptor, using the keywords from, on and with. The example defines a link that logs all the movements of a shape, using a LogMO metaobject class. The metaobject is obtained by simply instantiating it and it is associated to the hsMoveXY hookset using the BEFORE control and the call descriptor callMoveXY, also shown in Listing 3.13.

```
link %lnkMoveXY {
  from %hsMoveXY on BEFORE with %callMoveXY,
  to { new LogMO; }
}

call %callMoveXY {
  from MsgSend #op,
  to LogMO #mo,
  do {
    let (int #x := ((Integer) #op.getArgument(0)).intValue();
         int #y := ((Integer) #op.getArgument(1)).intValue();)
    in {
       #mo.beforeMovingShape(#x, #y);
    };
  }
}
```

Listing 3.13: Link and Call Descriptor Declaration <Kernel Code>

The call descriptor is defined using the keyword call followed by its name. The declaration of a call descriptor encloses an expression block, see Listing 3.13 after the keyword do, which is responsible for collecting all the information that must be passed to the metaobject and performing the invocation. The variable environment used to execute the expression block holds two variable references: one for the operation occurrence object and one for the metaobject. The user defines the expected type and name of these two variables by using the keywords from and to, respectively. Back to the example, the call descriptor specifies that it expects a MsgSend operation and a metaobject of class LogMO, assigning to them the variable names op and mo, respectively. The expression block gets the reified values of the two arguments from the operation occurrence, those of the method moveXY, and invokes the metaobject method beforeMovingShape passing the two values. Since the metaobject method expects two int values, the call descriptor must unbox them before invoking the method.

Note that the means to specify the binding that we have presented so far, only allows coarse-grained specification. Reflex allows to bind different control values and call descriptors to sub-hooksets of the hookset bound to the link (see Section 2.4.2.5). These fine-grained specification of the binding is very useful under situations like the following:

- The hookset bound to the link is composite and it requires exposing different pieces of information for each of its component hooksets.
- The hookset bound to the link is composite and its component hooksets select different operation types, requiring a specialized (on the operation type) declaration of the call descriptor that extracts the required parameters and performs the metaobject invocation.
- The link is defined with `BEFORE_AFTER` control, thus performing two reifications, and requires one call descriptor for specifying each one.

In order to support this fine-grained binding specification, the `link` may include more than one *hookset-control-call descriptor* line it its body. This allows the declaration of links like `lnkFine`, see Listing 3.14, where for each hookset $\texttt{hs}_x$ a different call descriptor is associated. Note that the hookset bound to the link is not explicitly mentioned, we only mention its component hooksets which are $\texttt{hs}_1$ to $\texttt{hs}_n$. The hookset bound to the link would be the union[11] of all the component hooksets. In addition, we can also modify the declaration of link `lnkMoveXY` to log point movements before and after they occur. The new declaration of the link is shown in Listing 3.14, where for a single hookset (`hsMoveXY`) we associate two different controls and call descriptors.

Unlike Reflex, the language does not allow associating control `BEFORE_AFTER` to a hookset at once. However, this can be done by associating both controls separately to a hookset. Actually, the language does not restrict the user from associating, in one link declaration, the controls `BEFORE`, `AROUND` and `AFTER` to the same hookset. It also does not impose any restriction on associating one hookset with one control and another hookset, completely different from the first, with another control. In the case that, for the same control, more that one hookset matches an operation occurrence, the hookset being first bound lexically, would be the only one that effectively matches it. Thus, only one reification can be done per-control defined in the link.

In the cases that the call descriptor is associated with the `AROUND` control, the invocation to the metaobject method must return a value, replacing the value returned by

[11] Also note that, for a given composite hookset of the form `hs1 && hs2` it does not make any sense to have different bindings for `hs1` and `hs2`, since they do not match independently. In the case of a hookset of the form `!hs` it also does not make sense to specify the binding for `hs` alone.

```
link %lnkFine {
  from %hs1 on BEFORE with %call1,
  ...
  from %hsn on BEFORE with %calln,
  to { ... }
}

link %lnkMoveXY {
  from %hsMoveXY on BEFORE with %callLogBefore,
  from %hsMoveXY on AFTER with %callLogAfter,
  to { new LogMO; }
}
```

Listing 3.14: Fine-Grained Binding <Kernel Code>

the original operation occurrence. The expression block associated to the call descriptor is expected to return that value. Thus, its last expression must return the value. Note that the expression block may also alter the value returned by the metaobject or return a different value, although this is not common.

Reflex provides the `RTLink` as a runtime representation of a link. If we consider Reflex as a language, the `RTLink` is actually a reification of the the link construction, which otherwise would be implicit in the language. The `RTLink` allows to introspect a defined link, e.g. accessing the associated metaobject(s), and also provides means to intercede on the defined link, e.g. changing the associated metaobject(s). In the Kernel language, the only reification capability that is provided for the link construction is accessing its metaobject. This is done by providing a new language expression which reifies the metaobject of a link, see Section 3.3.2.6. Therefore, we only provide partial introspection capabilities and no intersession capabilities for the link construction[12].

3.3.2.5 Metaobject

The metaobject classes are declared just as any regular class. The only difference is that metaobjects associated through an `AROUND` link may use the `proceed` expression (see next section), as opposed to regular classes and metaobjects associated with `BEFORE` or `AFTER` controls which cannot use it. The return type of the methods used to perform the meta-computation, i.e. the ones invoked by the call descriptors, in the case of `BEFORE` or `AFTER` control usually returns `void`. However, they may be declared with any type, but the returned value shall be ignored if they are invoked as a consequence of a reification. The Listing 3.15 shows the implementation of the `LogMO`

[12]The reification of the link's metaobject is included in the language because it is needed in order to support the control flow constructs of the PA language. For a detailed explanation see Chapter 5.

used in the previous section. The method `beforeMovingShape` logs the invocations to `moveXY` by printing them to the output and returns void.

```
class LogMO {
  void beforeMovingShape(int #x, int #y){
    write "Before moveXY(" ++ $#x ++ ", " ++ $#y ++ ")"; newLine;
  }
}
```

Listing 3.15: Metaobject Declaration <Kernel Code>

In the case of methods associated with the `AROUND` control, they must be declared as returning `Object`, since the methods might be replacing operation occurrences returning different types.

3.3.2.6 Expression Language

The Kernel language extends the BASE action language (see Section 3.2.1.2) with four new expressions:

Selector Application . It allows the user to fix the meaning of a selector by binding its parameters, resulting in a selector function. It is usually used in the declaration of primitive hooksets. The syntax for the selector apply expression is shown in the first two hookset declarations of Listing 3.11.

Selector Evaluation It allows to evaluate a selector function using a particular operation occurrence object. It is commonly used together with the apply expression, in order to reuse a selector in the declaration of another selector. See Listing 3.16 for an example. It defines a new selector that matches the operations occurring inside a method of some class. Note that the `inClassSel` is reused by first binding its parameter, which is the name of the class to be selected, and then evaluating it, by passing the operation occurrence object.

```
selector %inClassMethodSel(String #aCName, String #aMName){
  on Operation #op,
  when {#op.getWhereMethod() == #aMName &&
        %inClassSel(#aCName):(#op); }
}
```

Listing 3.16: Selector Evaluation <Kernel Code>

Link Metaobject It allows to access (i.e. reify) the metaobject associated to a defined link. The syntax of the expression is simple: the name of the link followed by `#MO`. For example, the expression `%lnkMoveXY#MO` returns the metaobject of the link defined in Listing 3.13. Using this construction, it is possible to assign the same metaobject instance to two or more links, by using the expression at metaobject specification. In addition, it can be used in order to communicate among metaobjects or by selectors to check properties that metaobjects must fulfill.

Proceed It allows `AROUND` metaobjects to proceed with the execution of the replaced operation. The `proceed` can be evaluated alone, proceeding with the original operation, or it can be evaluated along with an expression block that allows to alter the operation occurrence. The Listing 3.17 shows an example of the use of `proceed` to modify the operation occurrence and proceed with its execution. The associated expression block has access to an implicit variable, called `oper`, which holds a reference to the occurring operation. Through that reference the expression block may modify the operation occurrence. In the example, it increments the displacement originally specified by the invocation to `moveXY`. Note that since a metaobject can be bound to a hookset that gathers operations occurrences of different types, the type of the implicit variable is `Operation`. Thus, in the example the operation must be casted to the correct operation type, before invoking `setArgument`.

```
class IncMO {
  Object incMovement(int #x, int #y){
    proceed {
      let (Integer #newX := new Integer;
           Integer #newY := new Integer;)
      in {
        #newX.init(#x + 100);
        #newY.init(#y + 100);

        ((MsgSend)#oper).setArgument(0, #newX);
        ((MsgSend)#oper).setArgument(1, #newY);
      };
    };
  }
}
```

Listing 3.17: Proceed Modifying the Operation Occurrence <Kernel Code>

3.3.2.7 Abstract Syntax

A `Program` in the Kernel language is a list of classes, selectors, hooksets, call descriptors and links declarations. A selector declaration (`SelectorDecl`), as shown in Listing 3.18, consists of a name, a list of parameters and a body (`SelectorBody`), which in turn consists of an operation declaration and the selection predicate. A hookset, represented with the `HooksetDecl` data type, encompasses the three different types of possible declarations, which are: `PrimSelUse` representing a primitive hookset declared using a declared selector, `PrimSelDecl` representing a primitive hookset with an embedded selector declaration and `Composite`, which is a hookset defined through the combination of other hookset. Each hookset declaration must have a name. The hookset combinators are represented by `HooksetExpr` data type, which models a set combination tree, where nodes are set operators (union, intersection, complement) and the leafs are declared hooksets (identified by their name).

```
data SelectorDecl = SelDecl SelectorName [ParameterDecl] SelectorBody

data SelectorBody = CSelBody OperationDecl ExprBlock

data HooksetDecl = PrimSelUse HooksetName SelectorName [ArgumentExpr] |
                   PrimSelDecl HooksetName SelectorBody |
                   Composite HooksetName HooksetExpr

data HooksetExpr = HooksetUnionExpr HooksetExpr HooksetExpr |
                   HooksetInterExpr HooksetExpr HooksetExpr |
                   HooksetComplExpr HooksetExpr |
                   HooksetNameExpr HooksetName
```

Listing 3.18: Selector and Hookset Declaration <Haskell Code>

The call descriptors are represented by the `CallDescDecl` data type shown on Listing 3.21. It consists of a name, two parameter declarations (specifying the type and the name of the variables representing the operation occurrence object and the metaobject) and an expression block.

```
data CallDescDecl = CallDesc CallDescName OperationDecl
                             ParameterDecl ExprBlock

data LinkDecl = Link LinkName [LinkCallSpec] ExprBlock

data LinkCallSpec = LCallSpec HooksetName Control CallDescName

data Control = CBefore | CAround | CAfter
```

Listing 3.19: Call Descriptor and Link Declaration <Haskell Code>

The `LinkDecl` data type represents a link, consisting of a name, a binding specification and the metaobject initialization block. The binding specification is given as a list of `LinkCallSpec`. The `LinkCallSpec` data type specifies the binding between a hookset, a control value (represented by the data type `Control`) and a call descriptor.

3.3.3 Sketch of the Machine Operation

In order to introduce how the Kernel machine operates, we will do a parallelism with the model of Reflective Towers proposed by Smith (see Section 2.3.2.2). The *Kernel Interpreter* shown in Figure 3.3 represents the level-shift processor of the tower. The Kernel Interpreter is able to determine (dynamically) when an additional level of interpretation is required, setup that additional level and resume the interpretation on it. This is done based on the defined links, once a link matches an operation occurrence (represented by **(1)** in the Figure), the operation is reified and the control is shifted to the metaobject associated to the link (represented by **(2)**). From the perspective of the base level computation, metaobjects act as interpreters of the reified operations, in the sense that they may introspect and intercede over the operation occurrence, consequently affecting the usual semantics of the operation. For instance, a metaobject may extend the usual semantics of a method invocation by logging the invocation occurrence. Therefore, in the context of the tower, the metaobjects conform a new level of interpretation between the Kernel Interpreter and the user program. The state of the interpretation level is determined by the state of the metaobjects residing in it. Since metaobjects are created at link definition time, during program loading and initialization (see Section 3.3.6), setting up the additional interpretation level only involves obtaining the reference to the metaobject associated to the link.

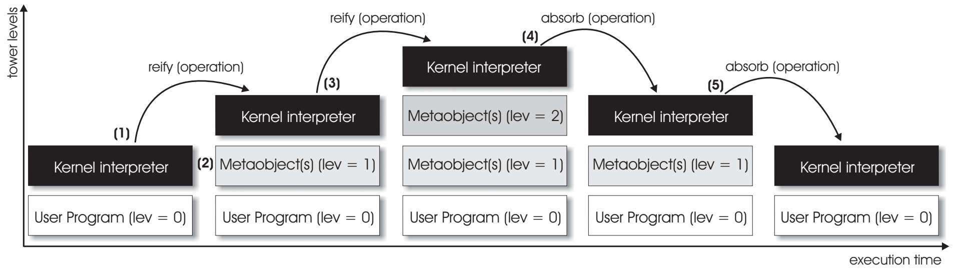


Figure 3.3: Kernel Reflective Tower

The degree of introspection (see Section 2.3.2.2), Δ, determine the number of additional interpretation levels that the tower must have. So far, we have considered the situation where $\Delta = 1$, thus, only one level of interpretation is required. If the program requires $\Delta > 1$, this means that further additional levels of interpretation shall be required. Setting up an additional level in the tower is done in the same way as for the first level, the only conceptual difference relies in that the new level represents a metaobject reifying another metaobject. This is represented by **(3)** in the Figure. There is no limitation in the number of levels the tower can have, but that number must be finite. The Δ is determined from the link specified in the program.

By taking a more general view of the tower, its exposed behavior is a continuous cycle of *reify-interpret-absorb*. Back to Figure 3.3, this cycle occurs:

- Level 1. Reify occurs at **(3)** in the Figure, whereas the interpret occurs from **(2)** up to **(5)** where the absorb occurs.
- Level 2. Reify occurs at **(3)**, while the interpret occurs from **(3)** up to **(4)** where the absorb occurs. Note that the cycle for Level 2 is immerse in the Level 1 cycle.

Step 1 - Reification. The reification of an operation occurrence can be viewed as a function, $reify$, that takes an interpreter structure and transforms it into a program $Value$. In order to introduce this function, we will refer to the interpreter structures as I_{elem}, where *elem* is a structure of the interpreter I.

$$reify : I_{elem} \longrightarrow Value$$

For instance, let's illustrate how the method call operation, represented by the interpreter structure `MethodCallE` (see Section 3.2.1.2), is reified. The reification of the operation results in the construction of an object, of class `MsgSend`, enclosing a representation of the operation occurrence.

$$reify(I_{MsgSend}) \ = \ obj_{MsgSend}$$

Note that the reification of the operation is not only limited to its static occurrence at the program text, but also comprises the particular evaluation of the `MethodCallE` expression occurring at runtime, within a particular state of the interpreter execution (see Section 3.3.2.1). The construction of the object representation also requires the reification of the elements, static and dynamic, that conform the method call expression, e.g. target method, target class, target object, this object, arguments, among others. As mentioned in Section 3.3.2.1 structural elements like classes and methods will be represented as strings, therefore its reification would be:

$$reify(I_{class}) \ = \ getClassName(class)$$

$$reify(I_{method}) \;=\; getMethodName(method)$$

returning the string values with the names of both elements. Since *this* and *target* are objects, they can be reified as themselves. There is no need to build any special representation of them.

$$reify(I_{obj_{target}}) \;=\; obj_{target}$$

$$reify(I_{obj_{this}}) \;=\; obj_{this}$$

The arguments that are objects are also reified as themselves, meanwhile the reification of the arguments which are primitive values requires boxing them in a wrapper class.

$$isObject(p_m) \;\vdash\; reify(I_{p_m}) \;=\; p_m$$

$$isPrimitive(p_m) \;\vdash\; reify(I_{p_m}) \;=\; \omega(p_m)$$

The reification step, besides building the operation occurrence, comprises the process of determining which are the links that match the operation occurrence and the evaluation of the corresponding call descriptors. The call descriptors actually determine what is the information required by the metalevel and perform the jump to the metalevel. This process is explained in more detail at the end of this section.

Step 2 - Interpretation. During interpretation, the actions that metaobjects can take are restricted by the kind of control offered by the links associated to them. Metaobjects with `BEFORE` control have access to the reified information. Since the reified operation is not executed yet, the return value is not accessible. Metaobjects with `AROUND` control replace the operation occurrence, therefore they do not only have access to the reified operation but they can actually perform a completely different behavior. The `proceed` mechanism can also be used to invoke the original operation occurrence. Metaobjects with `AFTER` control have access to the reified operation information, including the value returned by the operation.

The property that makes it possible for metaobjects to affect how the operation occurrences are interpreted is the causal connection as we will see in what follows.

Step 3 - Absorption. Absorption occurs once the metaobject execution ends and returns the control to the base level. All the changes performed by metaobjects must be absorbed by the base level, which may directly or indirectly affect its future computation. The interpreter implementation is responsible of assuring that the base level resumes its execution in the place it interrupted it and that changes done by the metaobjects are causally connected with the system [13] [14]. The way in which absorption takes place depends on how the execution control is given to the metaobject, i.e. the control attribute of its link.

- `BEFORE` Absorb must execute the reified operation after the metaobject ends its execution. Metaobject changes must be causally connected.
- `AROUND` Absorb must reinsert the value returned by the metaobject and continue with the evaluation that follows the reified operation occurrence. Reinserting the returned value may imply the unboxing of the value. Metaobject changes must be causally connected.
- `AFTER` Absorb must continue the evaluation that follows the reified operation occurrence. Metaobject changes must be causally connected.

Causal connection. The interpreter implementation is responsible for ensuring the causal connection between the levels of the tower. Causal connection is a bidirectional property; link implementation ensures that the metalevel is notified when some behavior happened at the base level (see Figure 3.4), thus ensuring that the metalevel is always up-to-date with the execution of the base level. Note that one level can be at the same time the base level of the level upwards and metalevel of the level downwards.

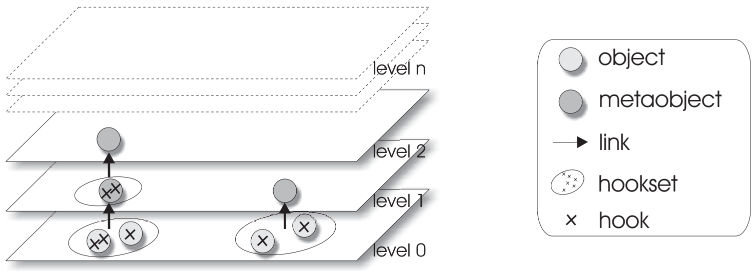


Figure 3.4: Causal Connection

In addition, the implementation must ensure that actions taken by the metaobject that affect base computation are causally connected with its base level (actually with the whole system). The actions that a metaobject may carry on affecting the base computation are:

[13]Note that absorption is a rather general concept. There are many different ways in which it can be understood. In this section we give only one vision of the concept, the one that we will follow.

[14]This assumes that the reified operations will never alter the "program counter".

- Invoke a method of some object reified in the operation occurrence (e.g. this, target, a parameter value or the return value). Since those objects are reified as themselves, any change in its state will be causally connected with the base level.

- Metaobjects with control around may invoke `proceed` so as to execute the operation occurrence. In addition, they may alter the operation occurrence (e.g. by changing a parameter value) before proceeding. As we will see in Section 3.3.5.1, each time an operation occurrence is reified a *closure* function capable of executing the operation is generated. This closure is responsible for assuring causal connection in the execution of the `proceed` expression.

- Metaobjects with control around may change the value returned by an operation. During the absorption, the implementation assures that the new return value will be inserted into the base computation, thus causal connection is assured.

- Perform some action that may affect the state of the interpreter, like writing to the output or creating a new object. Since the interpreter state is shared among all the interpretation levels of the tower, these changes are trivially causal connected.

In order to conclude the overview of the machine, we now illustrate in more detail the activities immersed in the reification of an operation occurrence. As shown in Figure 3.5, the reification consists of three phases: *selection phase*, *composition phase* and *evaluation phase*. The portion of the Kernel machine that is responsible for performing these three phases is what we call the *Reflective Core*, presented in Section 3.3.7. The occurrence of an operation triggers the *selection phase* (see Section 3.3.7.2) responsible for checking which links match the operation occurrence. Thus, for each `LinkCallSpec` defined in the link, it checks if its hookset matches the operation occurrence, which in turn ends up with the evaluation of the corresponding selectors. If the operation occurrence is not matched by any link, the execution continues normally, i.e. the occurrence is evaluated without performing the reification. On the contrary, if one or more links match the operation occurrence the process passes to the *composition phase* (see Section 3.3.7.3). The output of the selection phase is a list of pairs consisting of a `Link` and a list with the `LinkCallSpec` that matches.

In the case where more than one link matches a single operation occurrence, there is a link interaction (see Section 2.4.3) that must be resolved. The composition phase, based on the composition specification defined by the user[15], tries to resolve

[15]Note that the current implementation of the Kernel machine supports neither *interaction selectors*, nor *ordering and nesting* specifications (see Section 2.4.3). It provides a built-in, always successful, strategy to solve an interaction.

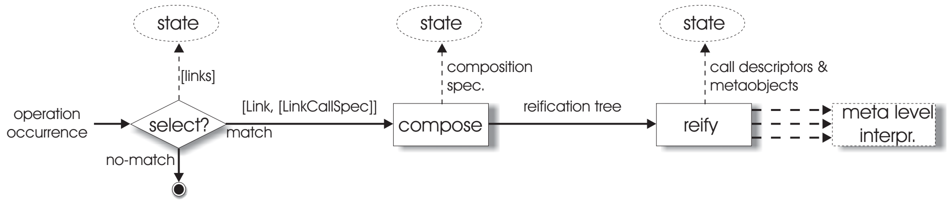


Figure 3.5: Selection – Composition – Evaluation/Reification

the interaction. If it successfully resolves it, a reification tree is generated, which describes the precise order, i.e. sequencing and nesting, in which the reifications must be performed. In the case it fails to resolve the interaction, it reports the problem to the user. The *evaluation phase* (see Section 3.3.7.4), based on the reification tree, performs the actual reifications (i.e. the reifications from the user's point of view) by evaluating the call descriptors, always respecting the order specified by the tree.

3.3.4 State Representation

The state of the Kernel machine extends the state of the BASE machine by incorporating environments for selectors, hooksets, call descriptors and links, along with a proceed stack. The `State` data type for the Kernel machine is shown in Listing 3.20. Since selector functions (see Section 3.3.2.2) are first-class values in the language, the `Value` data type is extended to incorporate them. They are represented by the data type `SelFunction` consisting of the name of the selector and a list of arguments.

```
data State = CState Environments Stores Output

data Environments = CEnv VariableEnv ClassEnv SelectorRep HooksetRep
                         CallDescRep LinkRep ProceedStack

data Stores = CSto ObjectStore

data Value = ... |
             ValueSel SelFunction

data SelFunction = CSelFun Name [Value]
```

Listing 3.20: Kernel Machine State Representation <Haskell Code>

The selector environment (`SelectorRep`) and the hookset environment (`HooksetRep`) contain the selectors and hooksets defined in the program, respectively. The `Call-`

DescRep data type, shown in Listing 3.21, represents the environment containing the defined call descriptors. The call descriptors are represented by the data type CallDesc, which contains the same information as the CallDescDecl elements of the AST. The LinkRep represents the environment containing the defined links, which in turn are represented by the Link data type. The Link consists of the name of the link, an ObjectID and the binding specification. The ObjectID is the reference to the metaobject obtained from the execution of the metaobject initialization block, performed during program loading (see Section 3.3.6).

```
type CallDescRep = MapGenerator CallDescName CallDescObj

type CallDesc = MapGeneratorPair CallDescName CallDescObj

data CallDescObj = CDesc OperationDecl ParameterDecl ExprBlock

type LinkRep = MapGenerator LinkName LinkObj

type Link = MapGeneratorPair LinkName LinkObj

data LinkObj = CLink ObjectID [LinkCallSpec]
```

Listing 3.21: CallDescRep and LinkRep Data Types <Haskell Code>

The proceed stack is a structure used for the implementation of the proceed expression, represented by the ProceedStack data type shown in Listing 3.22. Before performing the reification of an operation occurrence matched by an AROUND link, the reflective core pushes a closure into the proceed stack (of type Object -> StEState Value) and the reference to the operation occurrence object, both encapsulated by the ProceedClosure data type. The closure is a function built by the reflective core capable of continuing with the execution of the reification tree (see Section 3.3.3), which may imply performing other nested reifications or simply executing the original operation occurrence. Later, during the execution of the metaobject (associated to the AROUND link), if a proceed expression is evaluated, the interpreter pops the closure from the stack, along with the operation, and executes it. Thus, proceeding with the remaining of the reification tree. Note that a stack (or similar structure) is required since: nesting relations among AROUND links require many ProceedClosures coexisting and we may have AROUND links applying over AROUND metaobjects, which also requires many ProceedClosures coexisting.

In addition, the variable environment is also extended by incorporating location information, that is to say, when a new method environment is created during program evaluation, the variable environment also records to which method the environment corresponds. This information is later used during operation reification, to reify *where* the occurring operation takes place.

```
type ProceedStack = Stack ProceedClosure

data ProceedClosure = PClosure (Object -> StEState Value) ObjectID
```

Listing 3.22: `ProceedStack` Data Type <Haskell Code>

3.3.5 Implementation Strategy

Introducing reflection into the BASE language where a non-reflective interpreter is available, gives place to various alternatives (see Section 2.4.2 of [Tan04]). The first is making the interpreter reflective, for instance, following an approach like the one presented in [DS01]. They propose a generic reification technique that makes it possible to build, from a non-reflective metacircular interpreter, a specially-tailored reflective interpreter in which only the required elements are reflective. This technique is very attractive, since it presents nice properties of completeness and sound semantics. However we feel that using it would considerably increase the complexity of the implementation and may affect the original goal of having an implementation that allows a more simple transmission of the concepts behind the Reflex model. The second alternative is to use a strategy like the one that Reflex uses, which introduces hooks in the program code, as the interception and redirection mechanism. This strategy is rather oriented to compiled languages and is not aligned with our goal of obtaining a clear semantics of the Reflex model. The third alternative, which is the one that we shall use, is to modify the interpreter to create the interception mechanisms. This approach presents the advantage of having direct access to the internal structures of the interpreter, which provides greater flexibility and expressiveness. These two properties are of great value, considering that we want to experiment with new extensions to the Reflex model.

The interception mechanisms are introduced by modifying the pieces of the interpreter that are responsible for the interpretation of the supported operations. The interception for the `MsgSend` and `MsgReceive` modifies the interpretation of the `MethodCallE` and `SuperMethodCallE` expressions; the `FieldGet` and `FieldSet` modifie the interpretation of `FieldGetE` and `FieldSetE` expressions, respectively; and the `Instantiation` modifies the `NewObjectE` expression. Note that `Instantiation` does not reify the creation of array objects. In order to exemplify how these modifications are actually done, the changes introduced to the evaluation of the `MethodCallE`[16] expression by the `MsgSend` and `MsgReceive` operations are presented. The presentation is divided in two parts: the first one presents an operational view of the interception code, whereas the second presents a conceptual view of it, in terms of the concepts discussed

[16] In order to simplify the presentation, we focus only on the `MethodCallE` expression, the `SuperMethodCallE` expression requires analogous modifications.

on Section 3.3.3. The modifications introduced by other operations are analogous.

3.3.5.1 Operational View

Listing 3.23 shows the piece of the Kernel machine interpreter responsible for the `MethodCallE` expression evaluation. This code incorporates the interception mechanism to the original evaluation code of the BASE machine, presented in Listing 3.9 of Section 3.2.2.3. From the operational perspective, the interception code has the following responsibilities. The first is creating the operation occurrence object. Since for the `MethodCallE` expression there are two different perspectives, namely caller side (represented by the `MsgSend` operation class) and callee side (represented by the `MsgReceive` operation class), the interception code must create both objects. The second responsibility is executing the reflective core, which given an operation object, performs the process *selection–composition–evaluation* described at the end of Section 3.3.3. The third is building a closure that given the operation object executes the original operation. Finally, the fourth is to handle type boxing and unboxing of primitive values.

```
case e of
 (MethodCallE oe n args) ->
    do ref <- evalExprStEState oe
       obj <- lookupObjectStEState ref
       cls <- lookupClassStEState (getObjectClass obj)
       mth@(Method _ rt _ _ _) <- lookupClassMethodStEState cls n
       vals <- mapM (\e -> evalExprStEState e) args
       this <- lookupEnvValueStEState "this"
       op <- newMsgSendStEState this ref mth vals
       wVal <- RCore.evalStEState op (msClosure mth)
       unwrapPrimitiveTypeStEState wVal rt
    where
       msClosure mth op =
         do op' <- newMsgReceiveFromMsgSendStEState op
            RCore.evalStEState op' (mrClosure mth)
       mrClosure mth@(Method _ _ _ xp eb) op =
         do objR <- lookupObjectFieldStEState op "itsThis"
            argsR <- lookupObjectFieldStEState op "itsArguments"
            args <- lookupObjectStEState argsR
            vals <- unwrapArgumentsStEState (getArrayValues args) xp
            obj <- lookupObjectStEState objR
            createMthCallEnvStEState mth vals objR
            val <- evalBlockStEState eb
            dropEnvStEState
            wVal <- wrapPrimitiveTypeStEState val
            return wVal
```

Listing 3.23: `MsgSend` Expression Evaluation <Haskell Code>

The evaluation of the `MethodCallE` expression involves the following activities:

- *Line 3 to 7.* The target object and arguments expression are evaluated, and the method to be invoked is obtained, just as in the original evaluation code of BASE.
- *Line 8 to 9.* The operation object representing the caller side perspective of the `MethodCallE` expression evaluation is created, using the function `newMsgSend-StEState`. The function receives the reference to the *this* object (obtained at line 8), the reference of the target object, the `MethodDecl` to be invoked and the arguments' values of the invocation. Based on this information it instantiates the operation object, preforming the boxing of the arguments when it corresponds (this is presented in detail in the next section).
- *Line 10.* The reflective core is notified of the occurrence of the operation. It receives the operation occurrence object and a closure (`msClosure`). The reflective core checks if there are any links that match the occurrence, in which case it performs the reifications, otherwise it evaluates the closure. The closure is a function with the type `Object -> StEState Value` which given an operation object, evaluates the operation occurrence, thus continuing with the execution.
- *Line 13 to 15.* The closure `msClosure` passed to the reflective interpreter does not evaluate the original operation, but it constructs the operation object representing the callee side perspective using the `newMsgReceiveFromMsgSendStE-State` function. The function receives the `MsgSend` object and builds the corresponding `MsgReceive` object. Note that the `MsgSend` object will be passed by the reflective core, once the execution must continue. The reflective core is invoked again, passing the new operation object and the actual closure capable of executing the `MethodCallE` expression.
- *Lines 16 to 26.* Defines the closure capable of evaluating the `MethodCallE` expression. This closure obtains, from the operation object, the state values required to perform the invocation (lines 17 and 18), i.e. the target object and the argument values, and actually executes the method. Note that both closures receive the method to be invoked, along with the operation object, thus the method is not looked up again before it is executed. This implies that dynamic binding is not used (again) in order to determine the method to be invoked[17].

In the case of the `MethodCallE` expression, the boxing and unboxing may be required by the arguments of the invocation. The boxing of the arguments is performed once

[17]Also note that we do not check if the target object supports the method. This check must be done at runtime, but we leave it out in order to simplify the presentation.

the operation occurrence objects are created. Later, when the `mrClosure` is evaluated, in order to evaluate the operation occurrence, the argument values must be unboxed (line 20) before actually executing the method. Note that type unboxing requires the type information of the values, in order to unbox them. The value returned by the reflective core must also be unboxed (line 11), in order to be reinserted into the base level interpreter. Since the interception code is the one that has access to the type information, it is always responsible for handling the unboxing of the values. Actually, the interception code handles all the boxing and unboxing of the values, while the reflective core always receives and returns values in the boxed form. Note that since the `mrClosure` is meant to be invoked by the reflective core, it must box the result of the method execution before returning (line 25).

3.3.5.2 Conceptual View

In this section we shall present a conceptual analysis of the interception code of Listing 3.23 in terms of the concepts of reification, interpretation, absorption and causal connection, presented in Section 3.3.3.

Reification. The reification function, *reify* (presented in Section 3.3.3), which transforms interpreter elements into program values is partially implemented by the function `newMsgSendStEState` (line 9), responsible for transforming an occurrence of the `MethodCallE` expression evaluation into an object representation of it. The class `MsgSend`, presented in Section 3.3.2.1, defines the concrete representation used to represent the expression. Therefore, it determines exactly which information must be reified by `newMsgSendStEState`. The information to be reified by the function is divided in program structures and state structures. The program structures are: (1) the class and method where the operation occurs, obtained from the variable environment (see Section 3.3.4); (2) the target class and the invoked method, obtained from the target object and the `MethodDecl` received as a parameter. These program structures are reified following the guidelines of the function *reify*: the class structure is represented by a string value with the name of the class, whereas the method structure is represented by a string value with the name of the method, a string value with the return type name and an array of string values containing the names of the types of the method's formal parameters. The state structures are: the *this* object, the target object and the arguments values; all received as parameters. The object values are reified as themselves while the primitive values are reified into an object wrapping them. The reification of the `MsgReceive` operation is analogous to the `MsgSend` operation.

Once the operation occurrence object has been constructed, the reflective core is responsible for determining if a reification must really take place (lines 10 and 15), i.e. if there is any link interested in the operation occurrence.

Interpretation. The interpretation is performed when the reflective core is invoked (lines 10 and 15) and at least one link matches the operation occurrence, thus executing the metaobject(s). In case no links match the `MsgSend` operation or the `MsgReceive` operation, no reification occurs and the `mrClosure` is evaluated.

Absorption & Causal Connection. The causal connection from the base level to the metalevel is ensured by the implementation since each time a `MethodCallE` expression is evaluated, the reflective core is notified (lines 10 and 15), which in turn triggers the selection phase in order to perform the reifications of the interested links. In order to clarify how absorption and causal connection are ensured in the other direction, let's base our analysis on the control of the link that may be reified:

- `BEFORE`. The metaobject shall be executed before the closure execution. Therefore, any changes that it may have performed over an object, e.g. altering a property of the target object, shall be directly absorbed by the closure execution, because the identity property in the reification of objects trivially ensures the causal connection. Once the metalevel closure is evaluated, the changes made to the target object (obtained from the operation at line 17) will be reflected.

- `AROUND`. The metaobject replaces the occurrence of the operation, thus the original operation shall be only executed if the metaobject uses the `proceed` expression. If the metaobject uses the `proceed` expression, the causal connection is assured by the closure passed to the reflective core, capable of executing the operation occurrence. Note that changes made to the operation by `proceed` are reflected in the execution of the closure (lines 17 and 18)[18]. In addition, the value returned by the metaobject shall be absorbed by the base level interpreter (lines 10 and 11), which must unbox it before reinserting it in the interpreter. Also, the identity property of the reification of objets, assures that any other changes made to other objects by the metaobject are causally connected with the base level.

- `AFTER`. The metaobject shall be executed after the closure execution, thus having access to the result of the computation. The changes performed by the metaobject, over some property of the result object or any other objects shall be causally connected with the base level (because of identity property), and shall be absorbed by it. The changes over properties of the result value shall be absorbed at lines (10 and 11), since they will be visible when the value is reinserted.

Also note that the interception code ensures, by construction, that the base level interpretation resumes the evaluation at the place where it was interrupted.

[18]In the case of Reflex, the causal connection on the invocation to `proceed` is ensured by the Java Reflection API, since Reflex uses Java reflection to implement the execution of the original operation.

3.3.6 Program Evaluation

The program evaluation for the Kernel machine is essentially the same as for the BASE machine (see Section 3.2.2.2). The only differences rely on how the program is loaded and in the `initialState` used to evaluate the program.

Loading a Kernel program requires loading the classes (including metaobject classes), selectors, hooksets, call descriptors and links. The classes are loaded in the same way as for the BASE machine (see Section 3.2.2.2). The other four constructs are loaded by registering them in their corresponding environment. In addition, primitive hooksets and links require using the expression evaluator to evaluate the arguments of the selector application and the metaobject initialization block, respectively. For instance, consider the following primitive hookset

```
hookset %hsWithArgs using {%withArgs(new String[]("int", "int"))}
```

It selects the operations that have a specific list of parameter types, which is specified by giving an array with the type names. During program loading, the array instantiation expression is evaluated and the selector function (i.e. `SelFunction`, see Section 3.3.4), which corresponds to the selector application, is created. The links are loaded by first evaluating their metaobject initialization block, thus obtaining the reference to the metaobject, and then creating the correspondent `Link` which is registered into the `LinkRep` environment.

The metaobject initialization block and the arguments of the selector applications are evaluated with a non-reflective expression evaluator, which means that no operation reification is performed during its evaluation. The decision of not using the reflective interpreter is based on assuring that links do not accidentally interfere with each other, during the program loading. See Section 3.3.7.1 for a detailed explanation of the reflective and non-reflective evaluators.

The `initialState` now incorporates a new empty environment for each of the new environments and an empty proceed stack. The initial class environment, besides containing the `Object` class, now also contains the wrapper classes for each primitive type. Those classes are `Integer`, `Bool`, `String` and `Selector`.

3.3.7 Reflective Core

The reflective core is notified by the interception code (see Section 3.3.5) when an operation occurs, receiving the operation occurrence object and a closure capable of executing the operation occurrence, and is responsible for performing the actual reification (if it corresponds), that gathers the required information and invokes the metaobject. As introduced on Section 3.3.3, performing the reification involves a process of *selection–composition–evaluation*. In case there are no links that match the occurrence, there is no need to perform a reification, thus the closure is evaluated as to continue with the normal flow of the base level interpreter.

Before presenting the details of each of the phases involved in the reflective core, we shall review some circularity issues present in most implementations of languages that incorporate reflective mechanisms, along with its solutions. Since the Kernel machine is not an exception, we analyze the issues that affect its implementation, along with its solution.

3.3.7.1 Dealing with circularity

Incorporating reflection into a programming language has the consequence that some circularities appear in its design and implementation. For example, consider Java structural reflection, where classes are represented by `Class` instances. Note that the class `Class` is an instance of itself, revealing a circularity. This kind of circularity is commonly known as a *bootstrapping issue* and is resolved by creating the `Class` instance before the interpreter starts running, in an ad-hoc way, see Appendix C of [KRB91]. The concept of reflection is inherently circular, therefore, even though the circularity problems may be annoying, we must live with them. However, these circularities must be eliminated at the implementation level.

The Kernel machine does not have any bootstrapping issues, however it has another kind of circularity problem known as *metastability issue* [KRB91]. Metastability issues occur when some part of the implementation depends on itself, directly or indirectly, without base cases, thus conforming a non-well-founded recursion path. For example, in the Kernel machine this problem arises during selector evaluation. When an operation occurs, the Kernel would evaluate all the links to check if they match the operation occurrence. The evaluation of a link implies the evaluation of its hooksets, which in turn requires the evaluation of the selection predicates. Since the evaluation of a selection predicate may imply the occurrence of another operation (because selection predicates are expressed using language expressions), this evaluation may end in an infinite loop. As explained in [KRB91] this kind of circularities are usually solved by braking the loop at the implementation level.

This problem is solved by using a non-reflective version of the expression interpreter in order to evaluate the selection predicates. By non-reflective we mean that it does not reify any operation occurrence. This is aligned with the way in which Reflex works, since it does not reify the operations occurring inside: the class and operation selectors, restrictions and activation conditions. In addition, Reflex also does not reify any of its instrumentation code, therefore metaobject initialization code and the call descriptor generated code are neither reified. Although the metaobject initialization and the call descriptors do not constitute a metastability issue, we will not reify them in order to be fully aligned with Reflex.

In Reflex, not reifying the call descriptor only implies ignoring its code. In the Kernel machine, because of the strategy that we follow to implement hooks, i.e. modifying the interpreter implementation (see Section 3.3.5.1), we need to know when we must switch back to the reflective interpreter. Note that if we interpret all the expressions in the call descriptor with the non-reflective interpreter the metaobjects would also be interpreted with it. Consequently, the tower would only have a maximum of two levels. The switch strategy that we use is aligned with how Reflex does it. The first operation that Reflex reifies upon a reification is the `MsgReceive` operation, corresponding to the invocation to the metaobject method. In order to implement this we define the data type `SwitchOn`, shown in Listing 3.24 which defines two switch strategies: `NotSwitch` which never switches to the reflective interpreter, used for metaobject initialization and selector evaluation, and the `ObjectAccess` strategy which switches when the object referenced by the `Value` is accessed. Note that `evalExprStEState` and `evalBlockStESTate` receive the switch strategy as a parameter. Therefore, when the non-reflective interpreter is interpreting a call descriptor it will switch to the reflective interpreter when the invocation to the metaobject is performed, while the rest of the expressions are not reified.

```
data SwitchOn = NotSwitch |
                ObjectAccess Value

evalBlockStEState :: SwitchOn -> ExprBlock -> StEState Value
 ...

evalExprStEState :: SwitchOn -> Expr -> StEState Value
 ...
```

Listing 3.24: Non-Reflective Interpreter <Haskell Code>

There is another metastability issue in the Kernel machine that arises during the creation of an operation. Since operations are objects and object creation operation is reified, if we reify the operation occurrence object creation it would always end in a loop. As shown in Section 3.3.5.1 this issue is solved by taking a shortcut in the creation of those instances; not using language expressions to do so, but directly creating the instances in the interpreter interns. A more thorough discussion of the problems associated with implementing reflective languages can be found in [dRS84].

3.3.7.2 Selection Phase

The selection phase involves finding those links that match a given operation occurrence. A link matches an operation occurrence if at least one of its `LinkCallSpec` matches it, i.e. its associated hookset matches the operation occurrence. The function `collectLinksStEState`, shown in Listing 3.25, collects the `Links` that match the operation occurrence received as a parameter, along with the list of `LinkCallSpec` matched for each `Link`. Since more than one `LinkCallSpec` may match for a given `Link` and `Control`, only the first `LinkCallSpec` of each `Control` is matched (following the lexical order of definition). Therefore, the list of `LinkCallSpec` may have at most three elements, corresponding to the three `Control` values.

The hookset associated to a `LinkCallSpec` is evaluated using the function `evalHooksetStEState`. In the case of a primitive hookset, it evaluates its selector using `evalSelectorStEState`.

For a composite hookset, the union and intersection operators are evaluated using the `||` and `&&` logical operators, respectively, to combine the results of the evaluation of its components, and the complement operator is evaluated by using the `!` operator to negate the result of the evaluation of its component.

```
collectLinksStEState :: Object -> StEState [(Link, [LinkCallSpec])]
 ...

evalHooksetStEState :: Object -> HooksetName -> StEState Bool
 ...

evalSelectorStEState :: Object -> SelFunction -> StEState Bool
 ...
 where
   nonReflectiveExpr = evalBlockNRStEState newNotSwitch
```

Listing 3.25: Selection Phase <Haskell Code>

The `evalSelectorStEState` function receives the operation object and the selector to be evaluated and proceeds as follows:

- It checks if the type of the operation occurrence is compatible with the operation type expected by the selector. If an incompatibility is detected, the selector does not match the operation occurrence.
- It creates a variable environment holding the operation occurrence and the selector parameters (if any). Note that the selector parameters were evaluated during program loading, see Section 3.3.6.

- It evaluates the selection predicate. Note that as explained in Section 3.3.7.1, the non-reflective version of the expression interpreter must be used to evaluate the selection predicate, in order to avoid metastability issues. Therefore, the evaluation is performed using the `evalBlockNRStEState` interpreter with the `NotSwitch` strategy, since none of the expressions in the selector predicate must be reified. The interpreter returns to the reflective mode, once the evaluation of the selector predicate ends.

- Finally, it returns the result of the selection predicate evaluation.

3.3.7.3 Composition Phase

As explained in Section 2.4.3, Reflex provides two mechanisms, interaction selectors and composition rules, to specify how a link interaction must be resolved. The implementation of those mechanisms in the Kernel machine is left as future work. However, the current implementation provides a simple built-in strategy to solve a link interaction. The concept of *link element* (presented in Section 2.4.3) is represented, in the Kernel machine, as a pair `(Link, LinkCallSpec)`. Note that the input to this phase is a list of `(Link, [LinkCallSpec])`, where a given `Link` may have at most one `LinkCallSpec` associated for each control value. In other words, list of `(Link, [LinkCallSpec])` gives for each `Link` the list of link elements that apply. The strategy is simple: first perform the reifications of all the `BEFORE` link elements, followed by the sequential reification of all the `AROUND` link elements and finally the reification of all the `AFTER` link elements. The three sequential lists of reifications respect the lexical order in which the links are defined.

The built-in strategy makes use of a tree like structure, the so-called *reification tree*, to express the order in which the reifications must occur. Later, in the evaluation phase (see next section), this structure is evaluated in order to build the computation associated with the reification of the current operation occurrence. Even though the built-in strategy does not make use of nesting, the reification tree allows to express nesting relations and the evaluation phase is able to evaluate them. This design decision is meant to simplify the future extensions made to the Kernel machine to include support for composition rules' specification.

The reification tree structure is represented by the data type `ReifTree`, shown in Listing 3.26. A `ReifTree` is conformed by nodes, where each node may contain an arbitrary number of child nodes. A node is associated to a `LinkElement` which represents the reification that must be performed by the node. The `LinkElement` data type, represents either a link element (`LnkElem`), whose control is given by its `LinkCallSpec`, or represents a null link element (`LnkElemNone`). The nodes represent link elements that can nest other link elements, thus the link element associated to a node must be either a link element with `AROUND` control or a null link element. The link elements that are nested inside a node are: a list of `LinkElement` with `BEFORE` control, a list of child nodes containing link elements with `AROUND` control (possibly nesting other link elements) and a list of `LinkElement` with `AFTER` control. The root of the tree can be an `AROUND` node nesting all the other reifications in the link interaction or a node with a null link element which acts as a mere container of its nested link elements.

```
data ReifTree = RNode LinkElement [LinkElement] [ReifTree] [LinkElement]

data LinkElement = LnkElem Link LinkCallSpec | LnkElemNone
```

Listing 3.26: Reification Tree Structure <Haskell Code>

In order to illustrate how the reification tree can be used to express the solution to a link interaction, Figure 3.6 presents five interaction scenarios, along with an explanation of how they can be successfully resolved using the primitive composition operators `ord` and `nest`. For each scenario we show how a reification tree is built to represent the solution. Note that $\texttt{be}_x$, $\texttt{ar}_x$ and $\texttt{af}_x$ represent the `BEFORE`, `AROUND` and `AFTER` link elements of link $\texttt{l}_x$, respectively.

(1) There are two links, $\texttt{l}_1$ and $\texttt{l}_2$, with `BEFORE` control. Using the `ord` operator, the order between its `be` link elements is specified. The resultant tree is a single node tree with a null link element associated and it has the two `be` link elements in its list of nested `BEFORE` link elements (respecting the specified order). The `op` following the curved arrow represents where the occurring operation shall be evaluated during the reification tree evaluation.

(2) Now an additional `AFTER` link element is added to $\texttt{l}_1$ and also an additional link is added, $\texttt{l}_3$, with `AFTER` control. Again, using the `ord` operator, the order between the `be` elements and the `af` elements is specified. The resultant tree is analogous to the previous but it has the two new `af` link elements in its list of nested `AFTER` link elements (respecting the order).

(3) Now two additional links are added, $\texttt{l}_4$ and $\texttt{l}_5$, with `AROUND` control. In this case, suppose that the interaction is resolved only by using the `ord` operator, thus the link elements of the previous scenarios keep their order and the order between two new `ar` link elements is specified. The resultant tree is analogous to the previous but now the root node has two child nodes, one for each `ar` (respecting the specified order). Both new nodes have no nested link elements.

(4) Now suppose that the composition specification of the previous scenario is changed and the link elements $\texttt{be}_2$, $\texttt{ar}_5$ and $\texttt{af}_3$ are nested within $\texttt{ar}_4$. The tree for this scenario has three nodes. The root node contains the $\texttt{be}_1$ (resp.

`af`$_1$) in its list of nested `BEFORE` (resp. `AFTER`) link elements and has one child node for `ar`$_4$. The node corresponding to `ar`$_4$ contains the `be`$_2$ (resp. `af`$_3$) in its list of nested `BEFORE` (resp. `AFTER`) link elements and a child node for `ar`$_5$, which in turn does not nest any link element.

(5) The last scenario assumes a specification where the link elements `ar`$_4$ and `ar`$_5$ are in sequence. The former nests the two link elements of `l`$_1$, while the latter nests those of `l`$_2$ and `l`$_3$. The resultant tree has three nodes again. The root node only nests its two child nodes, whereas the nodes for `ar`$_4$ and `ar`$_5$ nest the correspondent `be` and `af` link elements in its lists of nested `BEFORE` and `AFTER` link elements, respectively.

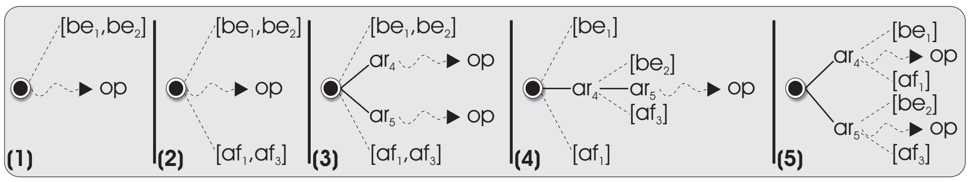


Figure 3.6: Link Interaction Scenarios

Although the reification tree is meant to express the order in which a link interaction is reified, a reification tree is also built in the cases that only one link matches. This allows the evaluation phase always receive a reification tree, independently of the number of links.

3.3.7.4 Evaluation Phase

The evaluation phase is responsible for the evaluation of the `ReifTree` produced by the composition phase. As a result, it will perform all the reifications specified in the tree, respecting the order and nest relations among them. Performing a reification implies gathering the information that must be actually reified, based on the call descriptor specification, and jumping to the metalevel by invoking the metaobject.

The evaluation of the `ReifTree` is built on top of three lower level functions: `rfySimpleStEState`, `rfyNestStEState` and `rfyOrdStEState`. The first function performs a simple reification, i.e. reification with `BEFORE` and `AFTER` controls, where nesting relations are not present. The second function is meant to reify the nodes of the `ReifTree` where nesting relations are present. The last function allows to reify an ordered list of reifications which can be formed by simple or nested reifications. Note the parallel between the `rfyNestStEState` and `rfyOrdStEState` functions with the two composition operators, `ord` and `nest`.

The `rfySimpleStEState` function is shown in Listing 3.27. Its input is the `LinkElement` to be reified along with the operation occurrence object. The function obtains the reference to the metaobject associated to the link and retrieves, from the state, the call descriptor associated to the `LinkCallSpec`. Then it invokes `reifyStEState` which actually performs the reification by evaluating the call descriptor expression block. We will delay the presentation of `reifyStEState` until the end of this section. Note that `rfySimpleStEState` always returns an empty result (i.e. `()`), since `BEFORE` and `AFTER` reifications are expected to do so.

```
rfySimpleStEState :: LinkElement -> Object -> StEState ()
rfySimpleStEState (LnkElem l (LCallSpec _ ctl cdn)) op
   | ctl == CBefore || ctl == CAfter =
        do cd <- lookupCallDescStEState cdn
           reifyStEState (getLinkMO l) (getElementData cd) op
           return ()
```

Listing 3.27: Simple Reifications <Haskell Code>

The `rfyNestStEState` function is shown in Listing 3.28. In addition to the `LinkElement` and the operation object, this function receives a closure function (see Section 3.3.4) which will be executed if the metaobject calls `proceed`. The closure encapsulates the evaluation of all the reifications nested within the reified node (remember that `AROUND` reifications are always represented as a node of the `ReifTree`), or in the case no reification is present it encapsulates the execution of the operation occurrence. The behavior of `rfyNestStEState` is very similar to `rfySimpleStEState` except that the nesting requires pushing and popping the closure from the proceed stack before and after performing the reification, respectively. The proceed closure is built using the function `newPClosure` passing the received closure and the reference to the occurring operation. In addition, `AROUND` reifications expect that the metaobject returns a value which would replace the value returned from the operation occurrence execution. Therefore, the value returned by `reifyStEState` is kept and returned as the result `rfyNestStEState`. The relation between the `rfyNestStEState` and the evaluation of the `proceed` expression is presented in the next section.

The `rfyOrdStEState` function is shown in Listing 3.29. It receives a list of reifications with the corresponding reification function, `rfySimpleStEState` or `rfyNestStEState`[19], and the operation object. Since these functions actually return different types of values, `()` and `Value` respectively, the `rfyOrdStEState` function is defined as polymorphic in the returned value. The monadic map function, `mapM`, is used in

[19]Note that the closure parameter must be bound before invoking `rfyOrdStEState`.

```
rfyNestStEState :: (Object -> StEState Value) -> LinkElement ->
                    Object -> StEState Value
rfyNestStEState clo (LnkElem l (LCallSpec _ CAround cdn)) op =
     do pushClosureStEState (newPClosure clo (getObjectReference op))
        cd <- lookupCallDescStEState cdn
        val <- reifyStEState (getLinkMO l) (getElementData cd) op
        popClosureStEState
        return val
```

Listing 3.28: Nested Reification <Haskell Code>

order to perform the reification sequentially, which produces a list with the result of each reification. The strategy followed after the evaluation of a list of `AROUND` reification is to return the last value. Note that in the case of `rfySimpleStEState` the returned value is always empty.

```
rfyOrdStEState::[(LinkElement, (LinkElement->Object->StEState a))]->
                Object -> StEState a
rfyOrdStEState xr op = do xResult <- mapM reify xr
                          return (last xResult)
                       where
                          reify (r, rfunc) = rfunc r op
```

Listing 3.29: Ordered Reification <Haskell Code>

The closure of an `AROUND` node is generated by the function `genClosureStEState`, shown in Listing 3.30. The function receives the three branches of the node and the operation occurrence closure, and generates the closure using currying. The function processes the three reifications in order using `rfyOrdStEState`: first it processes the list of `BEFORE` reifications (lines 5 and 10), then the list of `AROUND` nodes (lines 6, 11 and 13 to 17) keeping the result in `val` and finally the list of `AFTER` reifications (lines 7 and 12). Finally it returns the `val` value. The function defined in line 11 is responsible for building the appropriate list of `AROUND` reifications for `rfyOrdStEState`. It applies the `decomposeNest` function to all the `ReifTree` of the list of `AROUND` nodes, which generates a list of pairs where the first element is the actual reification in the node and the second element is `rfyNestStEState` function properly bound to the closure. In order to generate the closure for each of the `AROUND` nodes it invokes itself recursively. The `setResultStEState` function, used at line 17 and defined at lines 19 to 23, encapsulates the logic of registering the result of the closure execution in the operation occurrence result field. Thus, it receives a closure and generates (using partial application) another closure which executes it and updates the field. In case the list of `AROUND` nodes is empty, the operation occurrence closure must be executed. The function `chkEmpty`, used in line 6 and defined in lines 13 and 14, create a pair with the `LnkElemNone` link element and a function that executes the closure, `executeClosureStEState`, when the list is empty.

```
genClosureStEState::[LinkElement] -> [ReifTree] -> [LinkElement] ->
                    (Object -> StEState Value) ->
                     Object -> StEState Value
genClosureStEState xbe xar xaf clo op =
   do rfyOrdStEState rxbe op
      val <- rfyOrdStEState (chkEmpty rxar) op
      rfyOrdStEState rxaf op
      return val
   where
      rxbe = map (\r -> (r, rfySimpleStEState)) xbe
      rxar = map decomposeNest xar
      rxaf = map (\r -> (r, rfySimpleStEState)) xaf
      chkEmpty [] = [(LnkElemNone, (executeClosureStEState clo))]
      chkEmpty xar = xar
      decomposeNest (RNode rf xbe xar xaf) =
       (rf, rfyNestStEState
           (setResultStEState (genClosureStEState xbe xar xaf clo)))

setResultStEState::(Object->StEState Value)->Object->StEState Value
setResultStEState closure op =
   do result <- closure op
      updateObjectFieldStEState op "itsResult" result
      return result
```

Listing 3.30: Closure Generation <Haskell Code>

The function responsible for the evaluation of the `ReifTree` is `rfyTreeStEState`, shown in Listing 3.31. The function is divided in two cases which corresponds to: a tree where the root node does not perform any reification and a tree where the root node perform an `AROUND` reification. The former is reified by generating the closure and evaluating it by passing the operation object. The latter also generates the closure, but it nests it in the `AROUND` reification using `rfyNestStEState`.

```
rfyTreeStEState :: (Object -> StEState Value) -> ReifTree ->
                   Object -> StEState Value
rfyTreeStEState clo (RNode LnkElemNone xbe xar xaf) op =
     (genClosureStEState xbe xar xaf clo) op
rfyTreeStEState clo (RNode rf xbe xar xaf) op =
     rfyNestStEState (setResultStEState clo') rf op
     where
        clo' = genClosureStEState xbe xar xaf clo
```

Listing 3.31: `ReifTree` Evaluation <Haskell Code>

Now that the evaluation of the `ReifTree` has been presented, the presentation of the `reifyStEState` function remains and is shown in Listing 3.32. The function receives the metaobject, the call descriptor and the operation object, and performs the actual reification by evaluating the call descriptor's expression block. The expression block is evaluated using the non-reflective expression evaluator, see Section 3.3.7.1. The evaluation is done using the `ObjectAccess` switch condition, where the object is the metaobject at the metalevel. Therefore, when the metaobject is invoked, the evaluation is switched back to the reflective interpreter. Note that before evaluating the expression block, a variable environment is created, containing the bindings for the metaobject and the operation occurrences, using the appropriate names defined in the call descriptor.

```haskell
reifyStEState :: Value -> CallDescObj -> Object -> StEState Value
reifyStEState mo (CDescriptor (Parameter _ on)
                              (Parameter _ mon) eb) op =
      do env <- getVarEnvStEState
         setVarEnvStEState (makeMemberEnv LNone emptyBinds env)
         insertEnvStEState mon mo
         insertEnvStEState on (getObjectReference op)
         val <- nonReflectiveExpr eb
         dropEnvStEState
         return val
      where
         nonReflectiveExpr = evalBlockNRStEState (newObjectSwitch mo)
```

Listing 3.32: Reification Function <Haskell Code>

3.3.7.5 Proceed Evaluation

As mentioned in the previous section, before an `AROUND` metaobject is evaluated, the proceed stack contains the appropriate proceed closure (`rftNestStEState` ensures this) which once evaluated, continues with the reifications that may be nested within the metaobject or if there are not reifications left, it evaluates the original operation occurrence. Listing 3.33 shows the code responsible for the evaluation of a `proceed` expression that modifies the current operation before proceeding.

The evaluation of `proceed` performs the following activities. Firstly, the proceed closure is popped from the proceed stack (line 3). Then the expression block associated to the `proceed` expression is evaluated (lines 5 to 9). This implies the creation of a new variable environment holding the implicit variable `oper` (see Section 3.3.2.6), which has the reference to the current operation occurrence (lines 5 to 7). This reference is obtained from the proceed closure. Once the environment has been created, it evaluates the associated expression block (line 8), which may or may not modify the current operation. Finally, at line 11, the closure gets executed by passing the operation object obtained at line 10. Since the expression block may have changed the

```
 1  case e of
 2   (ProceedE eb) ->
 3       do pc <- topProceedStackStEState
 4          opCpy <- lookupObjectStEState (getPClosureOper pc)
 5          env <- getVarEnvStEState
 6          setVarEnvStEState (makeLocalEnv LNone emptyBinds env)
 7          insertEnvStEState "oper" operationType (getPClosureOper pc)
 8          evalBlockStEState eb
 9          dropEnvStEState
10          op  <- lookupObjectStEState (getPClosureOper pc)
11          val <- (getPClosure pc) op
12          updateObjectStEState opCpy
13          return val
```

Listing 3.33: `Proceed` Expression Evaluation <Haskell Code>

operation occurrence, the metaobject that gets executed by the closure shall see those changes. In Reflex, the changes made to the operation occurrence using `proceed` must only be visible for those metaobjects nested inside the underlying `AROUND` metaobject. Therefore, line 4 makes a copy of the the current operation before evaluating the `proceed`'s expression block. Such a copy is later used at line 12, in order to restore the operation original state once the proceed closure has been evaluated. By doing so, the changes made by the `proceed`'s expression block shall only be visible inside the closure. Note that only the changes made to the operation occurrence object (i.e. field modifications) are indeed preserved. In case the `proceed`'s expression block performs modifications to the state of the elements that conform the operation occurrence (e.g. changing a property of the *this* object), those modifications shall be permanent. Therefore, any metaobject executed after those changes have been made would see them.

3.3.7.6 Limitations

In the current state of the implementation of the Kernel machine, the selectors are evaluated only once, at the beginning of the reification process, generating the collection of links used to build the reification tree and perform the reifications. This behavior imposes some limitations on how links may interact upon the occurrence of an operation, because the modifications performed by the metaobject of a link that is evaluated first may affect the applicability of those links that are evaluated later. As a matter of fact, as a consequence of those modifications some links may *come out* and some links may *come in* of the initial collection of links that apply. For instance, consider the following scenarios:

- There are two links, $\mathtt{L}_{ar}$ with `AROUND` control and $\mathtt{L}_{be}$ with `BEFORE` control, and $\mathtt{L}_{be}$ is nested inside $\mathtt{L}_{ar}$. The `proceed` expression used by the metaobject associated to $\mathtt{L}_{ar}$ modifies some value of the operation occurrence and as a result $\mathtt{L}_{be}$ is not applicable any more.

- There are two links $\mathtt{L}_{be_1}$ and $\mathtt{L}_{be_2}$. The selector of the link $\mathtt{L}_{be_2}$ depends on the value of a property (`aProp`) of the *this* object to match the operation occurrence. Meanwhile the metaobject associated to $\mathtt{L}_{be_1}$ modifies the property `aProp`. Originally $\mathtt{L}_{be_1}$ matches the operation occurrence and $\mathtt{L}_{be_2}$ does not match it because the property does not have the required value, however, once the metaobject of $\mathtt{L}_{be_1}$ is executed it sets the appropriate value.

The current implementation does not consider possible *implicit dependencies* that can be set between the links interacting at a given operation occurrence. For instance, in the first scenario there is an implicit dependency between the links $\mathtt{L}_{ar}$ and $\mathtt{L}_{be}$, in the sense that the changes that $\mathtt{L}_{ar}$ performs determines the applicability of $\mathtt{L}_{be}$. Similarly, in the second scenario where the application of $\mathtt{L}_{be_2}$ depends on the modification done $\mathtt{L}_{be_1}$.

A possible solution to this issues may consist of extending the `selector` construct to provide two different type of conditions. One condition to be evaluated before composition takes place, that is to say, the type of selection provided by the current implementation. And another condition to be evaluated after composition takes place, just before its associated link is actually reified. For instance, consider the extended selector defined in Listing 3.34. This type of selector can be used in the definition of link $\mathtt{L}_{be_2}$ of the second scenario, in order to solve the issue there presented. The selector initially matches all the occurrences of operations inside the `Point` class, and it delays the check of the property value until the reification is about to be performed. Therefore, the associated link will be initially included in the list of selected links, and finally would be applied only if its `postComposition` condition is fulfilled.

```
selector %selPointAt(int #aXVal){
  on Operation #op,
  when
   preComposition  { #op.getWhereClass() == "Point"; }
   postComposition {
      if (#op.getThis() instanceOf Point) then {
        (Point)#op.getThis()).getX() == #aXVal;
      } else {
         False;
      };
   }
}
```

Listing 3.34: Selector Declaration Extension <Kernel Code>

This type of selector allows to define weaker conditions in order to decide if the link must be initially included into the reification tree. Those conditions usually depend on the static information of the operation occurrence, which cannot be altered by metaobjects. Those parts of the condition that indeed depend on dynamic information and whose results may be affected by changes done by other metaobjects, are checked by the `postComposition` condition[20].

In depth study of this limitation and the elaboration of a solution for it are left as further work.

[20]Note that this approach is similar to the one currently used by Reflex, having hooksets that check on the static occurrence of the operations and restrictions that further refine the selection by checking on the dynamic occurrence.

Chapter 4

Modeling the Pointcut and Advice Machine

This chapter presents the development of a machine that characterizes the AspectJ approach to AOP. This machine is used to give an operational semantics to the `Pointcut and Advice` (PA) language, which is an aspect-oriented extension to BASE. The PA language embeds the core constructions of AspectJ's dynamic crosscutting mechanism. The PA and Kernel machines together provide a concise theoretical framework to study how the Reflex AOP kernel can provide semantics to AspectJ. Furthermore, the comparison between the two machines reveals the abstraction gap that exists among them and points out the problems that a compiler, from the PA language to the Kernel language must deal with. In addition, having both machines permits to test such a compiler using the Reflex Sandbox testing environment.

This chapter is organized as follows. Section 4.1 further motivates and presents the subset of the AspectJ language included in the PA language. Section 4.2 presents the concrete and abstract syntax of the PA language. Finally, Section 4.3 presents the design and implementation of the PA machine.

4.1 Introduction

The Aspect Sandbox project (see Section 2.2.2.1) in its quest for a better understanding of the semantics of aspect-oriented languages, has studied the join point model used by AspectJ's dynamic crosscutting mechanism, the so-called *pointcut and advice* model. In order to expose the semantics of this model, the ASB has designed a tiny aspect-oriented language, the PA language, and its correspondent interpreter (in Scheme) that gives an operational semantics to the language. The PA language has sufficient features to faithfully represent the AspectJ approach to AOP. Inspired on this language, an aspect-oriented extension to the BASE language (see Section 3.2.1) has been designed. This extended language, which is also called PA language, incorporates all the aspect-oriented constructs of the ASB's PA language. Those constructs are[1]:

- **Pointcut Definition**. The pointcut designators for most relevant join point kinds have been included, namely `call`, `execution`, `get`, `set`, `instantiation` and `aexecution` (advice execution). In addition, context exposure specification, PCDs combinators, control-flow PCDs and user-defined PCD are supported.
- **Advice Definition**. The three fundamental advice kinds are supported, namely *before*, *after* and *around*. The use of the `proceed` expression (including context modification) within an *around* advice is also supported.

Note that advice precedence[2] is out of the scope of the PA language. Therefore, upon an advice interaction, the order in which advices are executed shall be determined by an ad-hoc strategy. In addition, neither the aspect construct nor pattern-based PCD definition are supported by the PA language.

4.2 Language Syntax

The syntax of the language is presented via the description of its abstract and concrete syntax. The abstract syntax is presented in terms of Haskell's algebraic types. The concrete syntax shall be briefly introduced through an example. For a complete formulation of the PA language syntax see Appendix B.

4.2.1 Abstract Syntax

The language incorporates two new class-level constructions: the pointcut definition and the advice definition. A PA `Program` is a list of definitions with no particular order, see the Listing 4.1.

These two new constructions are presented in the following two sections.

[1]Actually, various of the features here mentioned (e.g. advice-execution join point kind, proceed with context modification), are actually not available in the implementation of the ASB, available online at [asb], but are mentioned in the papers related to the ASB [WKD04].

[2]Actually, in AspectJ it is called aspect precedence.

```
type Program = [Definition]

data Definition = ClassDef ClassDecl        |
                  PointcutDef PointcutDecl |
                  AdviceDef AdviceDecl
```
Listing 4.1: PA Language Programs <Haskell Code>

4.2.1.1 Pointcut Definition

A pointcut definition consists of a name, a list of parameters and a pointcut designator (`PCD`). See Listing 4.2. A pointcut defines a new user-defined pointcut designator. The `PCD` specifies the set of join points that the pointcut selects. The list of parameters are the data that the pointcut exposes from the execution context of its selected join points. The `PCD` also specifies the binding between the selected join points data and the pointcut parameters.

```
data PointcutDecl = Pointcut Name [ParameterDecl] PCD
```
Listing 4.2: Pointcut Declaration <Haskell Code>

The Listing 4.3 shows the primitive PCDs and combinators provided by the language. Each of the first six PCDs match join points of a particular kind, which are *method-call*, *method-execution*, *advice-execution*, *object-instantiation*, *field-get* and *field-set*, respectively. These PCDs are commonly known as kinded PCDs. Each of them imposes restrictions over the specific join point kind. Note that we do no provide support for patterns in pointcut definitions. The `CWithinPCD` and `MWithinPCD` allow to match join points inside a particular class and method, respectively. The `ThisPCD` binds the current executing object to the specified parameter, imposing the restriction that the object must be an instance of the parameter type. The `TargetPCD` is analogous but for the target object. The `ArgsPCD` binds the join point arguments to specified parameters, imposing that each argument must be compatible with the correspondent parameter type.

The `NamedPCD` allows to reference a user-defined PCD, binding each parameter. The types of the correspondent parameters must be equal. The `CFlowPCD` and `CFlowBelow-PCD` are context sensitive PCDs, they impose control flow restrictions over the join point. The `AndPCD`, `OrPCD` and `NotPCD` are logical operators to combine or negate PCDs.

```
data PCD = MCallPCD ReturnType MethodName [Type] |
           MExecPCD ReturnType MethodName [Type] |
           AExecPCD |
           NewPCD ClassName |
           FGetPCD FieldName |
           FSetPCD FieldName |
           CWithinPCD Type |
           MWithinPCD MethodName |
           ThisPCD ParameterName |
           TargetPCD ParameterName |
           ArgsPCD [ParameterName] |
           NamedPCD Name [ParameterName] |
           CFlowPCD PCD |
           CFlowBelowPCD PCD |
           AndPCD PCD PCD |
           OrPCD PCD PCD |
           NotPCD PCD
```

Listing 4.3: Pointcut Designator Language <Haskell Code>

4.2.1.2 Advice Definition

The language provides support for `Before`, `Around` and `After` advices, see Listing 4.4. Each advice must be bound to a pointcut, which can be defined inline or using a named pointcut, specifying at which join points the advice must be executed. The pointcut must also bound all the advice parameters. The expression block of each advice, the so-called *body* of the advice, expresses its behavior.

```
data AdviceDecl = Before [ParameterDecl] PCD ExprBlock |
                  Around ReturnType [ParameterDecl] PCD ExprBlock |
                  After [ParameterDecl] PCD ExprBlock
```

Listing 4.4: Advice Definition <Haskell Code>

`Before` and `After` advices cannot return any value, while `Around` advices can. The return type of an `Around` advice must be compatible with the value returned by each of the join points that the associated pointcut may match. The body of an `Around` may make use of the `proceed` expression[3], which executes the computation of the original join point. The `proceed` expression takes as its arguments the arguments of the underlying `Around` advice, and returns whatever type the `Around` advice is declared to return.

[3]The action language of BASE is extended with the new expression `ProceedCallE [ArgumentExpr]`.

4.2.2 Concrete Syntax Example

The example shown in Listing 4.5 illustrates the pointcut and advice definition concrete syntax. The example is based on the Shape Editor example presented in Section 2.4.2.1. The concrete syntax of the language is similar to the one of the AspectJ language. Pointcuts are defined using the keyword `pointcut` along with its name, parameters and body. The `addShape` pointcut selects the executions of the method `addShape` inside the class `Composite`. The `addPoint` pointcut combines the `addShape` pointcut with the `args` PCD, thus selecting only the executions that add a `Point` and also exposing the `Point` as a pointcut parameter. The `moveXY` pointcut selects all the invocations to the method `moveXY`, exposing their arguments and the target `Shape`. The last pointcut definition selects all the executions of the method `moveXY` within the class `Composite`.

```
pointcut addShape(): execute(void addshape(Shape)) &&
                     within(Composite);

pointcut addPoint(Point #aPoint): addShape() && args(#aPoint);

pointcut moveXY(int #aX, int #aY, Shape #aShape):
         call(void moveXY(int,int)) &&
         args(#aX, #aY) && target(#aShape);

pointcut compositeMoveXY():
         execute(void moveXY(int,int)) && within(Composite);

after(Point #aPoint): addPoint(#aPoint) {
  write "The point" ++ $#aPoint ++ " was added to a composite.";
}

void around(Shape #aShape, int #aX, int #aY):
           moveXY(#aX, #aY, #aShape) && cflow(compositeMoveXY()) {
  write "Moving shape " ++ $#aShape ++ " inside a composite."
  proceed(#aShape, #aX + 10, #aY + 10);
}
```

Listing 4.5: PA Language Concrete Syntax Example <PA Code>

The example also includes the definition of an *after* and an *around* advice, defined using the keywords `after` and `around`, respectively. The *after* advice logs all the points added into a composite shape. It is associated to the pointcut `addPoint`, binding its only parameter. The *around* advice selects all the invocations to `moveXY` which are direct or indirect consequence of an invocation to the `moveXY` method in a composite shape, which is achieved by selecting the join points of the `moveXY` pointcut which are in the control flow of the execution of the pointcut `compositeMoveXY`. Note that the advice returns `void`, since method `moveXY` does so. The body of the advice invokes `proceed`, modifying the parameters of `moveXY` by adding ten to each.

4.3 Machine Implementation

In order to support the new aspect-oriented language constructions, the implementation of the BASE machine is extended by:

- The definition of structures to represent join points, enclosing the information of a point in the execution of the program where advices may be executed. This shall be presented in the next section.
- The implementation of the weaver, responsible for matching the occurring join points against the PCDs associated to advices and performing the actual weaving, i.e. evaluating the advices that match the occurring join point. In addition, the appropriate support for the `proceed` expression must be included. This shall be presented in Section 4.3.3.
- The extension of the machine state, in order to incorporate some structures used to implement the control flow relations and the `proceed` expression, besides of the usual structures used to represent the defined pointcut and advices. This shall be presented in Section 4.3.2.

4.3.1 Join Point Representation

Listing 4.6 shows the data type `JoinPoint` which represents a dynamic join point. There are six different kinds of join points:

- `CallJP` representing a method call, enclosing: the *this* and target object references, the `MethodDecl` (see Section 3.2.1.1) representing the method being called and the list of arguments to the call.
- `MExecJP` representing a method execution, enclosing: the *this* object reference, the `MethodDecl` of the method being executed and the arguments to the method execution.
- `FGetJP` representing a access to a field, enclosing: the *this* object reference[4] and the field name.
- `FSetJP` representing the modification of a field, enclosing: the *this* object reference, the name of the field and the new value for the field.

[4]Remember that fields are only visible from the object that contains them.

- `InstJP` representing the instantiation of an object, enclosing: the *this* object reference and the name of the class being instantiated. Note that this join point does not represent array instantiation.

- `AExecJP` representing the execution of an advice, enclosing the context exposed parameters passed to the advice.

```
data JoinPoint = CallJP This Target MethodDecl [Value] |
                 MExecJP This MethodDecl [Value] |
                 FGetJP This FieldName |
                 FSetJP This FieldName Value |
                 InstJP This ClassName |
                 AExecJP [Value]
```

Listing 4.6: Join Point Representation <Haskell Code>

The types `This` and `Target` are synonyms of the type `Value` (see Section 3.2.2.1). Those values hold the reference to the *this* and target objects, respectively.

4.3.2 State Representation

The state of the PA machine extends the state of the BASE machine with two new environments, one holding the user-defined pointcuts and the other holding the defined advices. The `State` data type for the PA machine is shown in Listing 4.7.

```
data Environments = CEnv VariableEnv ClassEnv AdviceRep PointcutRep
                         JoinPointStack ProceedStack

data Stores = CSto ObjectStore

data State = CState Environments Stores Output
```

Listing 4.7: PA Machine State <Haskell Code>

In addition, the state is extended with the join point stack and the proceed stack. The former is similar to a stack of method invocations, but keeping track of the join points that have occurred in the control flow of the current join point; used in order to match `cflow` and `cflowbelow` PCDs. As pointed out in [MKD02], using stack of join points in order to mach control flow restrictions offers an straightforward but inefficient implementation for matching context sensitive pointcuts, however, since we are more interested in obtaining a clear implementation than a efficient one, we use the straightforward approach.

The proceed stack holds the closure function used to execute the original join point (or the remaining advices) upon the evaluation of a `proceed` expression inside an *around* advice. Closures are represented by the `ProceedClosure` data type shown in Listing 4.8 which holds a function that receives the arguments passed to the `proceed` expression and returns the value resulting from executing the original join point. Since *around* advices may be executed nested inside others *around* advices or may be acting over a join point inside the control flow of another *around* advice, a stack structure is used in order to hold the coexisting closures, see the `ProceedStack` in Listing 4.8.

```
data ProceedClosure = PClosure ([Value] -> StEState Value)

type ProceedStack = Stack ProceedClosure
```
Listing 4.8: Proceed Stack <Haskell Code>

The variable environment is also extended by incorporating location information (like for the Kernel machine, see Section 3.3.4). Therefore, the variable environment created for the method execution holds the `MethodDecl` data type of the executing method. This is used in order to match the `within` and `withincode` PCDs.

4.3.3 Weaver Implementation

The program interpretation function of the PA machine acts as a weaver, it first separate the advices and pointcuts declarations from the rest of the program, i.e. an ordinary BASE program, and then proceeds with the normal interpretation of the BASE program, simultaneously checking when the advices must be executed and executing them. The program interpreter for the BASE machine, shown in Section 3.2.2.2, is extended in order to load the advice and pointcut declarations into the corresponding environments, `AdviceRep` and `PointcutRep` respectively, see Section 4.3.2. The rest of the program is loaded and interpreted as before. In addition, the pieces of the expression interpreter function where the supported join points occur (except for the advice execution) are modified in order to create the appropriate join point structure and to check whether any advice declaration matches the join point, in which case they are executed.

The `CallJP` and `MExecJP` join points require the modification of the interpretation code for the `MethodCallE` and `SuperMethodCallE` expressions. The `FGetJP`, `FSetJP` and `InstJP` join points require the modification of the interpretation code for `FieldGetE`, `FieldSetE` and `NewObjectE` expressions, respectively. The `AExecJP` does not require the modification of any piece of the expression interpreter, it is handled apart and shall be presented in Section 4.3.5. In order to illustrate how the expression interpreter is modified, Listing 4.9 shows the piece of interpreter corresponding to the `MethodCallE` expression, enclosing the `CallJP` and `MExecJP` join points[5].

```
 1  case e of
 2   (MethodCallE oe n args) ->
 3      do ref <- evalExprStEState oe
 4         obj <- lookupObjectStEState ref
 5         cls <- lookupClassStEState (getObjectClass obj)
 6         mth <- mthLookup cls mn
 7         vals <- mapM (\e -> evalExprStEState e) args
 8         this <- lookupEnvValueStEState "this" ValueNull
 9         evalStEState (CallJP this ref mth vals) mcClosure
10      where
11         mcClosure (CallJP _ ref mth vals) =
12           evalStEState (MExecJP ref mth vals) meClosure
13         meClosure (MExecJP ref mth@(Method _ rt _ _ eb) vals) =
14           do obj <- lookupObjectStEState ref
15              createMthCallEnvStEState mth vals ref
16              val <- evalBlockStEState eb
17              dropEnvStEState
18              return val
```

Listing 4.9: Expression Interpreter <Haskell Code>

The modification introduced by the weaver to the expression evaluator follows a similar structure of that used for the interception code in the Kernel machine, see Section 3.3.5.1. It creates the join point structure for the occurring join point, in this case `CallJP` (line 9), and passes it to the function `evalStEState` (line 9) which is responsible for the evaluation of the advices that match the join point (if any). The `evalStEState` function also receives a closure function (`mcClosure`) which, once evaluated, continues with the original execution. In this case, continuing with the original execution means creating the structure for the `MExecJP` join point (line 12) and passing it to the `evalStEState` which starts again the matching and evaluation of the advices (lines 12). The second execution of `evalStEState` receives the `meClosure` closure function which, once executed, performs the method execution (lines 13 to 18). Note that both `mcClosure` and `meClosure` take the values from the received join point to create the `MExecJP` and execute the method respectively, thus assimilating any changes made to the join point by *around* advices.

The function `evalStEState`, shown in Listing 4.10, performs two activities: collecting all the advices that match the received join point and performing the weaving of those advices with the closure received as a parameter. The advices are collected by the function `adviceMatchStEState` at line 3, which iterates over all the defined

[5]Note that the `SuperMethodCallE` expression also comprehends those join points. It requires analogous modifications.

advices, testing if its associated pointcut matches the join point. The function, besides collecting the advices, also groups them by advice kind. See Section 4.3.4 for a detailed explanation of how the pointcut matching is done. The advice weaving, see lines 5 to 9, involves coordinating the evaluation of the matched advices with the evaluation of the original join point. Since more than one advice may have matched the join point and the language does not provide means to specify the order (e.g. like aspect precedence in AspectJ) in which they must be evaluated, a default strategy is provided. The strategy[6] executes the *around* advices first, then the *before* advices, followed by the original join point and finally the *after* advices. The *around* advices are nested one inside the other so that the advice lexically declared later is nested inside the advices declared earlier. Besides, the last *around* advice to be evaluated nests the evaluation of the *before* advices, the current join point and the *after* advices. The advice weaving is divided in three cases:

- No advice matches the join point (line 6), thus advice weaving is not required. In this case, the closure function is executed in order to continue with the original computation.

- Only *before* and/or *after* advices match the join point (line 7). In this case, the `weaveBeAf` function is executed which first evaluates all the *before* advices, then the closure and finally the *after* advices. The advices are evaluated using the `evalAdviceStEState` explained on Section 4.3.5. Note that the *before* and *after* advices do not return any value, therefore the result of the closure evaluation is returned as the result of the overall computation.

- There are one or more *around* advices that match the join point (line 8), along with other *before* and/or *after* advices. In this case, the `evalAroundStEState` function is used to weave all the *around* advices. It receives the list of *around* advices, the join point and a closure function to be executed when the last *around* advice executes `proceed`. The closure function is built using the `weaveBeAf` function, thus weaving the *before* and *after* advices.

Note that `evalStEState` is also responsible of keeping updated the join point stack. It does so by pushing the join point into the stack before executing the weaved join point and popping it after its execution ends. Also note that advice matching only occurs once, when the `evalStEState` is invoked.

[6]It is the same strategy used in the Pointcut and Advice model implementation of the ASB, see [asb].

```haskell
evalStEState::JoinPoint->(JoinPoint->StEState Value)->StEState Value
evalStEState jp closure =
    do (mBe, mAr, mAf) <- adviceMatchStEState jp
       pushJPStackStEState jp
       val <- case (mBe, mAr, mAf) of
                ([],  [],  [])  -> closure jp
                (xBe, [],  xAf) -> weaveBeAf xBe xAf jp
                (xBe, xAr, xAf) ->
                       evalAroundsStEState xAr jp (weaveBeAf xBe xAf)
       popJPStackStEState
       return val
    where
       weaveBeAf xBe xAf jp =
            do foldM (\_ -> evalAdviceStEState jp) () xBe
               val <- closure jp
               foldM (\_ -> evalAdviceStEState jp) () xAf
               return val
```

Listing 4.10: Join Point Evaluation <Haskell Code>

4.3.4 Pointcut Match

Matching a pointcut is a twofold activity, it requires both testing if it matches the join point and binding its parameters to the join point context. A parameter can be bound to the *this* object, the target object or one of the arguments of either: the current join point (i.e. the one being matched) or a join point in its control flow. Therefore binding a parameter requires determining which value is associated to each pointcut parameter, however the fact that the language supports the `proceed` expression with context modification complicates performing those bindings for two reasons:

- An *around* advice may nest the execution of the advice associated to the pointcut being matched and may modify, using `proceed`, the context of the current join point. Since those changes must be visible for the nested advice, the real binding of the parameters to the values must be delayed until the nested advice is about to execute. Note that a pointcut is matched only once for each join point and is matched before any advice gets executed.

- The `proceed` expression takes as its arguments, the arguments to the underlying *around* advice, which in turn are bound to values in the context of the current join point. Therefore, once the `proceed` expression is evaluated, the associated closure must be capable of modifying the context values corresponding to each argument. Performing such a modification requires some sort of mapping between each argument and the corresponding context value of the current join point. Note that this mapping can only be determined from the pointcut associated to the *around* advice.

In order to overcome these two difficulties, the binding shall not be expressed as a pair associating a parameter name to a value, but as a triplet (see the element of the list `CtxBind` on Listing 4.11) associating a parameter name with two functions: one that, given the current join point, returns the bound value (`CtxGetter`) and another that, given the current join point and the new value, returns the modified join point (`CtxGetter`). Note that, as explained in [WKD04], modifying an advice parameter corresponding to a context value of a join point other that the current join point, i.e. a join point in the control flow, does not have any clear meaning, since that refers to an expression that has already been evaluated. In response to this issue, only the parameters corresponding to the current join point can be modified, any attempt to modify another parameter is ignored. This explains why the `CtxSetter` only receives the current join point. The `CtxSetter` function corresponding to a parameter bound to a value in the control flow of the current join point always returns the same join point. In addition, note that `CtxGetter` function only receives the current join point, because the values obtained from a control flow relation can be bound at pointcut matching time, since they cannot be changed by a `proceed` expression.

```
type CtxGetter = JoinPoint -> Value

type CtxSetter = JoinPoint -> Value -> JoinPoint

type CtxBind = [(ParameterName, CtxGetter, CtxSetter)]
```
Listing 4.11: Context Exposure Bindings <Haskell Code>

The pointcut matching is performed by the function `pcMatchStEState`, partially shown in Listings 4.12 and 4.13, which receives the current join point, the PCD to be matched against and the list of parameters of the pointcut that encloses the PCD. In addition, it receives a boolean flag which states if the PCD is inside a `cflow` or `cflowbelow` PCD. The function is simply a case-based test to see whether the given PCD matches the join point. If not, it returns `False` and an empty set of bindings, otherwise it returns `True` and a list with the bindings generated by the PCD. The first two cases in Listing 4.12 illustrate how the `CallJP` join point is matched by `MCallPCD` and `TargetPCD`. The former involves testing if the return type, method name and parameter types are equal[7]. If they are, the matching is successful and no bindings are generated. Note that `MCallPCD` only matches join points of method-call kind, thus this case is the only one involving `MCallPCD`. The latter involves obtaining the type of the parameter referenced by `TargetPCD` and testing if the type is compatible with

[7]Note that the AspectJ's wildcard `+` [aspa] is not supported by the PA language, thus subtypes are not matched by the `call` PCD.

the target object. If it is, the matching is successful and it must generate the binding for the parameter. If the `TargetPCD` is not inside a control flow PCD, the binding is generated with two functions: `ctxGetTarget` that given a join point returns its target object reference and `ctxSetTarget` that, given a join point and target object reference, returns the modified join point. On the opposite, if the `TargetPCD` is inside a control flow PCD, the target object reference never changes, therefore the value of the target object can be bound in advance by using a constant function (`ctxGetConst`) that always returns it. Meanwhile, the setter function `ctxSetID` is the identity, i.e. always returns the received join point without modifications. The last two cases in Listing 4.12 illustrate how the `CWithinPCD` is matched. Since advice executions are not lexically enclosed within any class, the first case states that `CWithinPCD` never matches them. The second case contemplates the rest of the join points, which can be lexically enclosed within a class. It is implemented by using the location information, available at the variable environment (see Section 4.3.2), to obtain the class in which the join point lexically occurs. This behavior is implemented by the function `pcMatchCWithinStEState` not shown here.

```
pcMatchStEState :: JoinPoint -> PCD -> [ParameterDecl] -> Bool ->
                   StEState (Bool, CtxBind)
pcMatchStEState jp pcd xp inCFlow =
 case (jp, pcd) of
   (CallJP _ _ (Method _ rt mn xp _) _, MCallPCD rt' mn' xt) ->
     return ((mn == mn') && (rt == rt') && ((getTypes xp) == xt),
             emptyCtxBinds)
   (CallJP _ ta _ _, TargetPCD pn) ->
     do (Parameter pt _) <- return (findParam pn xp)
        match <- testCompatTypeStEState pt ta
        case (match, inCFlow) of
         (False, _) -> return (False, emptyCtxBinds)
         (_, False) -> return (True, [(pn, ctxGetTarget, ctxSetTarget)])
         (_, True)  -> return (True, [(pn, ctxGetConst ta, ctxSetID)])
   (AExecJP _, CWithinPCD _) -> return (False, emptyCtxBinds)
   (_, CWithinPCD cls) ->
     do s <- getVarEnvStEState
        pcMatchCWithinStEState s cls
   (..., ...) -> ...
```

Listing 4.12: Pointcut Matching <Haskell Code>

The first two cases shown in Listing 4.13 illustrate how the control-flow PCDs are matched. In order to test the control-flow restriction they get the join point stack, from the state, and iterate over it looking for a join point that matches its enclosed PCD using the function `pcMatchCFlowStEState`. Note that `CFlowPCD` and `CFlowBelowPCD` are almost equal. The only difference is that `CFlowPCD` considers the current join point as a candidate to be matched by its enclosed PCD. Function pcMatchCFlowStEState (not shown here) uses pcMatchStEState in order to test each of the join points in the given stack, setting the control flow flag to True. If a join point is matched, it returns the value same as pcMatchStEState, i.e. a pair with the True value and the list of generated bindings, otherwise it returns False and an empty list of bindings.

```
case (jp, pcd) of
  (jp, CFlowPCD pcd) ->
    do s <- getJPStackStEState
       pcMatchCFlowStEState (pushS s jp) pcd xp
  (_, CFlowBelowPCD pcd) ->
    do s <- getJPStackStEState
       pcMatchCFlowStEState s pcd xp
  (jp, AndPCD left right) ->
    do (mL, bndL) <- pcMatchStEState jp left xp inCFlow
       (mR, bndR) <- pcMatchStEState jp right xp inCFlow
       if (mL && mR)
          then return (True, bndL ++ bndR)
          else return (False, emptyCtxBinds)
  (jp, OrPCD left right) ->s
     do (m, bnd) <- pcMatchStEState jp left xp cf
        if m
           then return (True, bnd)
           else pcMatchStEState jp right xp inCFlow
  (jp, NotPCD pcd) ->
     do (m, []) <- pcMatchStEState jp pcd xp cf
        return (!m, emptyCtxBinds)
```

Listing 4.13: Pointcut Matching (continued)

The last three cases of Listing 4.13 illustrate how the PCD combinator operators are matched. The AndPCD requires matching the two enclosed PCDs, in the case that both match the current join point, the returned list of bindings is the concatenation of the bindings generated by each of them. The OrPCD first tests the left PCD, if it successfully matches the join point it returns that the matching is successful along with the list of bindings generated by the left PCD, otherwise it returns the result of the matching of the right PCD. The NotPCD tests its enclosed PCD, which must always return an empty list of bindings, since, as mentioned in [aspa], negating a PCD which binds one or more parameters does not make any sense. The NotPCD returns the negation of the testing result, along with a empty list of bindings.

The weaver function adviceMatchStEState, introduced in Section 4.3.3, is the one that uses the pcMatchStEState to test if the pointcut associated to each advice matches the occurring join point. The result of the adviceMatchStEState function is a triplet with three lists, one for each advice kind, each containing pairs of the form (AdviceDecl, CtxBind). Each pair contains the AdviceDecl of a matched advice along with the list of bindings generated by its associated pointcut. Since the list of bindings returned by the `pcMatchStEState` may not follow the order defined by the list of parameters of the pointcut, `adviceMatchStEState` sorts the list before returning it. Next section presents how these advices are actually weaved and explains how the context information is used to obtain the parameters expected by the advice.

4.3.5 Advice Evaluation

The function `evalAdviceStEState`, shown in Listing 4.14, is in charge of the evaluation of *before* and *after* advices. The function receives the current join point (i.e. the one that causes the evaluation of the advice) and a pair with: the advice to evaluate and the parameter bindings for the advice. The evaluation of an advice, including the *around* advice (shown in Listing 4.15), entails the creation of the `AExecJP` join point structure before actually evaluating the advice and invoking the `evalStEState` in order to weave the new join point occurrence. For `evalAdviceStEState`, this occurs at line 3, the `aeClosure` function that `evalStEState` receives is the function that actually evaluates the advice.

```
1  evalAdviceStEState::JoinPoint -> (AdviceDecl, CtxBind) -> StEState ()
2  evalAdviceStEState jp (ad, bn) =
3    do evalStEState (AExecJP (calcBindingValues jp bn)) aeClosure
4       return ()
5    where
6       aeClosure (AExecJP xv) =
7         do  env <- getVarEnvStEState
8             setVarEnvStEState (makeMemberEnv LNone emptyBinds env)
9             foldM bindParameter () (zip (getAdviceParameters ad) xv)
10            val <- evalBlockStEState (getAdviceBody ad)
11            dropEnvStEState
12            return val
```

Listing 4.14: *Before* and *After* Advice Evaluation <Haskell Code>

The creation of the `AExecJP` join point requires calculating the values bound to each advice parameter (i.e. its arguments). Remember that the `CtxBind` provides only a getter function capable of obtaining the value, not the real value. The function `calcBindingValues` calculates those values by applying each of the getter functions to the current join point. If the `aeClosure` is finally evaluated, remember that an *around* advice may replace it and may not proceed, it evaluates the advice by: creating a variable environment containing the advice arguments (lines 7 to 9), evaluating its expression block (line 10) and returning the result of the evaluation (only required to fulfill the type of the closure, since it expects a `Value` to be returned). Note that any changes made to the advice arguments (i.e. by an *around* advice using `proceed`) are assimilated by `aeClosure`, because it takes those arguments from the received join point. Since the *before* and *after* advices do not return any value, the `evalAdviceStEState` always returns an empty result (line 4).

The *around* advices are evaluated by the function `evalAroundsStEState` shown in Listing 4.15. Conversely to `evalAdviceStEState` which only evaluates one advice, `evalAroundsStEState` evaluates the list of all *around* advices that match the current join point, nesting them one inside the other, like a chain, as explained in Section 4.3.3. It receives the list of advices with its corresponding parameter bindings, the current join point and the closure function to be nested by the last *around* advice in the chain. The function chain of advices is constructed recursively, when the advice at the head of the list is evaluated (by the equation at line 4), a proceed closure is generated for it (lines 16 to 18) which, once evaluated, proceeds with the evaluation of the tail of the list, thus continuing recursively until no more *around* advices are left, so executing the closure (line 3).

```
 1  evalAroundsStEState::[(AdviceDecl, CtxBind)] -> JoinPoint ->
 2                       (JoinPoint -> StEState Value) -> StEState Value
 3  evalAroundsStEState [] jp closure = closure jp
 4  evalAroundsStEState (((Around _ xp _ eb), bn):xAr) jp closure =
 5    evalStEState (AExecJP (calcBindingValues jp bn)) aeClosure
 6    where
 7      aeClosure (AExecJP xv) =
 8        do env <- getVarEnvStEState
 9           setVarEnvStEState (makeMemberEnv LNone emptyBinds env)
10           foldM bindParameter () (zip xp xv)
11           pushPStackStEState (newPClosure (proceedClosure jp))
12           val <- evalBlockStEState eb
13           popPStackStEState
14           dropEnvStEState
15           return val
16      proceedClosure jp xv =
17        do val <- evalAroundsStEState xAr (applyArgs jp xv) closure
18           return val
19      applyArgs jp xv =
20        foldl (\jp (v,setter) -> setter jp v ) jp
21              (zip xv (map (\(_,_,setter) -> setter) bn))
```

Listing 4.15: *Around* Advices Evaluation <Haskell Code>

As for the other advice kinds, before actually evaluating the advice the `AExecJP` is generated and passed to `evalStEState`, along with the `aeClosure`. Once the `aeClosure` is evaluated, it evaluates the advice almost in the same way as `evalAdviceStEState` does, except that the proceed closure must be pushed to the proceed stack (line 11) before evaluating the body of the advice and popped after the evaluation ends (line 13). The proceed closure is generated by the function `proceedClosure` passing to it the current join point (using partial application). Once the proceed closure is evaluated, as a consequence of the evaluation of `proceed` in the advice body, `proceedClosure` receives the `proceed` arguments, thus it gets evaluated. Before proceeding with the evaluation of the tail of the list, it must apply the changes that the proceed may have done over the current join point. This is done by the function `applyArgs` which, for each argument, applies the corresponding setter function to the current join point, therefore obtaining the modified join point.

The evaluation of the `ProceedCallE` expression, done by the expression evaluator (`evalExprStEState`), simply requires: obtaining the proceed closure from the stack, evaluating the arguments expressions, i.e. the `proceed` arguments, and evaluating the closure.

Chapter 5

Compiling to the Kernel Machine

This chapter presents the first and most fundamental step in the validation of the support for the AspectJ language by the Reflex AOP kernel: providing semantics for a subset of AspectJ's dynamic crosscutting mechanism. To that end, a theoretical case study is presented, describing the development of a compiler from the PA language to the Kernel language. This study supplies the foundations for understanding how the Reflex AOP kernel can be used to provide semantics to AspectJ. In addition, a practical case study is presented, which involves the implementation of a compiler in Java that transforms AspectJ programs into Reflex programs. Furthermore, micro-benchmarks are provided to show that reasonable efficiency levels can be achieved. These case studies together constitute the first serious empirical validation of the aptitudes of the Reflex model to be an AOP kernel for Java.

This chapter is organized as follows. Section 5.1 explains what it means to provide semantics to an aspect-oriented language using reflection. Also, it briefly describes the differences between the two compilers. The compiler implemented in the Reflex Sandbox (RSB) is presented in Sections 5.2 and 5.3. The former discusses, based on examples, how the constructions in the Kernel language can be used to provide semantics to those in the PA language. The latter presents the actual implementation of the compiler in Haskell. Section 5.4 presents the compiler from AspectJ to Reflex by explaining the main differences with the one implemented in the RSB. In addition, that section also explains how some advanced issues, like composition, are addressed by the compiler. Finally, Section 5.5 presents the Java implementation of this compiler, as a plugin for the Reflex's plugin architecture. Also, it presents benchmarks validating the implementation and the design of a reflective API for AspectJ.

5.1 Introduction

Aspect-Oriented Programming (AOP) is deeply connected with the work of computational reflection and metaobject protocols, as pointed out in [KLM+97] and studied in [KLLH03]. A reflective system provides a base language and one (or more) meta-languages that provide control over the semantics of the base language. For instance, in the Kernel machine the base- and meta- languages are the BASE language and the language composed by the reflective extensions introduced through the Kernel language, respectively. Those reflective extensions provide views of the computation that no one BASE language component could ever see, e.g. the operation occurrences scattered among various base level objects. As pointed out in [TN04a], aspect-oriented (AO) techniques offer principled ways to do reflection, hiding from the user most of the complexity associated to the implementation of reflective systems by providing high-level languages in order to express the semantic alterations over the base language. For instance, in the PA machine the AO mechanisms introduced by the PA language (i.e. the aspect language), allow to express semantic alterations over programs written in the BASE language. Note that the distinction made for the Kernel machine, between base- and meta- languages, is also valid for Reflex, where Java is the base language and Reflex's reflective extensions compose the meta-language. Similarly, the AO mechanisms of AspectJ compose the aspect language affecting the semantics of programs written in Java.

In this context, the two compilation processes here described shall transform an AO program into a reflective program as follows. The base level portion of the program is left intact. Meanwhile, the semantic alterations defined via the aspect language portion of the program are expressed in terms of the reflective mechanisms offered by the meta-language. Therefore, the compilers are functions of the following form:

$$compiler_{PA-Kernel} : Program_{PA} \longrightarrow Program_{Kernel}$$

$$compiler_{AspectJ-Reflex} : Program_{AspectJ} \longrightarrow Program_{Reflex}$$

The main difference between the two compilers, besides the obvious difference in the origin and target languages, is that the $compiler_{PA-Kernel}$ works with interpreted languages, while the $compiler_{AspectJ-Reflex}$ works with compiled languages. Reflex for Java is specially tailored for a compiled environment; hence it provides constructs to represent both static properties, fixed at compilation time[1], and dynamic properties whose evaluation is delayed until runtime. The $compiler_{AspectJ-Reflex}$ takes full advantage of these constructs to represents AspectJ's shadows and residues (see Section 2.6.3.1), in order to achieve efficiency. The PA and Kernel languages do not

[1]Actually, in the case of Reflex this occurs at load-time (see Section 2.4). However the same distinction applies.

distinguish between static and dynamic properties, everything is performed at runtime. Consequently, the *compiler*$_{PA-Kernel}$ does not have to differentiate between these two property types.

In addition, there is another difference in terms of the features of the AspectJ language supported by each compiler. Since the Kernel machine has been designed to only support part of the Reflex model constructs, not every PA program can be compiled into a Kernel program. The *compiler*$_{AspectJ-Reflex}$ does consider all the constructs of the Reflex model. Therefore, it supports the compilation of all the AO constructs defined in the PA language, except for some PCDs that were not included in order to simplify the implementation. In addition, that compiler supports some interesting features of the AspectJ language, like aspect composition and join point reflective information.

5.2 Conceptual Overview

At first glance, there are several similarities between the concepts in the two machines, from which an initial correspondence can be grasped: join points shall correspond to operation occurrences, pointcuts shall correspond to hooksets and advices shall correspond to metaobjects. The operational behavior of both machines is essentially the same, once a join point occurs (resp. an operation occurrence) the pointcuts (resp. hooksets) match it and perform the behavior defined by the advices (resp. metaobjects). Since the primary subject of matter of an advice is affecting the execution semantics of another program, it clearly fits into a metaobject. Note that for each join point kind supported by the PA machine, the Kernel machine must provide an operation capable of reifying it.

In order to get a clear intuition of the conceptual correspondence of the pointcut and advice constructions, the next two sections shall further analyze them. The presentation is informal, example-based and does not enter into details. The example used is the Shape Editor System, presented in Section 2.4.2.1.

5.2.1 Pointcuts

The pointcuts `move` and `movePoint`, shown in Listing 5.1, are implemented using a primitive hookset with a selector defined over the `MsgReceive` operation. The selection predicate simply selects the operation occurrences that match the restrictions imposed by the pointcut designators, based on the information exposed by the `MsgReceive` class (see Section 3.3.2.1). For instance, the selector for the `movePoint` pointcut requires checking that the operation occurrence has the specified method signature (i.e. the `returnType` is `void`, the `method` is `moveXY` and `methodParTypes` contains two `int` parameters) and that it is occurring within the `Point` class (i.e. `whereClass` is `Point`).

```
pointcut move(): execution(void moveXY(int, int))

pointcut movePoint(): move() && within(Point)

pointcut moveSinglePoint(): movePoint() && !cflowbelow(move())

pointcut moveSinglePointArgs(int x, int y):
             moveSinglePoint() && args(x, y)
```

Listing 5.1: PA Pointcut Examples <PA Code>

The pointcut designators related to control flow, `cflow` and `cflowbelow`, allow picking out join points based on whether they are in a particular control flow relationship with other join points. For example, the pointcut `moveSinglePoint` imposes an additional restriction to the `movePoint` pointcut, which is that selected join points must not be within the control flow of the `move` PCD, in other words, stand-alone `Points` not included inside a `Line` or `Composite` shape. Note that this pointcut definition implies the definition of two nested pointcuts: the pointcut inside the `cflowbelow` (`move`) and the pointcut affected by a control flow restriction (`movePoint`).

To be able to determine whether `movePoint` is matched in the control flow of `move`, we first need to expose the control flow information of `move`: this is done using the notion of *event collectors* [TN04b]. Event collectors gather execution events to expose parts of a program execution (nesting, sequences, etc.), under any structure (counter, stack, tree, DAGs, graphs, etc.), for dynamic introspection. In particular they are used to expose control flow information. Indeed, event collectors are just like metaobjects, except that their purpose is only to *expose* elements of program execution, rather than to *affect* program execution. Consequently, mapping a control flow PCD implies two separate tasks:

- defining a separate link for an event collector exposing control flow information of the pointcut passed as argument to `cflowbelow`;
- defining a control flow condition that checks the exposed control flow information. This check is included in the selection predicate of the hookset that models the `moveSinglePoint` pointcut.

The pointcuts that expose context information require, besides of a hookset selecting the appropriate operation occurrences, a call descriptor which gathers the appropriate context information and passes it to the metaobject during the reification.

For example, the pointcut `moveSinglePointArgs` exposes the two parameters of the method `moveXY`, thus, there is an associated call descriptor which gets the two parameter values from the operation occurrence object and passes them to the associated metaobject during reification.

5.2.2 Advices

In the PA language, the advice construction has the dual responsibility of defining the advice behavior and binding it to a pointcut definition. Meanwhile, in the Kernel language those two responsibilities correspond to a metaobject definition and a link definition, respectively. For instance, consider the advice definition in Listing 5.2. The advice logs all the movements of stand-alone points, including the two parameters exposed by the pointcut `moveSinglePointArgs`. The advice behavior is encapsulated inside a method of a metaobject class, which receives the same parameters of the advice definition. The binding between an instance of the metaobject class and the hookset that represents `moveSinglePointArgs` is done through a link. The link associates the hookset with `AFTER` control value (corresponding to the advice kind) and a call descriptor, which in this example must get the two parameters from the `MsgReceive` operation occurrence and invoke the advice method at the metaobject. The link also includes the metaobject instantiation block (see Section 3.3.2.4), which simply instantiates the metaobject.

```
after(int x, int y): moveSinglePointArgs(x, y) {
  write "Point moved: " ++ x ++ ", " ++ y;
}
```

Listing 5.2: PA Advice Example <PA Code>

Since `moveSinglePointArgs` is affected by a control flow restriction over the pointcut `move`, an additional link is required. This link binds the hookset corresponding to the pointcut `move` to an event collector metaobject. Such a metaobject is a simple counter that keeps track of entries and exits in the hookset. Therefore the link would associate the hookset twice, one with control `BEFORE` invoking the `enter` method at the metaobject and one with control `AFTER` invoking the `exit` method.

5.3 Compilation

The compiler is implemented as a multi-staged process, which is illustrated in Figure 5.1. The boxes in the figure represent the different stages of the process. The stages are presented according to their order of execution, i.e. stages at the left of the figure are executed before stages at the right. The dashed strip at the bottom of the figure shows which stage produces each part of the resultant program. The compilation function is implemented using the `StE` monad (see Section A.2) in order to propagate a state among the stages. The state is used to store the different parts of the resultant Kernel program, as they are generated, and also to store intermediate structures used by the compilation process.

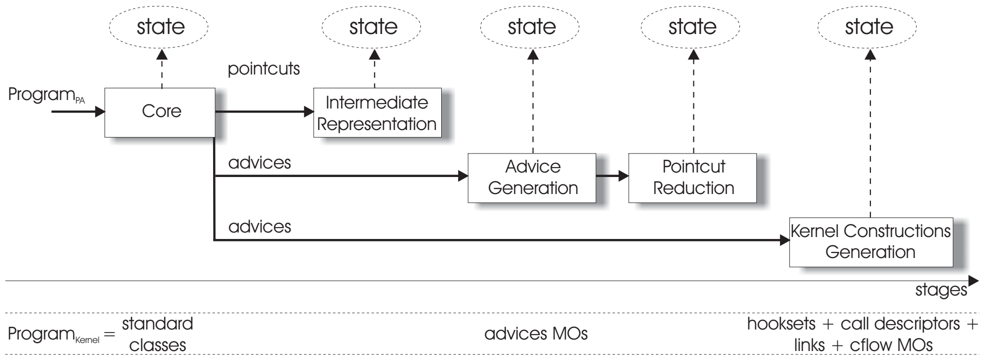

state
state
state
state
state
pointcuts
Program_PA
Core
Intermediate Representation
advices
Advice Generation
Pointcut Reduction
advices
Kernel Constructions Generation
stages
Program_Kernel = standard classes
advices MOs
hooksets + call descriptors + links + cflow MOs

Figure 5.1: Compilation process

The stages involved are:

Compiler Core: it is responsible for coordinating all the stages of the compilation process, converting all the standard classes in the PA program into classes of the Kernel program and assembling the the resultant Kernel program, once all the stages have been performed. The standard classes do not require any special transformation but converting from the PA language AST to the Kernel language AST.

Pointcut Intermediate Representation: it transforms each declared pointcut into an intermediate structure which expresses the semantics of the pointcut in terms of elements closer to the Kernel language, e.g. operation types, restrictions over the fields of those operations and field reification requirements for those operations. This intermediate structure conforms a common schema to express both the predefined PCDs of the PA language and the user-defined PCDs, over which the rest of the stages base their work. The state stores those intermediate structures for later use. The construction of these intermediate structures is explained in detail in Section 5.3.1.1.

Advice Generation: it is responsible for generating the metaobject representing each advice and performing the reduction of the pointcuts associated to the advices. The generated metaobject and the reduced pointcut are stored in the state. Section 5.3.2 gives a detailed description of how the metaobjects are generated.

Pointcut Reduction: it is responsible for transforming the intermediate structure that represents a pointcut into an semantically equivalent structure, which is more suitable for the generation of the hooksets and call descriptors used to represent the pointcut. The motivation for this stage relies on the fact that pointcut predicates intermix the join point matching predicate and the context exposure requirements (so does the initial pointcut intermediate structure), which makes it difficult to extract the hooksets and call descriptors definitions from them. See next section for deeper analysis of the motivations for this stage and the detailed description of the reduction algorithm.

Kernel Constructions Generation: it is responsible for generating the links required for each advice, along with the required hooksets and call descriptors. These Kernel constructions are generated based on the advice definition and the reduced intermediate structure of the pointcut bound to the advice. In addition, for each pointcut that has a control flow restriction, it also generates the appropriate event collector metaobject along with its link, hooksets and call descriptors. See Section 5.3.3 for a detailed description.

5.3.1 Pointcuts

The pointcut construction plays a dual role in the PA language by specifying both the matching predicate and the context exposure requirements. Conversely, in the Kernel language these two roles are handled independently by the hooksets and call descriptors, respectively. As mentioned in Section 5.2.1, in order to generate the hookset(s) and call descriptor(s) from a pointcut definition, the compiler must decompose it. Such a decomposition implies, on the one hand, expressing the matching predicate in terms of restrictions over the fields of the operation types it matches, and on the other hand, expressing the context exposure requirements in terms of fields of those operations that must be reified. The major issue regarding the compilation of a pointcut is determining how many hooksets and call descriptors are required to represent a given pointcut. As initially presented in Section 5.3.1, a single *primitive* hookset and a single call descriptor should be enough. However, this is not always the most appropriate solution, because a pointcut may define different context exposure requirements for subsets of the join points it matches. In order to illustrate this, consider pointcut `bar` shown in Listing 5.3. It selects two join point subsets, one of `MCall` join points and the other of `MExec` join points, and imposes different context exposure requirements for them. The `MCall` subset binds the `moveXY` parameters in the natural order, while the `MExec` subset binds them in the reverse order. Also consider pointcut `foo`, which selects two join point subsets, one enclosing the `MCall` occurring inside lines and the other occurring inside composite shapes. For both subsets it imposes context exposure requirements analogous to those of the `bar` pointcut.

```
pointcut bar(int #aX, int #aY):
  call(void moveXY(int,int)) && args(#aX, #aY) ||
  execute(void moveXY(int,int)) && args(#aY, #aX)

pointcut foo(int #aX, int #aY):
  call(void moveXY(int,int)) && args(#aX, #aY) && within(Line) ||
  call(void moveXY(int,int)) && args(#aY, #aX) && within(Composite)
```

Listing 5.3: Pointcut Examples <PA Code>

Note that representing the `bar` pointcut with only one hookset is possible, but is far from being an elegant solution. The selector of this hookset would have to be defined over the `Operation` class, since the pointcut selects both `MsgSend` and `MsgReceive` operation types. The selector predicate would have to check which is the type of operation object (i.e. using `instanceOf`) and cast the operation object into the appropriate operation class in order to check the restrictions[2]. In the same way, using only one call descriptor (for the context exposure requirement)[3], would have to also check on the type of the operation in order to gather the exposed parameters; or even worst, in pointcuts like `foo` it would have to reevaluate the selector to decide how context exposure should be done. In order to achieve a more elegant solution and take full advantage of the Kernel language, a pointcut usually shall be represented with a *composite* hookset, which is composed by a collection of `primitive` hooksets. Each primitive hookset selects occurrences of only one operation type (avoiding the `instanceOf` issues). Also, each primitive hookset has an associated call descriptor specifying its context exposure requirements (avoiding the `instanceOf` and selector re-evaluation issues). For example, Figure 5.2 illustrates how pointcuts `bar` and `foo` are represented using this approach. Pointcut `bar` requires two hooksets, one selecting the method invocations and the other the executions, along with their corresponding call descriptors specifying how the arguments must be exposed for them. The `foo` pointcut is analogous, but the two hooksets select the method invocations inside the `Line` and `Composite` shapes, respectively.

The next section presents the intermediate structure used by the compiler to represent the decomposition of a pointcut. Section 5.3.1.2 explains the pointcut reduction process, which allows to extract the required hooksets and call descriptors form the

[2]At least, for those restrictions that require using getter methods not defined in the `Operation` class.

[3]Actually, it would be one call descriptor for each advice defined over the pointcut.

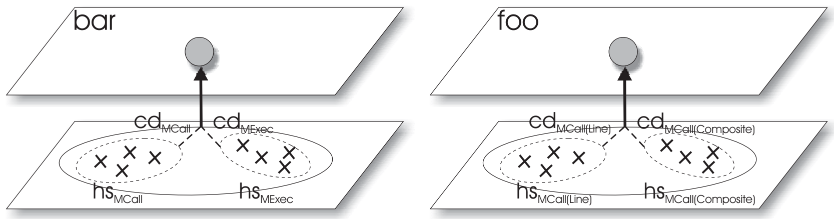


Figure 5.2: `bar` and `foo` Pointcut Representation

intermediate structure of a pointcut. Such a process is illustrated in Section 5.3.1.3. Finally, Section 5.3.1.4 discusses how the order in which the PA machine evaluates a pointcut may affect parameter bindings and informally shows that the compiler indeed preserves such an order.

5.3.1.1 Pointcut Representation

The intermediate structure of a pointcut allows to express the semantics of the pointcut declaration using the Kernel langauge vocabulary rather that the PA language vocabulary, i.e. it expresses a pointcut in terms of: operation types to be selected rather than join point kinds, restrictions over those operation types rather that over the join point kinds and context exposure requirements over those operation types rather that over the join point kinds. The intermediate structure consists of the pointcut name, the list of parameters exposed to the context and the body of the pointcut represented through the data type `RTree Trip` shown in Listing 5.4. The data type `RTree Trip` expresses the semantics of the pointcut in terms of a tree with logical operators as nodes and `Trip` data types as leafs. The `Trip` data type is meant to express a selection predicate in terms of Kernel language operation types and a list of values to be reified upon the matching of the selection predicate. The name of the data type `Trip` is an abbreviation for *triplet* and comes from the fact that a PCD can be decomposed in three main elements: a restriction over the kind of the join point, restrictions over the data exposed by that join point kind and context exposure requirements.

The intermediate structure is built based on the `PointcutDecl` structure used by the PA language to represent a pointcut (see Section 4.2.1.1). Its name and parameters are those of the `PointcutDecl` data type. The body of the pointcut, which in the PA machine is represented by a tree where the nodes are PCD combinators and the leafs are PCDs, is transformed into an *isomorphic* `RTree Trip` tree by:

```
data RTree r = RTAnd (RTree r) (RTree r) |
               RTOr (RTree r) (RTree r)  |
               RTNot (RTree r)           |
               RTLeaf r

data Trip = TKinded Kind [Res] [Ctx] |
            TUnKinded [Res] [Ctx]    |
            TNone
```

Listing 5.4: Pointcut Representation <Haskell Code>

- representing the `AndPCD`, `OrPCD` and `NotPCD` logical combinators (see Section 4.2.1.1) with the `RTAnd`, `RTOr` and `RTNot` tree constructors, respectively.
- representing the `PCDs` at the leafs with the `RTLeaf` tree constructor, each of them containing a `Trip` element expressing the matching predicate and context exposure requirements of the corresponding PCD.[4]

For example, Listing 5.5 illustrates the transformation of a simple pointcut body. The pointcut selects all the advices and method executions that fulfill some criteria (not shown here) and binds their arguments to the pointcut arguments. The isomorphic tree is built by replacing the logical combinators in the nodes, as previously mentioned, and replacing the PCD in the leafs as follows. The kinded PCDs, like `AExecPCD` and `MExecPCD`, are represented with `TKinded` leafs. This type of leafs represent selection predicates defined over an operation of certain type. The unkinded PCDs, like `ArgsPCD`, are replaced by `TUnKinded` leafs, which represent selection predicates that can be matched against any operation type.

```
AndPCD                          RTAnd
   (OrPCD                          (RTOr
      (AExecPCD)          ⟹           (RTLeaf (TKinded ...))
      (MExecPCD ...))                 (RTLeaf (TKinded ...)))
   (ArgsPCD ...)                   (RTLeaf (TUnKinded ...))
```

Listing 5.5: Pointcut Representation Example <Haskell Code>

The `TNone` triplet shown in Listing 5.4 is not used during the construction of the pointcut representation, it shall be used by the pointcut reduction process. It is meant to model selection predicates that never match any operation occurrence. For

[4]In the implementation there is an extra constructor for the `Trip` data type, `TReference`. It is used to represent a reference to a user-defined pointcut inside a pointcut declaration, similar to the `NamedPCD` used in the PA machine (see on Section 4.2.1.1). Since presenting it does not offer any new interesting notion, we shall omit it.

instance, consider a pointcut definition like `call(..)  && execute(..)` that never matches any join point, because a join point cannot have two kinds at the same time. See the next section for a detailed explanation.

The selection predicate of a `T(Un)Kinded` triplet is represented by a list of restrictions to impose over the operation type(s) it selects, where a restriction is represented by the data type `Res`, see Listing 5.4. The specification of which values must be exposed to the context for a `T(Un)Kinded` triplet is represented by a list of `Ctx` data types, where each element of the list specifies a value that must be reified upon the selection of an operation occurrence, see Listing the 5.4.

```
data Kind = MsgSend | MsgReceive | FieldGet | FieldSet |
            Instantiation | MsgReceiveMO

data OField = FRetType | FName | FParams | FWhereCN | FWhereMN |
              FThis | FTarget | FArgs

data Res = RField OField (RTree ResCond) |
           RGeneral (RTree ResCond)

data ResCond = RCInsOf Type | RCName Name | RCType Type |
               RCList [ResCond] | RCCFlow Bool PCName

data Ctx = CtxThis ParPos | CtxTarget ParPos | CtxArg ParPos ArgPos |
           CtxCFlow ParPos PCName ParPos
```

Listing 5.6: Pointcut Representation (continued)

Listing 5.6 shows all the data types related to the definition of the `T(Un)Kinded` triplets, which are:

Kind Data Type: it represents the kinds over which a `TKinded` triplet may be defined, expressed in terms of Kernel operation types. The PA machine supports six different kinds of join points that can be matched by a pointcut, which are: `MCall`, `MExec`, `FGet`, `FSet`, `Inst` and `AExec` (see Section 4.3.1). For these kinds, the Kernel machine provides an operation capable of reifying them. The first five join point kinds are reified by the: `MsgSend`, `MsgReceive`, `FieldGet`, `FieldSet` and `Instantiation` operation, respectively. Since advices are compiled into methods, the advice execution is also reified by the `MsgReceive` operation, but restricted to those methods generated for the advices defined in the PA program. The `MsgReceiveMO` is a fictitious operation type, representing the `MsgReceive` operation restricted to the advice methods.

OField Data Type: it represents the fields of the five supported operation types required in order to express all the restrictions that the PCDs of the PA language may impose. Each operation type supports a subset of those fields. For instance, the `MsgSend` operation supports all of them, whereas `Instantiation` supports only: `FName` for the name of the class to be instantiated, `FWhereCN` and `FWhereMN` for the name of the class and method where the operation occurs and `FThis` for the *this* object.

Res Data Type: it represents the restrictions that a triplet may impose. A restriction can be either a field restriction or general restriction. The former is represented by the constructor `RField` and corresponds to a restriction defined over a field. It consists of an `OField` value, representing the field that it restricts, and a condition represented by the `RTree ResCond` data type[5]. The latter is represented by the constructor `RGeneral` and corresponds to a restriction that is not related to any field. Actually, the only restrictions that fit in that category are the control flow restrictions. Those restrictions only consist of a condition represented by the `RTree ResCond` data type.

ResCond Data Type: it represents the different types of conditions that the PCDs of the PA language may impose. The `RCInsOf` condition is applied over the fields that hold operation's dynamic information, i.e. `FThis`, `FTarget` and `FArgs`. The `RCName` is applied over the field `FName`, e.g. to restrict the name of a method or field. The `RCType` is applied over the field `FRetType`, e.g. to restrict the return type of a method. The `RCList` is applied in combination with the `RCType` and `RCInsOf` to restrict the type of the formal parameters of a method (`FParams`) and the type of its actual arguments (`FArgs`), respectively. The `RCCFlow` represents a control flow restriction, where the boolean value states if it corresponds to a `cflow` or `cflowbelow`, and the `PCName` is the name of the nested pointcut.

Ctx Data Type: it specifies the different pieces of information that can be exposed form the context of a join point. Those pieces of information can be: the *this* object, the target object, an argument value and a value exposed by an event collector metaobject. The first parameter of each constructor in `Ctx` (i.e. `ParPos`) states the binding to a parameter of the underlying pointcut, by giving its position. For instance, consider a pointcut with a signature like `moveXY(Shape shape, int x, int y)`, `CtxTarget 0` would state that the parameter at position zero (i.e. `shape`) is bound to the target object. The second parameter of the `CtxArg` specifies the index of the argument in the captured join point. For instance, `CtxArg 1 0` would state that the second parameter of the pointcut `moveXY` is bound to the first argument of the join point. Finally, `CtxCFlow` specifies a binding between a parameter of the underling pointcut (i.e. the first `ParPos`) and a parameter of the nested pointcut, where the nested pointcut is

[5]Note that the `RTree ResCond` represents a logical expression tree, which may combine various individual conditions. The definition of the polymorphic type `RTree` is shown in Listing 5.4.

identified by its name (i.e. PCName) and the parameter by its position (i.e. the second ParPos).

Listing 5.7 shows three examples of the generation of the RTree Trip for a pointcut. In order to simplify the presentation, we do not include all the type constructors needed. In the first example, the conversion of the call PCD produces a TKinded triplet defined over the MsgSend operation, along with three restrictions: one for the return type, one for the name of the method and one for the type of the formal parameters of the method. Since the call PCD does not bind any pointcut parameters, the Ctx list is empty. The conversion of the second example produces a TUnKinded triple with one restriction imposing that the *this* object must be an instance of the Point class. Also, it specifies that the *this* object must be bound to the first parameter of the pointcut. The third example also produces a TUnKinded triple imposing a control flow restriction over the pointcut inPoint and exposes the first parameter of the inPoint pointcut as its first parameter.

```
pointcut callmoveXY(): call(void moveXY(int, int)) ⟹
  TKinded MsgSend [RField FRetType (RCType void)
                   RField FName (RCName "moveXY")
                   RField FParams (RCList [RCType int, RCType int])] []

pointcut inPoint (Point aPoint): this(aPoint) ⟹
  TUnKinded [RField FThis (RCInsOf Point)] [CtxThis 0]

pointcut inCFlowPoint (Point aPoint): cflow(inPoint(aPoint)) ⟹
  TUnKinded [RGeneral (RCCFlow False "inPoint")]
            [CtxCFlow 0 "inPoint" 0]
```

Listing 5.7: Pointcut Representation Generation Example

In the rest of this chapter we shall use the term *pointcut intermediate structure* to refer indistinctly to both the whole intermediate structure of a pointcut (containing the pointcut name, parameters and RTree Trip) and the structure RTree Trip representing its body.

5.3.1.2 Pointcut Reduction

The pointcut reduction process is meant to transform a RTree Trip representing a pointcut into another RTree Trip that preserves the semantics of the pointcut and where all its nodes are RTOr (union) operators and the leafs are TKinded elements. Each TKinded leaf represents a subset of the join points matched by the pointcut. Also, the join points of each subset are all of the same kind and all of them share the same context exposure requirements. In other words, the set of join points matched by a pointcut, let's call it S, is decomposed into subsets containing the same kind of join points, let's call those subsets S_{k_i} where k_i is the kind of the subset, thus $S = S_{k_1} \cup ... \cup S_{k_n}$. In addition, those subsets are further decomposed into subsets containing the join point with the same context exposure requirements, let's call those subsets S_{k_i,ce_j} where ce_j represents a particular way of binding the pointcut parameters, thus $S_{k_i} = S_{k_i,ce_1} \cup ... \cup S_{k_i,ce_{m_i}}$. From the point of view of the `RTree Trip` generated by the reduction process, each S_{k_i,ce_j} represents a `TKinded` leaf in that tree, enclosing: the kind, the restrictions over that kind and context exposure requirements. The $\cup$ operators represents the the `RTOr` nodes of the tree. The leafs of the resultant tree contains the information required by the compiler in order to generate the primitive hooksets and call descriptors that together represent the reduced pointcut. The algorithm described here illustrates only one approach to performing such a decomposition, it does not aim to be the best approach. Improving this algorithm is left as further work.

The reduction algorithm is pretty straightforward. It consists in three steps which are performed sequentially. Those steps are:

Not Reduction The first step of the algorithm eliminates all the `RTNot` nodes from the tree. The algorithm walks through the tree looking for `RTNot rt` nodes, which are replaced by the tree resulting from the negation of its nested tree (`rt`). The negation of a tree involves the following cases:

- Trees of the form `RTAnd` $\mathtt{rt}_1$ $\mathtt{rt}_2$ and `RTOr` $\mathtt{rt}_1$ $\mathtt{rt}_2$ are negated by applying *De Morgan Laws*, resulting in `RTOr` $\mathtt{rt}'_1$ $\mathtt{rt}'_2$ and `RTAnd` $\mathtt{rt}'_1$ $\mathtt{rt}'_2$, respectively, where $\mathtt{rt}'_i$ is the negation of $\mathtt{rt}_i$.

- Trees of the form `RTNot rt` are negated to `rt`.

- The leafs of the tree are negated by negating its `Trip` element. The `Trip` at this point can be either `TKinded` or `TUnKinded`, since the `TNone` may only appear as the result of the composition of two `Trip` elements and composition is performed once the `RTNot` elements have been eliminated. A `TKinded` leaf represents the set of join points of certain kind (k_i) which fulfill the specified restrictions ($r_1...r_n$), consequently the negation of the `TKinded` leaf is the tree that express the complement of that set. The complement tree must match all the join points which are not of kind k_i and all the join points of kind k_i that do not fulfill at least one of the restrictions $r_1...r_n$. The former is represented by one `TKinded` element for each supported kind, with the exception of k_i, with an empty set of restrictions. The latter is represented by n `TKinded` elements of kind k_i, where the `TKinded` number i has only one restriction, which is the negation of r_i. The complement tree is the union of all these `TKinded` elements, thus it is `RTOr Trip` tree containing them. The negation of a `TUnKinded` leaf is analogous, but it only imposes a collection of restrictions, $r_1...r_n$, thus its complement tree includes n `TUnkinded` leafs, where the `TUnKinded` number i has the negation of r_i as its only restriction.

And Reduction Once all the `RTNot` nodes have been eliminated, the algorithm goes through the tree once more, now looking for `RTAnd` $\texttt{rt}_1$ $\texttt{rt}_2$ nodes and replacing them with the tree resultant from composing its two child nodes. The composition of the two child nodes is done by first performing the *and reduction* over the two child trees, let's call those reduced trees $\texttt{rt}'_1$ and $\texttt{rt}'_2$ respectively, and then performing the distributive composition of the two reduced trees. Note that at this point of the algorithm $\texttt{rt}'_1$ and $\texttt{rt}'_2$ can be either an `RTOr` tree or a leaf. The distributive composition is performed by flattening $\texttt{rt}'_1$ and $\texttt{rt}'_2$, and distributively composing the leafs of the two lists, resulting in a list of composed leafs. Once that list has been calculated, it is transformed back intro the `RTOr` tree representing the *and reduction* of `RTAnd` $\texttt{rt}_1$ $\texttt{rt}_2$. The composition of two leafs is performed by composing the two `Trip` elements, Listing 5.8 shows the `tripCompose` function responsible for performing such a composition.

```
1  tripCompose :: Trip -> Trip -> Trip
2  tripCompose t t' =
3   case (t, t') of
4    (TKinded k xr xc, TKinded k' xr' xc')
5      | (k /= k') -> TNone
6      | otherwise -> TKinded k (resCompose xr xr') (xc ++ xc')
7    (TUnKinded xr xc, TUnKinded xr' xc') ->
8      TUnKinded (resCompose xr xr') (xc ++ xc')
9    (TKinded k xr xc, TUnKinded xr' xc')
10     | (support k xr') -> TKinded k (resCompose xr xr') (xc ++ xc')
11     | otherwise       -> TNone
12   (TNone, _) -> TNone
13   (_, TNone) -> TNone
14   (t, t') -> tripCompose t' t
```

Listing 5.8: `Trip` Composition <Haskell Code>

The composition of two `TKinded` elements of different kinds (line 5) results in `TNone`, since a join point cannot be of two different kinds at the same time. In the case that the kinds are equal (line 6), the composition results in a `TKinded` where both the lists of restrictions and the list of context exposure requirements are merged[6]. The function `resCompose` merges both list of restrictions by grouping restrictions of the same type. The composition of two `TUnKinded` always results in a `TUnKinded` element where the two lists of restrictions and the two lists of context exposure requirements

[6]Note that the merging is performed by concatenating both lists. Once a pointcut is fully-reduced, the compiler checks that each parameter is bound exactly once.

are merged respectively, see lines 7 and 8. As shown in lines 9 to 11, the composition of a `TKinded` and a `TUnKinded` elements has two cases. The first is when the restrictions imposed by the `TUnKinded` elements `xr'` are supported by the kind `k`, in which case the composition is done as previously mentioned. The second is when one or more of those restrictions are not supported by the kind, which results in a `TNone` triplet. For instance, consider the pointcut `fget(..) && args(..)`, since the argument restriction is not supported by the field get kind, we know in advance that the composition of both PCDs would never select a join point.

Homogenization Once the algorithm reduces all the `RTAnd` nodes, the following tasks are done:

- All the `TNone` leafs are eliminated, since they represents empty set of join points in a tree where all the nodes are union operators.
- All the `TUnKinded xr xc` leafs that may remain in the tree are expressed in terms of `TKinded` elements. Hence, for each supported kind $\texttt{k}_i$ that supports all the `xr` restrictions a triplet of the form `TKinded` $\texttt{k}_i$ `xr xc` is formed. The triplet `TUnKinded xr xc` is expressed as the union of those `TKinded` triplets.

5.3.1.3 Illustration

In order to illustrate the reduction process, consider the pointcut `moveFromComp` shown in Listing 5.9. It selects all the invocations to the method `moveXY` made over a `Point`, which are performed either from a `Composite` shape or in the control flow of an invocation to `moveXY` on a `Composite` shape. In addition, it exposes both the `Composite` and the `Point` shapes. Note that the pointcut `moveFromComp` is somehow redundant in the sense that if we remove the `this(c)` PCD, the pointcut would select the same set of join points[7]. Nonetheless, we define it in that way in order to better illustrate the reduction process. Also, in the next section we use it to show how the order of evaluation of the pointcut may affect the parameters' binding.

```
pointcut compMove(Composite c):
  execution(void moveXY(int, int)) && this(c);

pointcut moveFromComp(Composite c, Point p):
  call(void moveXY(int, int)) && target(p) &&
  (this(c) || cflow(compMove(c)));
```

Listing 5.9: Pointcut Reduction Example - Declaration <PA Code>

[7]Assuming that the Shape Editor example is closed.

Listing 5.10 shows the intermediate representation of the pointcut `moveFromComp` (lines 1 to 4) and how this representation is reduced (lines 5 to 13). With the purpose of simplifying the example presentation, the restrictions and context exposure requirements are abbreviated. For instance, `Target`$_{Point}$ represents a `RCInsOf` restriction over the target object and `CFlow`$_c$ represents that the parameter `c` is bound to the parameter exposed by the `compMove` pointcut. Since the pointcut does not include any `!` operators, the *not reduction* returns an identical tree. The *and reduction* is divided in two steps. The first step, on lines 6 to 8, reduces the innermost `RTAnd` node, which implies distributively composing the `TUnKinded` element corresponding to the `target` PCD with the two `TUnKinded` leafs of the `RTOr` tree. The result is another `RTOr` tree holding the result of such a composition. The second step, on lines 10 to 13, reduces the remaining `RTAnd` by distributively composing the `TKinded` element with the two `TUnKinded` elements. The result is an `RTOr` tree where the left leaf selects the invocations to `moveXY` performed directly from a `Composite` shape, while the right branch selects those in the control flow of the pointcut `compMove`. Since the tree resulting from the *and reduction* is already homogenized, the *homogenization step* returns an identical tree. Note that the two `TKinded` leafs exhibit the two different forms in which parameters are bound by the pointcut `moveFromComp`.

```
1  RTAnd (TKinded MsgSend [RetType$_{void}$, Name$_{moveXY}$, Args$_{int,int}$] [])
2        (RTAnd (TUnKinded [Target$_{Point}$] [Target$_p$])
3               (RTOr (TUnKinded [This$_{Composite}$] [This$_c$])
4                     (TUnKinded [CFlow$_{compMove}$] [CFlow$_c$])))
5  $\Longrightarrow_{red_{And-Step1}}$
6  RTAnd (TKinded MsgSend [RetType$_{void}$, Name$_{moveXY}$, Args$_{int,int}$] [])
7        (RTOr (TUnKinded [This$_{Composite}$, Target$_{Point}$] [This$_c$, Target$_p$])
8              (TUnKinded [CFlow$_{compMove}$, Target$_{Point}$] [CFlow$_c$, Target$_p$]))
9  $\Longrightarrow_{red_{And-Step2}}$
10 RTOr (TKinded MsgSend [RetType$_{void}$, Name$_{moveXY}$, Args$_{int,int}$,
11                        This$_{Composite}$, Target$_{Point}$] [This$_c$, Target$_p$])
12      (TKinded MsgSend [RetType$_{void}$, Name$_{moveXY}$, Args$_{int,int}$,
13                        CFlow$_{compMove}$, Target$_{Point}$] [CFlow$_c$, Target$_p$])
```

Listing 5.10: Pointcut Reduction Example - Process <Haskell Code>

5.3.1.4 Pointcut Evaluation Order

The order in which the PA machine evaluates the PCD combinators, involved in a pointcut definition, has semantic effects in the bindings it generates. For instance, consider the pointcut `moveFromComp` defined in the previous section and a join point representing an invocation to the method `moveXY` on a `Point` object, which is performed from inside the method `moveXY` of a `Composite` object. Such a join point is matched by `moveFromComp` and its two parameters, `c` and `p`, are bound to the *this* object and the target object, respectively. However, if we swap the two PCDs of the `||` combinator (i.e. *alter the order*), the join point would also be matched, but parameter `c` would be bound to the `Composite` object exposed by `compMove`. In the case of `moveFromComp`, such a swap would not have any semantic effects that can be perceived by the user, because in both definitions of the pointcut, `c` would be bound to the same object. However, in a pointcut like `foo(Composite c):this(c) || target(c)` it does have a semantic impact. From the point of view of our compiler, preserving the order in which the parameters must be bound is crucial to ensure that both the original pointcut and the compiled version bind the same pieces of information.

Figure 5.3 illustrates how the pointcut `moveFormComp` gets compiled. The tree on the left side of the figure represents the AST of its definition. The one in the middle is the `RTree Trip` used by the compiler to represent the pointcut definition. Since this tree is isomorphic to the pointcut's AST, it preserves the order specified in the pointcut definition. Finally, the tree on the right side is the reduced `RTree Trip`. Based on the reduced tree, the compiler generates the corresponding hookset and call descriptor for each leaf. Those elements are used in the definition of the corresponding `link`, preserving the order defined by the reduced tree.

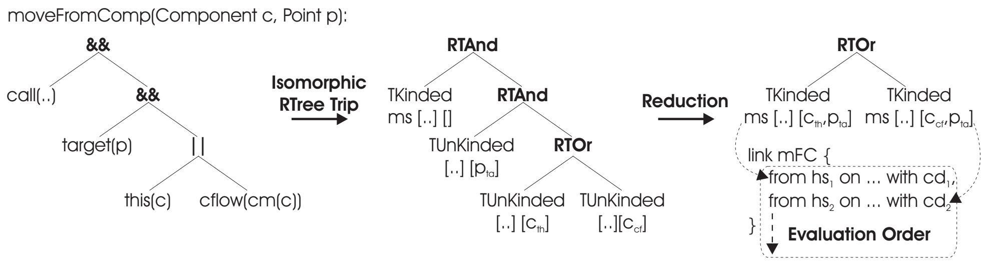

moveFromComp(Component c, Point p):
&&
call(..)
&&
target(p)
||
this(c)
cflow(cm(c))
Isomorphic RTree Trip
RTAnd
TKinded ms [..] []
RTAnd
TUnKinded [..] [p_{ta}]
RTOr
TUnKinded [..] [c_{th}]
TUnKinded [..][c_{cf}]
Reduction
RTOr
TKinded ms [..] [c_{th},p_{ta}]
TKinded ms [..] [c_{cf},p_{ta}]
link mFC {
from hs_1 on ... with cd_1,
from hs_2 on ... with cd_2
}
Evaluation Order

Figure 5.3: Pointcut Evaluation Order

Note that the reduced tree would be evaluated by the Kernel machine following the same order used by the PA machine. That is to say, the `RTOr` first tries to match the operation with the left branch, and if it fails, it tries with the right branch (see the `pcMatch` function in Section 4.3.4). Since both matching functions, `pcMatch` and `link` evaluation, evaluate the tree following the same order, it remains to analyze if the reduction of the tree does indeed preserve the order in which parameters are bound.

In order to analyze this, let's analyze if each of the three steps of the reduction process preserves such an order.

- *Not reduction* transforms the negated sub-tree, which does not bind any parameter (remember that AspectJ explicitly forbids the negation of a parameter binding, see Section 4.3.4), into a semantically equivalent tree matching the same join points. From the point of view of the matching algorithm, this does not affect the parameters' bindings, since a branch that is expected to return a boolean value plus an empty list of bindings is replaced by another one doing the same.

- *And reduction* transforms a tree of the form `RTAnd` $\texttt{b}_{left}$ $\texttt{b}_{right}$, where the $\texttt{b}_{left}$ and $\texttt{b}_{right}$ branches have already been *and-reduced*, into a `RTOr` tree. Such a tree represents the union of the distributive composition of the leafs of $\texttt{b}_{left}$ with the leafs of $\texttt{b}_{right}$. Such a composition is performed in the following way. In first place, the `RTOr` trees of both branches are flattened preserving the order (i.e. the nodes in the left branch are before the ones in the right branch). Therefore, the two branches have the form `left=`$\cup[L^l_1, .., L^l_n]$ and `right=`$\cup[L^r_1, .., L^r_m]$. Secondly, both lists are composed, resulting in the following list $\cup[L^l_1 \cdot L^r_1, .., L^l_1 \cdot L^r_m, .., L^l_n \cdot L^r_1, .., L^l_n \cdot L^r_m]$, where $\cdot$ is the leaf composition operator. Finally, the resulting list is transformed back into an `RTOr` tree, once again preserving the order. Note that the `pcMatch` function of the PA machine, would evaluate the tree `RTAnd` $\texttt{b}_{left}$ $\texttt{b}_{right}$ as follows. First, it evaluates L^l_1. If L^l_1 matches then $\texttt{b}_{left}$ has been matched, hence it continues by evaluating L^r_1. If L^r_1 matches, then $\texttt{b}_{right}$ has been matched, hence it ends returning those bindings of $L^l_1 \cdot L^r_1$. If L^r_1 does not match, it continues with L^r_2 `up to` L^r_m. If L^l_1 does not match, it continues with L^l_2 up to L^l_n. Indeed, this is exactly the order in which parameters are bound as a result of the distributive composition, hence *and reduction* does preserve the order. See Listing 5.10 for an example of how *and reduction* applied to pointcut `moveFromComp` preserves such an order.

- *Homogenization reduction* performs well-localized transformations over the *and-reduced* tree resultant from the previous step. By well-localized we mean that it replaces `TUnkinded` elements by `RTOr` trees with `TKinded` leafs, having the same context exposure requirements. Therefore, it does not alter the order in which parameters are bound. In addition, it also removes the `TNone` leafs, also preserving the order of the `RTree Trip`.

5.3.2 Advice Metaobject Generation

The behavior that an advice defines is compiled into a metaobject class containing a single method holding it. The metaobject class is an standard BASE class. The generation of the methods for the `before` and `after` advices is straightforward (see next section), while for `around` advices it also implies the compilation of the `proceed` expression that may be immersed in its body (see Section 5.3.2.2).

5.3.2.1 Before and After Advices

A `before` or `after` advice is compiled into a method that has the same parameters and body as the advices declaration. Since advices are nameless, the compiler generates the name of the method. The return type of the method is `void`, because neither advice kind returns a value. Listing 5.12 shows an example of the compilation of a `after` advice.

```
after(int x, int y):                  class AdvMO {
     moveSinglePointArgs(x,y){          void adv(int x, int y){
  write "Point moved: " ++ x ++   ⟹      write "Point moved: " ++ ...;
        ", " ++ y;                      }
}                                     }
```

Listing 5.11: Advice Compilation Example <PA/Kernel Code>

5.3.2.2 Around Advices

The compilation of an `around` advice has to deal with two additional tasks: `proceed` expression generation and boxing/unboxing of primitive types. Both tasks are discussed in what follows. In addition, note that the generated method shall be associated to a link with `AROUND` control, hence its return type must be `Object` (see Section 3.3.2.5).

Proceed expression As explained in Section 4.3.4, the PA machine uses a mapping, represented by the *internal* machine structure `CtxBind`, in order to perform the modifications to the context of the current join point during the evaluation of the `proceed` expression. Such a mapping states, for each argument that the `proceed` expression receives, which value in the context must be modified. Even though the Kernel machine provides the `proceed` expression in order to continue with the evaluation of the original operation occurrence (the original join point), it does not provide any built-in mechanism to know which fields of the operation occurrence must be modified before proceeding with the operation occurrence evaluation. This difference in the level of abstraction of both `proceed` expressions implies that the compiler must handle the mapping explicitly in the generated Kernel code.

Since the PA `proceed` expression takes as its arguments those passed to the underlying around advice, the mapping specification can be inferred from the `RTree Trip` of the associated pointcut. As explained in Section 5.3.1.2, each leaf of the reduced tree contains the specification of the values that must be reified to the advice metaobject. Consequently, each leaf encloses the knowledge of which piece of information is represented by each reified value. The compilation of the `proceed` embodies two activities. On the one hand, the call descriptor generated for each leaf of the reduced tree must build a structure holding the mapping specification for the set of operations occurrences it selects and it also must pass the structure as an additional parameter to the advice method. On the other hand, each PA `proceed` expression inside the advice must be replaced by the appropriate Kernel `proceed` expression, which based on the mapping specification performs the appropriate modification to the operation occurrence before proceeding. Figure 5.4 graphically illustrates how the mapping is generated and passed to the metaobject. The dashed ovals and lines represent the hookset and the call descriptor generated for each leaf of the `RTree Trip`, respectively. Once an operation is selected by one of the hooksets, its associated call descriptor, besides reifying the corresponding values, generates the mapping structure and passes it to the advice metaobject. The mapping structure is modeled as an array of integers that specifies, for each parameter, which piece of context information it represents.

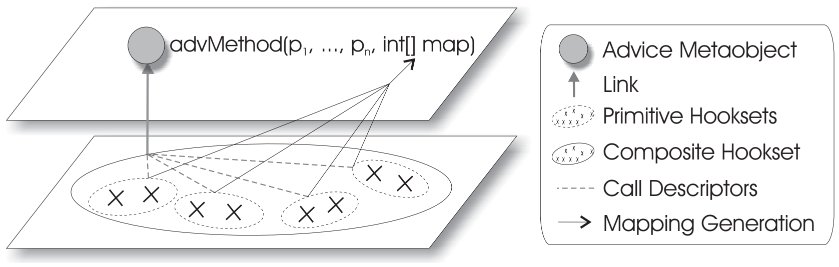


Figure 5.4: `proceed` Mapping

Listing 5.12 illustrates the generation of the advice method for an around advice that make use of `proceed`. The advice is bound to the pointcut `moveFromComp`, presented in Section 5.3.1.3. Once the advice gets executed, it logs the `moveXY` call before and after invoking `proceed`. Note that the `proceed` expression changes the `Composite` shape by calling the `c.getParent()`. Assume that such a method returns the `Composite` shape that contains the `c` shape. The generated method receives, besides the two parameters of the advice, the mapping array. As mentioned in Section 5.3.1.3, `moveFromComp` defines two forms of binding `c`: if the `moveXY` call occurs inside the `Composite` shape, `c` is bound to the *this* object, and if the call occurs in the control flow of the method `Composite.moveXY`, `c` is bound to the value exposed by an event collector. Meanwhile, `p` is always bound with the target object. The generated `proceed` expression, based on the `map` argument, performs the changes to the operation occurrence (available through the `oper` implicit variable, see Section 3.3.2.6) using the setter methods. Note that based on the `map` argument, the proceed expression determines whether the `c` parameter represents the *this* object, in which case it modifies the operation occurrence using `setThis`, or if it represents the event collector value, in which case no modification is performed[8]. In the case of `p`, there is no need to use the mapping, since it is always bound to the target object. Actually, the `setTarget` can be removed from Listing 5.12, since it does not alter the operation occurrence. However, we leave it for illustration purposes.

```
void around(Composite c, Point p): moveFromComp(c,p){
  write "Before: Moving " ++ p ++ " from inside " ++ c;
  proceed(c.getParent(),p);
  write "After: Moving " ++ p ++ " from inside " ++ c;
}
⟹
     class advMO {
       Object adv(Composite c, Point p, int[] map){
         write ...;
         proceed{
           if (map[0] == THIS_const) oper.setThis(c.getParent());
           oper.setTarget(p);
         }; write ...; null;
       }
     }
```

Listing 5.12: `proceed` Generation Pseudocode <PA/Kernel Code>

Boxing and Unboxing There are two situations in which the compiler must deal with boxing and unboxing. On the one hand, when at least one parameter of the advice is of a primitive type and the advice encloses a `proceed` invocation, those parameters must be boxed before performing the operation occurrence modification. On the other hand, when the return type of the advice is a primitive type, the value returned by the `proceed` expression may need to be unboxed. For instance, consider an around advice declared with `int` as its return type and that its body encloses an expression like `proceed(..) + 1;`. Since the `proceed` in the Kernel machine returns an object, the `int` value it returns must be unboxed in order to perform the arithmetic operation. In addition, since the return type of an around advice is `Object` the value returned by the advice may also need to be boxed. For instance, consider an advice like `int around()..{..; 1;}`. The `1` must be boxed before returning.

[8]Remember that values exposed through the control flow relation cannot be modified by the `proceed` expression, see Section 4.3.4.

5.3.3 Kernel Constructions Generation

The generation of the Kernel constructions required for an advice is straightforward: a hookset and a call descriptor is generated for each `TKinded Kind [Res] [Ctx]` of the `RTree Trip` tree, and a link is generated binding the hooksets to the advice metaobject.

Hookset The hooksets are generated using primitive hooksets with an embedded selector declaration (see Section 3.3.2.2). The selector only admits operation occurrences of the type specified by the `Kind` of the corresponding triplet. Each `Res` in the list is converted into an expression imposing its restriction. The selection predicate of the selector is an expression that denotes the *and* of all the expressions generated for the `Res` list. The `RField` restrictions produce expressions checking some condition over the operation occurrence. The generation of the expressions for the control flow restrictions (i.e. `RGeneral` restrictions) is explained in the next section.

Call Descriptor In the call descriptor definition, the expected type of the operation occurrence and the metaobject are obtained from the `Kind` and the associated advice metaobject, respectively. The body of the call descriptor is generated by generating an expression that is capable of obtaining the value corresponding to each `Ctx`, and invoking the advice method passing those values. In addition, for *around* advices, the mapping array may also be generated and passed as an extra argument to the advice method. The next section explains how the values exposed by an event collector metaobject can be obtained from the call descriptors.

Link The binding specification of the link is generated by associating each pair hookset-call descriptor with the *control* value corresponding to the kind of the advice being generated. The bindings must be generated respecting the order defined by the `RTree Trip`, as explained in Section 5.3.1.4. The metaobject initialization block of the link simply instantiates the advice metaobject.

In addition, if the pointcut associated to the advice has control flow restrictions, for each different restriction an event collector metaobject must be generated, along with its corresponding hooksets, call descriptors and link. This is explained in the next section.

5.3.3.1 Control Flow Restrictions

The compilation of a control flow restriction requires an event collector metaobject to be setup for the nested pointcut, exposing the control flow information; and for the pointcut containing the control flow restriction, its hooksets must check whether or not the operation is in the control flow of the event collector and its call descriptors must obtain from the event collector the values exposed by the nested pointcut.

The event collector metaobject is setup in such a way that it is notified before and after the occurrence of an operation matched by its nested pointcut. Since the nested pointcut may expose context information, there are two different implementations for the event collector metaobject: *counter* and *stack* based. Listing 5.13 shows an example of the code for both implementations. The counter based implementation is used when the nested pointcut does not expose any information from the context. Its implementation is based on a counter which is incremented (i.e. method `enter`) before the execution of the operation occurrence and decremented after (i.e. method `exit`) the execution. The event collector exposes the control flow information through the method `isInside`, thus when a hookset wants to check if the operation occurrence being matched is inside the control flow its nested pointcut, it simply invokes this method. The stack based implementation is used when the nested pointcut exposes information from the context of the join point. The `enter` method receives the exposed values as its arguments and pushes them into the stack. Actually, in the `StackBased` implementation shown in Listing 5.13 there is one stack, implemented as an array, for each exposed value (i.e. p_1 to p_n). The `free` variable holds the top of the `n` stacks. Besides providing the method `isInside`, exposing the information of being inside the control flow of its nested pointcut, this implementation also provides methods for accessing the context exposed values at the top of the stack, i.e. `getP1` to `getPN`.

```
class CounterBased {
 int counter;
 void init(){ counter:=0; }
 void enter(){
   counter := counter + 1;
 }
 void exit(){
   counter := counter - 1;
 }
 bool isInside(){
   counter > 0;
 }
}
```

```
class StackBased {
  $T_1$[] $p_1$; ... $T_n$[] $p_n$; int free;
  void init(){ $p_1$ := new $T_1$[MAX]; ...}
  void enter($T_1$[] $a_1$ ... $T_n$[] $a_n$){
    $p_1$[free] := $a_1$; ... $p_n$[free] := $a_n$;
    free := free + 1;
  }
  void exit(){ free := free - 1; }
  bool isInside(){ free > 0; }
  $T_1$ getP1(){ $p_1$[free - 1]; }
  ...
  $T_n$ getPN(){ $p_n$[free - 1]; }
}
```

Listing 5.13: Event Collector Metaobject Class <Kernel Code>

The generation of the link for the event collector is similar to the advice link generation. First its nested pointcut must be reduced. Then the hooksets and call descriptors must be generated for the reduced intermediate structure. Since the link binds the hooksets with `BEFORE` and `AFTER` controls, the call descriptor to be bound with `BEFORE` control must invoke the `enter` method passing the context exposure information (if any), and the call descriptor to be bound with `AFTER` control must invoke the `exit` method. Finally, the binding specification and metaobject initialization block of the link are generated. The metaobject initialization block must instantiate the corresponding event collector metaobject and invoke the `init` method (see Listing 5.13) in order initialize it.

As the reader may have noticed, up to this point we have not made any explicit distinction on how `cflow` and `cflowbelow` restrictions are implemented. The semantic difference between these two restrictions relies on the way in which they are composed in relation with the affected advice link. As mentioned in Section 4.3.4 the only difference between the `cflow` and `cflowbelow` is that the former considers the current join point as a candidate to be matched, while the latter does not. This semantic difference is illustrated in Figure 5.5. The link of the affected advice is represented $\texttt{L}_{adv}$. The first and third cases consider that the kind of the advice is *before* or *after*, while the second and fourth consider that the kind is *around*. The event collector link is $\texttt{L}_{cflow}$ (not shown in the Figure) and it is always associated with `BEFORE` and `AFTER` controls. The first two cases show how the two links, $\texttt{L}_{adv}$ and $\texttt{L}_{cflow}$, are composed to represent a `cflow` restriction. Note that in both cases $\texttt{be}_{cflow}$ and $\texttt{af}_{cflow}$ surround the link $\texttt{L}_{adv}$, thus the `enter` method of the event collector shall be invoked before checking the restriction. Meanwhile, in the last two cases $\texttt{be}_{cflow}$ and $\texttt{af}_{cflow}$ are wrapped inside the link elements of $\texttt{L}_{adv}$, thus the restriction is checked before invoking the `enter` method, representing the semantics for the `cflowbelow` restriction.

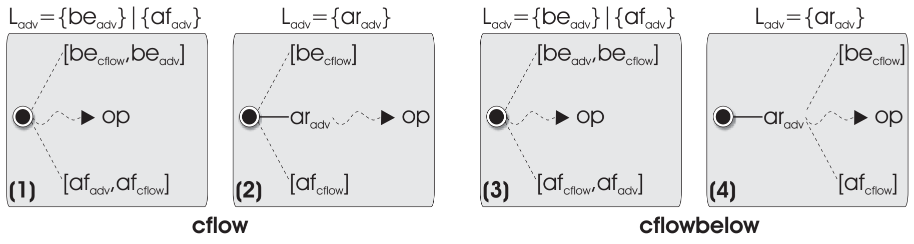


Figure 5.5: Control Flow Semantics

The current state of the Kernel machine does not allow to provide appropriate support for control flow restrictions in the cases that there is a link interaction between the control flow link and the advice link. The reasons are two: the lack of link composition specification and the implicit dependency limitation explained in Section 3.3.7.6. The former does not allow to appropriately differentiate between the `cflow` and `cflowbelow` semantics, because the appropriate link composition rules cannot be specified. The latter does not allow to state the dependency that exists between the links $\texttt{L}_{cflow}$ and $\texttt{L}_{adv}$. Therefore, in the context of an interaction scenario like the one described in the first case of Listing 5.5 and where the event collector is empty (i.e. the counter is zero or the stack is empty), the link $\texttt{L}_{adv}$ would never be selected, because the selector of $\texttt{L}_{adv}$ is evaluated before the `enter` method is actually invoked by $\texttt{be}_{cflow}$. This problem can be easily solved with the extension to the selector construction shown in Section 3.3.7.6, because the invocation to the `isInside` method could be done as a post-composition activity.

An alternative solution to the implicit dependency issue using the constructions available in the current implementation of the Kernel machine would be to differ the `isInside` check up to the call descriptor evaluation, i.e. doing it post-composition. Even though this solves the problem, it is extremely ad-hoc, therefore we do not use it for the compilation.

Providing an appropriate solution for the two issues mentioned is left as a future work. Once the implicit dependency limitation has been solved and the support for the specification of composition rules has been included, the solution to this issues is straightforward. As it is presented in Section 5.4, these two problems are solved in the context of the compilation from AspectJ to Reflex.

The expression required in order to check from a hookset that an operation occurrence is inside the control flow of an event collector must get a reference to the event collector metaobject (using the `linkName#MO`) and then invoke the `isInside` method over the metaobject. In the case of a call descriptor, the expression required in order to get a value exposed by an event collector metaobject must also get the metaobject reference and then invoke the appropriate `getPX()` method. Note that the number of the parameter (i.e. the `X`) is obtained from the corresponding `CtxCFlow` element.

5.3.3.2 Advice Execution

As presented in Section 4.3.1 the PA machine provides internal support for the advice execution join points, through the `AExec` join point constructor. The PA interpreter is in charge of constructing the `AExec` join point before evaluating an advice and performing the usual match-evaluate process for the join point. On the other hand, the Kernel machine does not provide any built-in mechanism to know whether an operation occurrence corresponds to either an object at the base or meta level. Therefore, the compilation process is responsible for providing the information of when a method execution represents a simple method or an advice method.

The hooksets generated for the `TKinded` triples with kind `MsgReceiveMO` are defined using the `MsgReceive` operation and require an additional restriction besides those imposed by the list of `Res` elements, which is that the executed method must be one of the methods generated for the advices. Note that, since the user may impose argument restrictions over the advice execution (using the `args` PCD), special care must be taken when generating the expression that checks that restriction on around advices, because their advice methods may include the additional parameter for the `proceed` mapping which must be ignored.

The fact that `MsgReceive` operation is used for reifying two different join points of the PA language, i.e. methods and advice executions, implies that the compiler must also ensure that the operation occurrences representing "standard" method executions are never selected by the hooksets generated for a `TKinded` triple with kind `MsgReceive`. Therefore the selection predicate of those hooksets must explicitly exclude the advice methods.

5.3.4 Limitations

There are some design decisions regarding the compilation process and the Kernel machine which do not allow to achieve a faithful representation of the PA language semantics under some circumstances. Those limitations on the support of the PA language semantics are presented next.

Visibility of the pointcut predicate. In order to express the semantics of a pointcut predicate, the compilation process must ensure that there is a bijective correspondence between the set of join points matched by a pointcut predicate and the set of operation occurrences selected by the hooksets representing the pointcut predicate. There is an issue in the compilation process, regarding the synthetic code introduced during the compilation, that makes it impossible for that correspondence to hold. By synthetic code here we mean, code inserted by the compiler, at the base or meta level, in order to support some high-level feature that the PA language supports internally and the Kernel language does not. Since this code resides at base or meta objects, it is a potential candidate to be reified and selected by a compiled pointcut predicate. This gives rise to a semantic difference. Those pieces of code are the event collector metaobject classes, the `proceed` expression blocks (e.g. invoking the setters methods of the operation object), the primitive type boxing/unboxing expressions included in the *around* advices.

A possible solution to this issue consists in imposing additional restrictions to the hooksets generated for the pointcuts, in order to exclude any operation originated by those pieces of code. The implementation of such a solution is easy, it requires generating a hookset that captures the operation occurrences related to synthetic code and composes it with all the other generated hooksets. Such a hookset has to include the following operation occurrences: those whose *where* class is an event collector, those which invoke a method over an operation class, those that instantiate a wrapper class and those which invoke a method of a wrapper class.

However, we believe that this solution, as any other solution that could be elaborated with the current implementation of the Kernel machine is ad-hoc. We believe that the Kernel machine should be extended to provide explicit mechanisms that facilitate dealing with this abstraction gap between the PA language and the Kernel language, giving place for a more general and principled approach to solve this kind of issues. Furthermore, in the general case, each aspect-language compiled into the Kernel machine must be unaware of the synthetic code used for its compilation and also for the one generated during the compilation of other aspect-languages. Therefore, this issue is left as a limitation of the current compiler and the elaboration of a mechanism to manage the abstraction gap is left as future work. In Section 5.4.4 we shall come back to this limitation, but in the context of the AspectJ to Reflex compiler.

Control flow. The semantics of control flow restrictions cannot be appropriately provided when there is an interaction between the advice link and the event collector link. The causes for this limitation are two. In first place, the hookset associated to the advice link includes the check of whether or not the current operation is in the control flow of the event collector. Since the event collector link also matches the same operation occurrence, the result of such a check depends on the execution of the event collector. However, since hooksets are evaluated prior the reification of all links, and hence prior to the execution of the event collector, this dependency cannot be correctly specified. In second place, the semantic difference between `cflow` and `cflowbelow` cannot be achieved, because link composition specification is not supported by the Kernel machine. See Section 5.3.3.1 for a detailed description of this limitation of the compilation process.

Advice composition. Neither the PA language nor the Kernel language are designed to provide support for composition specification. Both the PA machine and the Kernel machine provide a built-in strategy for composing advices and links, respectively, when an interaction is detected. However, those two strategies were not designed to be compatible. Therefore, when multiple advices match a join point, the order in which they are executed by the PA machine and the Kernel machine may differ.

5.4 Compiling AspectJ to Reflex

The compilation process for the AspectJ language essentially involves the same activities that the one for the PA language. In order to simplify the compiler implementation, only some of the PCDs included in the PA language are supported, i.e. `call`, `execute`, `within`, `args`, `this`, `target`, `cflow` and `cflowbelow`. The three advice kinds supported by the PA language are supported[9], besides the *aspect* construction present in AspectJ. Only singleton aspects are supported. Since Reflex supports the specification of composition rules and does not have the implicit dependency limitation, the control flow PCDs shall be completely supported. In addition, AspectJ inner-aspect precedence rules shall be also supported by the generation of the corresponding advice composition rules. Finally, access to the `thisJoinPoint` and `thisJoinPointStaticPart` implicit variables from the advice's body is also supported.

The presentation of the compiler for AspectJ shall be done following the same structure used for the PA language. Since both compilers are very similar, only the main differences between the two shall be mentioned.

5.4.1 Conceptual Overview Reviewed

Conceptually, the correspondence between AspectJ constructions and Reflex constructions is analogous to the correspondence between the PA-Kernel languages. The main differences relies in how pointcuts can be efficiently implemented in a compiled environment and in that AspectJ has the notion of aspect rather than isolated advices.

5.4.1.1 Pointcuts

As shown in Section 2.6.3.1, pointcut designators may impose statically- and dynamically-matched restrictions over join points. In the AspectJ terminology, statically-matched restrictions are responsible for the selection of the join point shadows matched by the pointcut, meanwhile dynamically-matched restrictions are evaluated at runtime by residues [HH04]. In order to achieve efficiency, the mapping must make explicit the difference between static restrictions, which must be matched at compile-time, and dynamic restriction.

Let's us first consider static restrictions, with the simple user-defined pointcut designator `move` on Listing 5.14[10]. Note that the `call` PCD can entirely be matched statically, based on the program text. In other words, it is fully determined by its shadow. In Reflex vocabulary, a join point shadow is a hook. By extension, a pointcut shadow is a hookset. Therefore the pointcut `move` is mapped to a primitive hookset defined over the `MsgSend` operation, with a class selector that selects all classes and an operation selector that only selects the methods with the signature of `moveXY`.

[9]However, the *after returning* and *after throwing* advice kinds included in the AspectJ language, are not supported.

[10]Note that we slightly change the pointcuts in the example, with respect to the ones in Listing 5.1, in order to better illustrate residues. Now they select the invocation (join point kind `call`) of

```
pointcut move(): call(void *.moveXY(int, int))

pointcut movePoint(): move() && target(Point)

pointcut moveSinglePoint(): movePoint() && !cflowbelow(move())

pointcut moveSinglePointArgs(int x, int y):
            moveSinglePoint() && args(x, y)
```

Listing 5.14: AspectJ-Reflex Pointcut Examples <AspectJ Code>

Now consider the pointcut `movePoint`. It adds the restriction that the target object of the call to `moveXY` must be of type `Point`. Such a restriction cannot be completely resolved statically (although a static analysis of concrete types could partially help). In the program text, it is only possible to select calls that are done on an object whose declared type is either `Point`, or any super or sub-class of it. Since the concrete type of a variable is only determined at runtime, this pointcut requires a residue that dynamically checks the type of the target of a call (via `instanceOf`). In Reflex, residues are represented by associating a *restriction* (see Section 2.4.2.5) to the hookset. Therefore, `movePoint` is mapped with a hookset, determining the pointcut shadow, and a restriction which receives the target object and checks if it is instance of the `Point` class. Note that the hookset for the `movePoint` is the same of that of the pointcut `move`, the only thing that changes are the dynamic restrictions defined over it.

Control flow relations are handled as presented in Section 5.2.1, but now, a restriction is responsible for checking that the join point is inside the control flow of another pointcut. So, back in the example, the pointcut `moveSinglePoint` is represented by the hookset of the pointcut `move` plus a restriction that checks the two residues, `instanceOf` and `cflow`.

Context information is exposed using *call descriptors* (see Section 2.4.2.5). In the example, the pointcut `moveSinglePointArgs` requires a call descriptor specifying that the two parameters of method `moveXY` must be passed to the metaobject. This is done by getting the `Parameter` objects from the `MsgSend` parameter pool.

Actually, restrictions and call descriptors were not present in earlier versions of the Reflex model. Without them, the mapping requires that all the join point information be reified to the metaobject, which was responsible for checking the residues, extracting the required parameters and performing the advice's behavior. Clearly, this was far from being efficient, besides obfuscating the mapping. Therefore, in [RTN04] we introduce these two small but effective extensions to the Reflex model, which allows

---

the method `moveXY` on stand-alone points.

to achieve a more efficient and clear mapping.

5.4.1.2 Aspects

An aspect may contain any legal Java class member, plus multiple advices. It is mapped as a metaobject class, containing the same Java class members of the aspect and one additional method for each advice. Since we only support singleton aspects, the metaobject class follows the singleton pattern [GHJV95]. For instance, the aspect `MovingPoint` defined in Listing 5.15 is mapped to a class, named `MovingPointMO`, with one method with the same parameters that the `after` advice has. Since advices are nameless, a name is generated for them. Figure 5.6 shows a graphical illustration of the mapping.

```
aspect MovingPoint {
  after(int x, int y): moveSinglePointArgs(x, y) {
    System.out.println("Point moved: " + x + ", " + y);
  }
}
```

Listing 5.15: AspectJ-Reflex Aspect Example <AspectJ Code>

As for the PA-Kernel mapping, the binding is performing using a link. The link binds the hookset corresponding to `move` pointcut, i.e. `moveHookset` in Figure 5.6, to the aspect metaobject, using `AFTER` control attribute. It also associates a restriction to check the two residues required by the pointcut `moveSinglePointArgs` and a call descriptor that specifies the advice method to be invoked along with the parameters to reify. The event collector metaobject is much like the one of the PA-Kernel mapping.

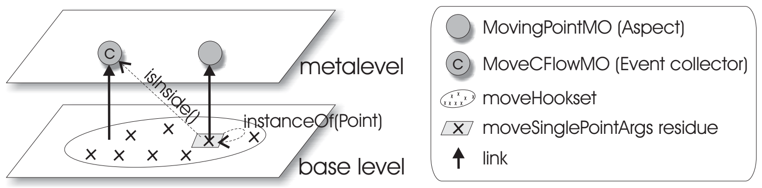


Figure 5.6: AspectJ-Reflex: Conceptual Mapping

Note that in the Figure, both metaobjects are linked to the same hookset. The only difference relies in the restrictions imposed for the `MovingPointMO` to check the residues.

5.4.2 Compilation Reviewed

The compilation process for AspectJ involves the same conceptual activities as the one for the PA language presented in Section 5.3. The main differences are related to how the pointcuts and aspects (including advices) are compiled. These differences shall be reviewed in the rest of this section. In addition, note that the output of the compilation process is a set of Java classes, which includes: class selectors, operation selectors, aspect classes, event collector classes and a Reflex configuration class (see Section 2.4.2.2). The *configuration* class shall include the definitions of hooksets, links, hookset restrictions and call descriptors.

Pointcuts The pointcuts are represented using a structure similar to the `RTree Trip`, but with the difference that the list of restrictions is divided in two lists: one for static restrictions and one for dynamic restrictions. Therefore, during the construction of the structure each PCD is expressed in terms of four elements, namely a kind restriction, a list of static restrictions, a list of dynamic restrictions and a list of context exposure requirements. This four-element representation of a PCD is called a *quadruple*, conversely to the triplet used for the PA language.

The only difference between the pointcut reduction process presented in Section 5.3.1.2 and the one used here is in how the quadruples are negated and composed. The complement of a quadruple of the form $(\mathtt{k}_i,\ \{\mathtt{r}^s_1,..,\mathtt{r}^s_n\},\ \{\mathtt{r}^d_1,..,\mathtt{r}^d_m\},\{\})$ now must select those operations which are not of kind $\mathtt{k}_i$ and those operations which are of kind $\mathtt{k}_i$ that do not fulfill one of the static restrictions $\{\mathtt{r}^s_1,..,\mathtt{r}^s_n\}$ or one of the dynamic restrictions $\{\mathtt{r}^d_1,..,\mathtt{r}^d_m\}$. The difference between the composition of the triplets and the composition of quadruples is that now, static and dynamic restrictions must be composed separately. The rest of the reduction process is exactly the same as that of the PA language.

Once the intermediate structure has been reduced, a primitive hookset and a hookset restriction (for the residues) are generated for each quadruple. The primitive hookset is generated based on the kind and the list of static restrictions of the quadruple. A class selector and an operation selector are generated for each hookset. The class selector encloses those parts of the static restrictions related to the class where the operation occurs, while the operation selector encloses the rest of the static restrictions. The remaining list of dynamic restrictions is checked using a hookset restriction. The hookset restriction is generated by creating a Java static method, which receives as parameters those pieces of information over which the dynamic restrictions are established (i.e. *this* object, target object or an argument), and whose body encloses the expressions to check the dynamic restrictions. In addition, the hookset restrictions are bound to the associated hookset and appropriately configured to receive the required parameters using the call descriptor mechanism (see Section 2.4.2.5).

Aspects An aspect is compiled into a Java class implementing the interface `BMeta-object` (see Section 2.4.2.6) and containing all the standard Java members defined in the aspect. The advices are compiled in the same way as for the PA language, thus the aspect class contains an additional method for each advice. If there is at least one *around* advice that uses the `proceed` statement, the aspect class must also extend the `ProceedMO` class[11] (see Section 2.4.2.6). The `proceed` statement is compiled by: generating an additional method in the aspect class which receives those parameters to the underlying advice and replacing the `proceed` statement by an invocation to that method. The mapping specification is represented by an array of integers, like for the PA language (see Section 5.3.2.2), and passed to the around advice as an additional parameter. The method generated for the `proceed` also receives the mapping specification, and is responsible for modifying the operation occurrence (based on the mapping), performing the boxing of the parameters (if they are primitive) and invoking the `proceed` method of Reflex (see Section 2.4.2.6).

For each defined aspect, AspectJ provides the static method `aspectOf` which allows to obtain a reference to the aspect. In order to support it, the `aspectOf` method is added to the generated aspect class and, once executed, it returns the only instance available for the aspect.

Link Definition The definition of the link for an advice requires creating a composite hookset containing the union of all the primitive hooksets corresponding to the associated pointcut (note that a link in Reflex is defined over only one hookset) and specifying how to obtain the associated metaobject. The metaobject is obtained by querying a metaobject factory (see Section 2.4.2.4) which in turn invokes the `aspectOf` method of the metaobject class to obtain the reference. In addition, the appropriate control and scope must be specified. The control is specified based on the advice kind, while the scope is always `GLOBAL`. In addition to this, a call descriptor must be created for each primitive hookset specifying the name of the advice method and the parameters that must be passed (obtained from the operation parameter pool).

The control flow restrictions are implemented as explained in Section 5.3.3.1, with the difference that the `isInside` method is now checked from the hookset restriction and not from the hookset. Section 5.4.3.1 explains how the composition rules are defined in order to distinguish between `cflow` and `cflowbelow` restrictions.

[11]Note that this implies that the aspect cannot extend another aspect or class. This limitation is resolved in a later version of Reflex, which does not require that the aspect class extend the `ProceedMO` class.

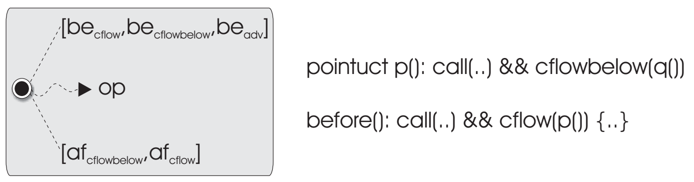
5.4.3 Advanced Features

5.4.3.1 Composition

The compiler also generates the composition specification to differentiate between `cflow` and `cflowbelow` restrictions, and establish how the links of the advices in a single aspect must be composed, in order to respect the AspectJ semantics.

As explained in Section 5.3.3.1 the semantics of the `cflow` PCD states that the event collector link ($\texttt{L}_{cflow}$) must enclose (*wrap*) the execution of the link of the affected advice ($\texttt{L}_{adv}$), consequently considering the current operation occurrence as a candidate for the selection. This composition semantics is specified using the composition rule $\texttt{wrap(L}_{cflow}\texttt{, L}_{adv}\texttt{)}$, as presented in Section 2.4.3. On the other hand, the `cflowbelow` PCD requires the opposite specification, the $\texttt{L}_{adv}$ link must wrap the $\texttt{L}_{cflow}$ link, in order to exclude the current operation occurrence from the selection, thus it is specified using the composition rule $\texttt{wrap(L}_{adv}\texttt{, L}_{cflow}\texttt{)}$. Therefore, the compiler shall generate the appropriate `wrap` rule for each pair of links related by a control flow restriction.

Since the $\texttt{L}_{cflow}$ link may also depend on other control flow restrictions, the composition specification must also define how the $\texttt{L}_{adv}$ link must be composed with those other links, over which it indirectly depends on. For instance, consider the pointcut definition shown in Figure 5.7, where the nested pointcut in the advice definition is affected by a `cflowbelow` restriction. The figure also illustrates how the three participant links must be composed in the case an interaction is detected. The composition specification includes the definition of the rules: $\texttt{wrap(L}_{cflow}\texttt{, L}_{cflowbelow}\texttt{)}$ to compose the two event collector links and $\texttt{wrap(\{L}_{cflow}\texttt{, L}_{cflowbelow}\texttt{\}, L}_{adv}\texttt{)}$ to compose those two links with the advice link. Note that the advice link is composed in the same way (i.e. using the same rule) with: the links from which it directly depends (i.e. $\texttt{L}_{cflow}$) and the links over which it indirectly depends (i.e. $\texttt{L}_{cflowbelow}$).


$[be_{cflow}, be_{cflowbelow}, be_{adv}]$
op
$[af_{cflowbelow}, af_{cflow}]$

```
pointuct p(): call(..) && cflowbelow(q())

before(): call(..) && cflow(p()) {..}
```

Figure 5.7: Multiple Nested Control Flow Restrictions

Reflex defines the notion of *linkset* [Tan04] to represent a set of links which are conceptually related. In addition, it allows to define composition rules between two linksets, meaning that the rule is applied distributively between the links in both sets, or between a link and a linkset, meaning that the rule is applied between the link and each link in the linkset. The linkset enhances the level of abstraction in which composition can be specified, since the entities of the aspect language can be encapsulated in linksets, consisting in one link or several links (i.e. the `cflow` restriction of Figure 5.7), and the composition can be specified in terms of aspect language entities (represented through linksets) rather that individual links. For instance, the `cflow` restriction in the figure can be encapsulated in the linkset $\mathtt{L}^{s}_{cflow}$ = {$\mathtt{L}_{cflow}$, $\mathtt{L}_{cflowbelow}$}, and composed with the advice link using the rule `wrap(`$\mathtt{L}^{s}_{cflow}$`, `$\mathtt{L}_{adv}$`)`.

In the cases where more than one advice is defined inside a single aspect and two or more of those advices match a join point, AspectJ semantics establishes that the advice that appears first lexically inside the aspect must execute first [Lad03]. For instance, Listing 5.16 shows part of the definition of an aspect that includes five advices, all of them defined over the same pointcut, and also shows the order in which they are expected to be executed. The first *before* advice is followed by the *around* advice, which in turn is followed by the second *after* advice, due to their lexical ordering. The second *before* advice must be executed after the *around* advice, but before the current join point, because of the nature of the *before* advice, thus it is nested inside the *around* advice. Likewise, the first *after* advice must be executed before the *around* advice, but after the current join point, thus it is also nested inside the *around* advice.

```
after(): somePointcut(){af1}              be1
before(): somePointcut(){be1}             ar1
around(): somePointcut(){ar1}     ⟹         be2 jp
before(): somePointcut(){be2}               af1
after(): somePointcut(){af2}              af2
```

Listing 5.16: Advice Execution Order

In order to specify the desired composition behavior for advices, composition rules must be defined between each pair of advices defined in the same aspect. Note that it is not possible to know in advance which advice links will interact, hence the complete specification must be defined. In order to illustrate the required composition specification, consider an aspect definition containing `n` advices, where ($\mathtt{L}^{s}_{1}$, . . , $\mathtt{L}^{s}_{n}$) is the list of linksets of those advices respecting the lexical order of definition. Therefore, for each pair ($\mathtt{L}^{s}_{i}$, $\mathtt{L}^{s}_{j}$), `i < j` means that the advice `i` is lexically declared before the advice `j`. Each linkset includes any event collector link over which the advice depends on (directly or indirectly), besides the advice link itself. The composition specification consists on the following rules:

```
∀ $L_i^s$, $L_j^s$ | 1 <= i < n and i < j <= n
    if (isAfter $L_i^s$) or (isAfter $L_j^s$) ⟶ wrap($L_j^s$, $L_i^s$)
    otherwise ⟶ wrap($L_i^s$, $L_j^s$)
```

Where `isAfter` determines if the advice link enclosed in the given linkset has `AFTER` control. In order to grasp the underlying intuition behind those rules, consider how they are applied to the following cases:

- There are two *before* advices. There is only one rule, wrap(L_1^s, L_2^s), which states that the first advice must be executed first.
- There are two *around* advices. There is only one rule, wrap(L_1^s, L_2^s), which states that the first advice must nest the second, thus L_1^s is executed before L_2^s.
- There are two *after* advices. There is only one rule, wrap(L_2^s, L_1^s), that wraps the first advice inside the second, thus the first declared *after* advice executes first.
- Consider the example at Listing 5.16, the *before* advice declared after the *around* advice shall be nested inside the *around* advice. The *after* advice declared before the *around* advice also gets nested inside it. Meanwhile the *before* and *after* advice declared before and after the *around* advice respectively, wrap the execution of the *around* advice.

Finally, there is an inefficiency issue related to the compilation of the control flow restriction that is worth being discussed. The issue occurs when there are more than one advice affected by a control flow restriction over the same pointcut. In order to illustrate this, consider the aspect definition shown at the bottom of the Figure 5.8. Note that both advices include a control flow restriction over the pointcut `q()`. In the current compiler implementation, two independent event collector metaobjects are defined, one for each control flow restriction, and both collecting the same information; besides defining the appropriate composition rules. This is illustrated by the reification tree at the left side of Figure 5.8, which exemplifies the scenario where the four links involved are interacting upon a single operation occurrence. This implementation faithfully represents the semantics of the aspect definition, however it is not the optimal implementation.

The optimal implementation should have only one event collector exposing two methods: `isInside`, which considers all the collected join points in order to respond (i.e. including the *current join point*), and `isInsideBelow`, which does not consider the *current join point* in order to respond. By current join point we mean the join point from where the query to the event collector is performed. The optimal implementation is illustrated by the reification tree in the middle of Figure 5.8. Note that the

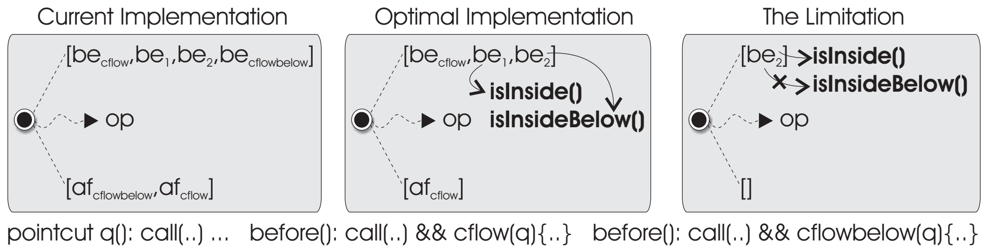

Current Implementation
[be_cflow, be_1, be_2, be_cflowbelow]
op
[af_cflowbelow, af_cflow]
pointcut q(): call(..) ...
Optimal Implementation
[be_cflow, be_1, be_2]
isInside()
op
isInsideBelow()
[af_cflow]
before(): call(..) && cflow(q){..}
The Limitation
[be_2]
isInside()
isInsideBelow()
op
[]
before(): call(..) && cflowbelow(q){..}

Figure 5.8: Control Flow Efficiency Issue

event collector wraps the execution of the two advice links hence capturing the join point before the restrictions of both links are evaluated. In the interaction scenario where the four links are present, the `isInsideBelow` method could be implemented as `(counter - 1) < 0`, thus it does not consider the current join point. However, such an implementation is not correct, since in the scenario where the $\texttt{be}_2$ link element is not interacting with the event collector link (shown by the tree at the right side of Figure 5.8), the `isInsideBelow` method still substracts one to the counter. Note that the event collector does not have to exclude the current join point, since such a join point was not matched by its associated pointcut. Therefore, in order to solve this problem there are (at lest) two alternatives. The first relies on implementing the `isInsideBelow` is such a way that it can check if the current join point is part of its collected join points, in order to determine if it must be excluded or not. This requires some sort of *hook reification*[12] facility, not supported by Reflex, which allows to check if the control flow link is present in the hook that is invoking `isInsideBelow`. The second alternative uses different restrictions, for different interaction scenarios. For instance as it is illustrated in tree at the right side of Figure 5.8, when $\texttt{be}_1$ is not interacting with the control flow link it could use the `isInside` method instead of the `isInsideBelow`. However, this solution is also not supported by Reflex, since it does not support the definition of different restrictions for different interaction scenarios, in other words, only one restriction can be attached to a link and it must be the same for all the scenarios. The study of this inefficiency issue and its solution is left as future work.

[12]Remember, the hook is the piece of code responsible for performing the reification. When more than one link applies for the same program point, the hook performs the reification for all of them, respecting the composition specification.

5.4.3.2 Join Point Reflective Information

AspectJ provides three implicit references which provide reflective information about the current join point, which can be used from the body of any advice. These three implicit references are:

- `thisJoinPoint` provides reflective information about the static part (i.e., the signature of the join point) and the dynamic part (e.g., the values of the arguments) of the current join point.
- `thisJoinPointStaticPart` is similar to `thisJoinPoint` but only provides reflective information about the static part of the current join point.
- `thisEnclosingJoinPointStaticPart` provides reflective information about the static part of the *enclosing* join point, that is to say, the method (or constructor, or advice, or static initializer) in which the current join point has occurred.

These instances essentially collect all the current join point context information, plus the reflective static information describing the actual signature of the join point and expose it by implementing a set of interfaces defined by AspectJ.

As seen in Section 2.4.2.5, parameters specified in MOP descriptors give access to context information of a hook, thus making handling the dynamic part of `thisJoinPoint` easy. In addition to this, when parameters are resolved at generation time, they have access to the static information of operation occurrences, which includes where the occurrence is located, thereby making it possible to compute `thisJoinPointStaticPart` and `thisEnclosingJoinPointStaticPart`.

Therefore, providing access to these variables in the context of the compiler simply implies providing (1) a set of classes that implement the interfaces defined by AspectJ, and (2) three parameters (one for each instance) that instantiate such classes with the necessary information. These parameters are passed as additional parameters to the advice body, exactly as is done for AspectJ in [HH04], without having to transform advice bodies. The actual implementation of the compiler only supports the `thisJoinPoint` and `thisJoinPointStaticPart` implicit references.

5.4.4 Limitations Reviewed

In this Section we will review the limitations mentioned in Section 5.3.4, since some of them have been overcome in the context of the compiler from AspectJ to Reflex.

Visibility of the pointcut predicate. The limitation regarding the synthetic code generated by the compiler still remains. However, here we shall analyze some features of Reflex that may help in solving it in the near future.

Reflex provides a mechanism to exclude entire classes from the BLS phase (see Section 2.4.3), that is to say, those classes are invisible for all links defined in the kernel. This mechanism can partially help in hiding the synthetic code generated during the compilation process. For instance, operation occurrences inside event collector classes can be made invisible for the pointcut predicates. However, this mechanism is too coarse-grained in order to deal with, for instance the primitive type boxing and unboxing expressions included in the compilation of around advices.

In Reflex, the load-time reflective API (e.g. the one used by selectors in order to introspect the base level code) is based on interfaces. Such a design decision allows to provide implementations of the API that are "Reflex aware" in the sense that they hide synthetic members and expressions generated by Reflex [Tan04]. For instance, Reflex can ensure that modifications done during both application of structural links and installation of behavioral links are not seen by other links. This mechanism used by Reflex to hide its own synthetic code, seems to be a promising mechanism to be reused for hiding the synthetic code generated the compilation of aspect-oriented languages (AOLs), like AspectJ. However, the current implementation of Reflex only offers a limited possibility of customization for such an API, which does not fulfill the needs of our AspectJ compiler.

In [Tan05] the author studies the design of metalevel facilities (e.g. reflective APIs) in the context of the Reflex AOP kernel. He motivates the need for having a reflective API for all the languages involves in the AOP kernel, e.g. AOLs, Kernel language and the Java language. In addition, he brings to the fore the importance of using a mirror-based approach in order to design reflective APIs. Mirrors [BU04] provide a principled way of designing the APIs. In [BU04] the authors suggest that encapsulation and stratification are two important principles for the design of such APIs. Encapsulation suggests that their implementation must be encapsulated behind interfaces. Stratification suggests that the reflective APIs must be separate from base-level functionality (in contrast to approaches like Java, where the access to reflective entities is provided by base entities themselves, e.g. `o.getClass()`). Together, these two principles allow to provide different implementations of the reflective APIs for different usage scenarios of those APIs. Therefore, offering an interesting framework for the definition of a high-customizable reflective API for hiding synthetic code.

Control flow. The control flow restrictions are now fully supported, including those cases where the advice link and the event collector link are interacting upon the occurrence of an operation. Since the control flow residue is implemented as a restriction, and so it is outside the hookset, it does not affect the interaction detection and resolution. Therefore, upon the static occurrence of the operation both links are matched (through their hooksets) and composed appropriately. Then, upon the runtime occurrence of the operation, the event collector gets executed first [13], hence collecting the operation occurrence, and then the advice's restriction is evaluated. As a result, the dependency that exists between the two links is correctly implemented, in contrast to the compiler$_{PA-Kernel}$ which cannot do this due to the implicit dependency limitation of the Kernel machine (see Section 3.3.7.6). In addition, Reflex provides support for link composition specification, thus the `cflow` and `cflowbelow` PCD are appropriately compiled by the definition of the corresponding composition rules, as explained in Section 5.4.3.1.

Advice precedence. The advice precedence is now partially supported, because the compiler generates the composition rules required to represent the inner-aspect precedence rules, as defined by AspectJ. However, if an interaction between advices defined in different aspects occurs, the order in which they are executed may differ. In order to also overcome inter-aspect advice composition issue, support for the AspectJ `declare precedence` construct [Lad03] must be included. Extending the compiler to support such a construct should be easy, but it is left as future work.

5.5 Implementation of the AspectJ Plugin

The AspectJ Plugin (AJP) is a plugin for the Reflex AOP kernel that implements the compilation process previously described, along with the necessary components to execute a compiled AspectJ program in the kernel. The plugin also experiments in providing metalevel facilities to introspect an AspectJ program in the kernel (see Section 5.5.3).

5.5.1 Reflex Plugins Architecture

The plugin architecture of Reflex [Tan04] is meant to bridge the abstraction gap between the aspect languages and the kernel core constructions. In order to add support for a new AO language (domain specific or general purpose), a plugin must be provided to translate those constructions of the AO language into low-level kernel constructions (e.g. metaobjects, links, hookset, composition rules).

As illustrated in Figure 5.9, the different plugins allow the aspect programmer to implement the aspects in the language that best suite his needs, without knowing about the low-level kernel constructions used to implement them. In addition to the generation of the low-level constructions, the plugin also generates a linkset containing the set of links required to represents each defined aspect[14]. For instance, in

[13]Note that we are assuming a `cflow` restriction.

[14]Additional linksets may also be defined to represents lower-level constructions of the aspect

Figure 5.9 linkset `A` represents the two B-Links required to represent aspect `A`, while linkset `C` represents the B-Link and S-Link required to represent aspect `C`. Specially tailored languages can be designed in order to specify the composition of heterogeneous aspect languages (e.g. the `P3` plugin's language), expressing the composition in terms of linksets, thus being obliviouss to the particular implementation details of the composed languages.

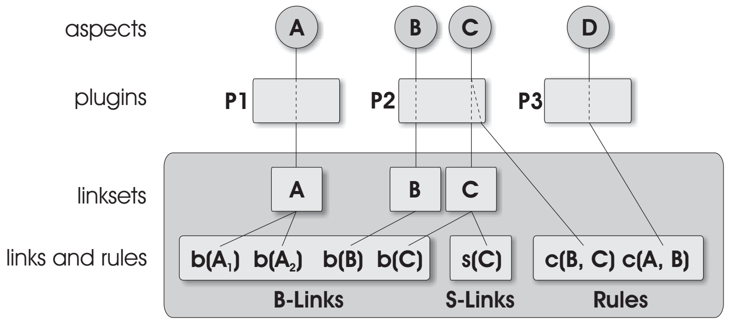


Figure 5.9: Reflex Plugins Architecture

The plugins can work in two different modes: offline and online. The offline mode is meant to perform activities which are too expensive to be done on each execution of the kernel. For instance, the compilation of an AspectJ program involves the generation of several implementation elements, like metaobject classes, selector classes, event collector classes, configuration files. The generation of those elements is a typical activity to be done offline. The online mode is meant to configure the Kernel to execute the aspect program.

5.5.2 AspectJ Plugin Architecture

The compilation process that transforms the AspectJ program into a Reflex program is performed offline. Interestingly, the compiler is implemented by using Reflex to transparently change the behavior of the standard AspectJ Compiler (AJC) so that the intermediate representation is transformed to a Reflex program. Figure 5.10 illustrates the metalevel architecture used in the design of the plugin. At the base level resides the compiler, responsible for: parsing the AspectJ program, performing the semantic analysis and transforming the program into a Reflex program. Both the parsing and the semantic analysis are delegated to the AJC. At the metalevel resides

language, e.g. advices in AspectJ.

the *AspectJ Bind* component, responsible for synchronizing the AST structure built by the AJC with the *Intermediate Model* component at the base level.

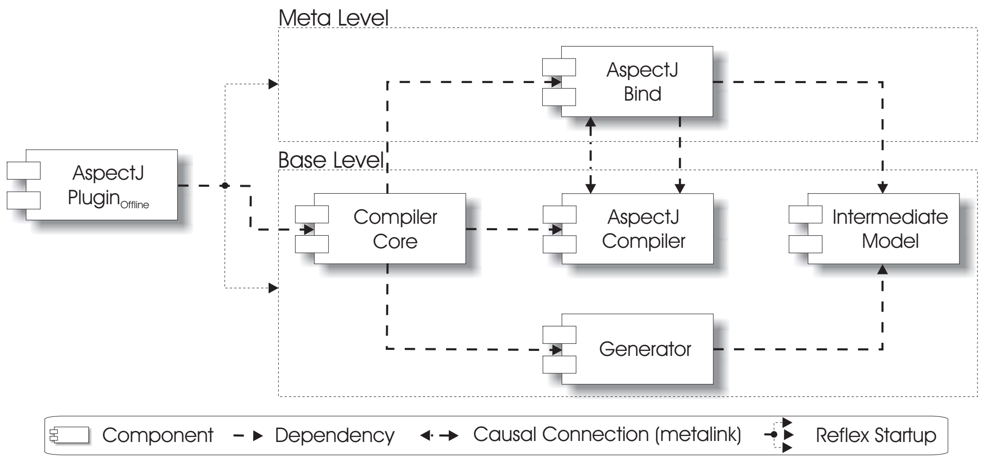


Figure 5.10: AspectJ Plugin Architecture (Offline mode)

Once the *AspectJ Plugin* component is executed, it sets up the metaobjects of the *AspectJ Bind* component and executes the *Compiler Core*. Those metaobjects are configured to reify some key points of the AJC compilation process, in order to obtain the AST structures corresponding to the aspects defined in the program, check if the compilation ends successfully or ends with errors, and stop the AJC from actually performing the weaving of the AspectJ program. The *Compiler Core* component coordinates the compilation process, which consists in invoking the AJC to parse the programs, checking possible errors that may happen during the AJC execution, invoking the *AspectJ Bind* component to generate the intermediate model and invoking the *Generator* to create the resultant Reflex program.

The online mode of the AJP is responsible for setting up the program in the kernel for its execution. In addition, it generates all the required information for supporting the reflective API provided by the plugin, explained in the next section.

5.5.3 Plugin Reflective API

The AJP provides a reflective API for the AspectJ language, offering introspection capabilities for programs compiled through the plugin. Through the API, the program entities are accessed using the appropriate high-level constructions (e.g. aspect, advices), as opposed to the low-level Kernel constructions (e.g. metaobjects, links) used for implementing those entities in the Kernel. Furthermore, the API also provides access to the linksets used to represent those aspects or advices in the Kernel, hence providing traceability between those high-level constructions and the Kernel constructions.

One interesting use of the API is composition of heterogeneous aspect languages. For instance, consider that the aspect `D` of the plugin `P3`, shown in Figure 5.9, specifies the composition in terms of explicit references to the aspect-level constructions defined in the languages `A` and `B`. On the one hand, if `P1` and `P2` do not have a reflective API for their language, they usually end up defining their own convention for naming the linksets that represent those aspect-level constructions. `P3` must be aware of those conventions in order to find those linksets and use them to specify the required composition rules. Furthermore, if both plugins do provide a reflective API, the `P3` plugin, making use of both reflective APIs, can obtain a reification of those aspect-level constructions mentioned by `D`, navigating through aspect language constructions (as opposite to Kernel constructions). Once `P3` finds the constructions mentioned by `D`, it can access their associated linksets and use them to specify composition.

The AJP reflective API consists of a collection of interfaces that encapsulates the most relevant concepts of the language. Listing 5.17 illustrates an example of the use of the API. Firstly, a reference to the `AspectJAPI` is obtained by querying the Kernel to get the plugin's API and using it to get the reference to the reflective API. The reification of an aspect construction can be obtained based on the aspect name (i.e. using the `getAspect` method) or by iterating over all the defined aspects (not shown here). The `Aspect` interface allows, for instance, to obtain a reference to the aspect object (similar to the `aspectOf` method in AspectJ) or getting a list of its defined advices. The `Advice` interface allows, for instance, to access the kind of the advice or its declared parameters (neither shown here). The last two lines of Listing 5.17 show how to access the linkset associated to an aspect or an advice. In addition, if the advice involves control flow restrictions, the `Advice` interface also provides means to introspect them.

```
AJPAPI ajp = (AJPAPI) API.plugins().getPluginAPI("AJP");
AspectJAPI theAPI = ajp.languageAPI();

Aspect theFooAspect = theAPI.getAspect("Foo");
Foo theAspect = (Foo) theFooAspect.getAspectObject();
Advice theFstAdvice = theFooAspect.getAdvices().get(0);

Linkset theFooLinkset = theFooAspect.getLinkset();
Linkset theFstAdvLinkset = theFstAdvice.getLinkset();
```

Listing 5.17: AspectJ Reflective API <Java Code>

The current reflective API of the AJP is limited. It is meant to be only a proof of concept of the relevance of having a reflective API for the aspect languages. There are many issues regarding the API definition that must be reviewed:

- The current API intermixes the constructions of AspectJ with the Kernel constructions used for its implementation (e.g. links, linksets). A redesign of the API following the guidelines presented in [Tan05], based on the notion of mirrors, will enhance its clarity. The API should be divided in two APIs: one encapsulating only AspectJ language constructions, while the other allows to reify those constructions in terms of the Kernel constructions. This latter API provides cross-language facilities, that is to say, it provides a view of the reification of a construct of one language (i.e. the aspect language), but expressed in terms how it is implemented in another language (i.e. the Kernel language).

- Providing fine-grained introspection capabilities for both APIs. For instance, the AspectJ language API could include pointcut introspection capabilities, while the cross-language API could provide access to each of the Kernel constructions used to implement the different parts of the pointcut.

- Extending it with intercessive facilities (see Section 2.3.2.1) of the AspectJ API could be interesting as well as the study of the applications of such new possibilities.

5.5.4 Preliminary Benchmarks

We report on the performance of programs compiled using the AJC (`ajc` 1.2) and programs translated by the AJP. The benchmarks were run on a Pentium III 1.1GHz with 512MB of memory, running Windows XP and Java 1.4.2_05 (HotSpot client VM). We allocated a large heap size in order to limit the number of garbage collections. We benchmarked specific features of AspectJ's dynamic crosscutting in programs based on a simple shape editor example as presented in Section 2.4.2.1, applying a logging aspect in different scenarios (Table 5.1). The two first sets of test cases consist in applying a before advice with and without context information, respectively. For each, three features are tested: first without residues, and then using an instanceOf residue and a cflow residue. When using residues, scenarios where residues match and do not match are measured. The last set of test cases is based on the use of an around advice, to which a composite shape and two integers are passed as parameters, in three different scenarios: *1)* the advice always calls `proceed`; *2)* the advice calls `proceed` half of the time, otherwise it reorders the shapes contained in the given composite shape; *3)* `proceed` is never called, the advice always performs the reordering.

Features	*Scenario*	*AJC*	*AJP*	Δ
before	**(w/o context)**			
no residue		1542	1705	10%
instanceOf	match	1185	1305	10%
	no match	868	894	2%
cflow	match	841	951	13%
	no match	998	1218	22%
before	**(w/ context)**			
no residue		4533	4616	1%
instanceOf	match	3241	3034	-6%
	no match	884	904	2%
cflow	match	10295	11102	7%
	no match	697	647	-7%
around	**(w/ context)**			
around	always proceed	3404	5721	68%
	half proceed	4416	6349	43%
	never proceed	5711	6990	22%

Table 5.1: Execution time (in ms) and overhead of the AspectJ plugin (AJP) vs. standard AspectJ (AJC).

The results were obtained by performing five measurements in each scenario, discarding the best and worst cases and taking the average of the remaining three measurements. The third and fourth columns in Table 5.1 show the average execution time using AJC and AJP, respectively. The last column shows the relative overhead of AJP.

The test cases related to the before advice show a very reasonable overhead of AJP (less than 10%), in particular when considering that our implementation is more prototypical than production quality. Test cases with context exposure demonstrate particularly good performance of AJP, even slightly better than AJC in some cases. The most important overhead (up to 22%) in the case of cflow without context seems to be due to better HotSpot optimizations for AJC, since AJP shows much better performance when running these scenarios in interpreted mode (up to 29% better than AJC).

Execution of the around advice however shows a significant (though still acceptable) overhead. This overhead is not mainly due to advice inlining done by AJC, since it is not significantly reduced when running the scenarios without inlining. This strongly suggests that the major cause of inefficiency is the use of standard reflective method invocations to implement `proceed`, whereas AJC generates specific stubs for each join point shadow [HH04]. These results motivated the development of a similar implementation strategy for the `proceed` in Reflex, which has been already integrated into Reflex [RFX]. However, such a development was carried on after this thesis thus the benchmarks for the new implementation strategy are not included here.

Chapter 6

Conclusions and Future Work

6.1 Summary and Conclusions

In this work, a formal development, using the Haskell language, of the Reflex model for partial behavioral reflection (PBR) has been presented. This development embodies the design of the Kernel language and its correspondent abstract execution machine. For the Kernel language, a dedicated syntax for the constructions of the PBR model has been designed, as opposed to the object-oriented syntax used for its implementation in Java [RFX]. This new syntax eases the presentation of the constructions in the model, along with enhancing its readability. The Kernel language and the corresponding machine have been formally written down in Haskell. This, we claim, has made it possible to express the semantics of the PBR model in a more abstract and simple way than it is achieved by the informal semantics described in [TNCC03, Tan04] and the current implementation in Java [RFX]. Furthermore, a conceptual explanation of the PBR model in terms of the reflective tower [Smi82] has been presented, which further clarifies the underlying concepts of the PBR model. In addition, the implementation of the Kernel machine has reported on a concrete experience of the development of a language containing behavioral reflection features, along with the description of the problems (i.e. bootstrapping and meta-stability issues, see Section 3.3.7.1) and their solution encountered during its development.

Not surprisingly, the PBR model is expressive enough to provide semantics for the kind of dynamic crosscutting offered by the AspectJ language, a reference amongst the existing AOP approaches. This is not surprising because conceptually, any form of behavioral reflection can cover dynamic crosscutting. The real challenge is to do so in a natural and efficient manner. In this direction, two compilers have been implemented. The compiler implemented within the Reflex Sandbox (RSB) has provided the foundations for understanding the transformation required in order to bridge the underlying abstraction gap between the AspectJ language and the Reflex model. The compiler implemented in Java has shown that the transformation is feasible in an industrial environment. This compiler has been implemented as a plugin for the Reflex's plugin architecture, the so-called AspectJ Plugin (AJP). Preliminary benchmarks have been done to compare the performance of programs compiled with the AspectJ Compiler [aspb] and programs compiled by the AJP. The results have shown that the performance of Reflex is close to that of AspectJ on dynamic crosscutting. These results are encouraging since the AJP is a proof of concept implementation, meanwhile the AspectJ Compiler is a production quality tool. In addition, this work deepens the understanding of the relation between reflection and AOP by showing how the essential part of AspectJ, an efficient AOP language implemented without resorting to reflection, could actually be supported by reflection in an efficient way.

Since the Kernel machine has been designed to only support part of the Reflex's PBR model, not every construction in the PA language can be compiled into the Kernel language. In particular, *link* composition specification and *link* dependency specification (see Section 3.3.7.6) have not been included in the Kernel machine. As a consequence, the control flow constructions of the PA language cannot be compiled under some particular conditions (i.e. when the control flow link is interacting with the advice link). On the other hand, the Java implementation of the compiler, transforming AspectJ programs into Reflex programs, does consider the whole Reflex model. Therefore, it supports the compilation of the control flow constructions under all circumstances. In addition, it also supports inner-aspect precedence rules, hence aspect composition is partially supported, and join point reflective information.

The development of the Kernel machine and the two compilers made it possible to identify three limitations of the Reflex model. The first limitation, presented in Sections 5.3.4 and 5.4.4, is the lack of an appropriate centralized mechanism to hide the synthetic code generated during the compilation of an aspect language into the AOP Kernel. As a consequence, each compiler needs to develop its own ad-hoc mechanism to hide synthetic code from its selection mechanism (e.g. pointcut in AspectJ). However, such an ad-hoc mechanism only partially solves the problem since other languages in the Kernel may still be seeing the code. The second and third limitation, both presented in Section 5.4.3.1, are the lack of a *hook* reification facility, which allows to introspect the links that are involved in a hook, and the impossibility to define hookset restrictions whose applicability depends on the link interaction scenario in which they are involved, respectively. Finding a solution for any of these two limitations would allow to overcome the inefficiency issues presented in Section 5.4.3.1, regarding the compilation of control flow restrictions. In addition, the results of the benchmarks presented in Section 5.5.4 have suggested that the `proceed` statement should be implemented without using the Java reflective API. In order to enhance its performance an approach like in [HH04], which is based on stub generations, should be adopted. This enhancement has been included in the current Reflex implementation available in [RFX].

In addition, in [RTN04] we have presented two small but effective extensions to the Reflex model, namely hookset restrictions and call descriptors. The hookset restrictions allow a better representation of AspectJ residues (see Section 2.6.3.1), since before their inclusion in the Reflex model, residues must be checked at the metalevel. This has the bad property of forcing reification even in cases where the residues reject a particular occurrence. And reification is a major source of overhead in reflective systems. The call descriptors allow to specify precisely which information must be passed to the metaobject. Without using call descriptors, all the operation occurrence information must be passed to the metaobject, which is highly inefficient.

Finally, the RSB has been developed, providing an environment for Reflex where theoretical studies can be carried out. In the RSB, new extensions to the Reflex model can be prototyped and their semantics exposed in an abstract and simple way. In addition, other aspect-oriented languages can be defined, and their correspondent execution machines implemented, within the RSB in order to study their compilation into the Kernel machine. Through the testing environment (see Appendix C) those execution machines and compilers can be tested to check if they behave as expected.

6.2 Related work

Historically, reflection is at the heart of AOP [KLM$^+$97], which can be seen as a disciplined and principled way of doing metaprogramming. [Sul01, KLLH03] are two works that study the relation between reflection and AOP. In [Sul01] the author presents a short conceptual description of how reflection can be used to support AOP. In [KLLH03] the authors explore the relation and interactions between reflection and AOP. They show the tradeoffs of implementing AOP with reflection and, reciprocally, to implement reflection via AOP. A result of this study is that a metaobject protocol should be expressive enough to enable AOP, while AOP makes it possible to apply reflection more selectively. We have gone one step further [Tan04, RTN04] by actually showing that an essential part of an efficient AOP language, AspectJ, implemented without resorting to reflection, could actually be supported by refection in an efficient way, thanks to the PBR model, which provides high-selectivity. Furthermore, we have implemented a compiler that automatically translates AspectJ programs into Reflex programs.

The Aspect Sandbox (ASB) [asb] was a project that aimed to provide concise models for AOP for theoretical studies, where the semantics of the models can be cleanly defined, and to provide a tool for prototyping alternative AOP semantics. The ASB essentially consists on a series of interpreters, characterizing different models for AOP, implemented in Scheme. Several works have been developed in the context of this project, for instance: [MKD03b] describes a common frame of reference in order to characterize the most fundamental AOP approaches, [MKD02, MKD03a] explain the weaving strategy for the core features of the AspectJ's dynamic crosscutting mechanism and [WKD04] presents a denotational semantics for the core constructions of AspectJ's dynamic crosscutting mechanism. The RSB shares the same motivations of the ASB. However, we aim to provide a concise model for an AOP kernel, where different AOP approaches can be supported. In addition, we are interested in studying how those different AOP approaches can be compiled into the AOP kernel, rather than studying how the weaving of an AOP approach can be achieved.

The Event-Based AOP (EAOP) project [DMS01, DFS02, DS02] also has the motivation of building a test-bed for AOP to study the expressiveness of the mechanisms provided for AO language definition. To that end, they propose a new model for AOP based on a monitor that pattern-matches a stream of events from the program execution, and applies aspect code when a match is detected. Using that model, in [DMS01] they propose a formal framework for AOP, using the Haskell language, in order to clearly expose the semantics of its AOP approach. In addition, they also present an implementation of that framework in the Java language. In [DT04] the authors have also used Haskell in order to express the formal semantics of an extension to the pointcut language of AspectJ for control-flow.

The AspectBench Compiler (*abc*) [ACH$^+$05] is an extensible AspectJ compiler that facilitates easy experimentation with new language features and implementation techniques. The compiler is designed with extensibility as its primary goal, and also aims for an optimized implementation of the AspectJ language and its extensions. It allows to handle a wide variety of extensions, including frontend scanner and parser, the type checker, the weaver, and potentially requiring sophisticated program analysis to ensure correctness and efficiency. The motivation of the *abc* compiler is to facilitate the research of the AOP design space, which is the same goal that motivates us to build the RSB. However, the RSB is oriented to the study of such a space by the construction of an AOP kernel supporting a wide range of AOP languages and language features. In addition, the RSB is advocated to a more formal study of those languages and features, by using a simplified environment that facilitates the definition of their semantics. At the Java level, and through the implementation of plugins for Reflex, we also take care of implementation techniques and efficiency, but we do not provide support for language compilation activities, like parsing or type checking. Those activities must be completely resolved by the plugin programmer. In Reflex, optimization techniques are also limited to the possibilities offered by the kernel. Josh [CN04] is an open-implementation of an AspectJ-like language, which is close in spirit to *abc*.

6.3 Future Work

An important topic for further work is experimenting with several aspect-oriented languages interacting in the AOP Kernel. This is a challenging and multi-faceted issue. One facet of this issue embodies the development of the compilers for those languages for which the RSB promise to be a tool of great value. Another interesting facet is related to the definition of a mechanism to specify the appropriate levels of isolation between the different languages. In [TN04a] the authors presents a first step in this direction by suggesting that an AOP kernel should have a *collaboration protocol* to control the visibility of *structural changes* among aspects. For instance, such a mechanism may allow hiding, from other aspects, all the methods that an aspect may add to a class. Such a mechanism is useful but yet limited. For instance, consider the synthetic code issue, presented in Section 5.3.4. Hiding synthetic code requires a fine-grained mechanism, since we cannot assume that such a code would always correspond to a method or a class, but can also be an expression. Such a mechanism must also be flexible, allowing to specify the code that is visible and the code that is not. As explained in [Tan05] and briefly mentioned in Section 5.4.4, an interesting approach to solve this issue is providing an extra level of indirection between the *selectors* and the actual operations occurring during the program execution. In the middle, there is a highly-customizable mechanism that allows to specify whether an operation is visible or not. As suggested in [Tan05], in Reflex for Java this mechanism could be implemented by the load-time reflective API that it uses. However, this facet must be further analyzed; in particular, it is not clear how the visibility requirements should be specified. Once more, the RSB offers an attractive reduced environment for studying this facet. Yet another interesting facet is to further experiment with defining reflective APIs for those languages, in order to have high-level views of the aspect-oriented programs. These APIs provide an interesting approach for the specification of composition between heterogeneous aspects (see Section 5.5.3).

Finally, in order to experiment with more complex environments, like the interaction of several aspect languages, the Kernel machine must be extended to support other features of the Reflex model. In particular, providing support for link composition, including interaction selectors and composition rules specification, is mandatory in order to further experiment on the support of aspect languages. The composition specification is a very complex and delicate topic. It embodies the algorithm that, given a composition specification and a set of links that apply, generates the correspondent reification tree. Since the user may specify rules that contradict other rules, it requires a mechanism to detect those contradictions and report them, as clearly as possible and hopefully at rule definition time (as opposed to rule application time), to the user. Therefore, providing a formal semantics for the composition specification would be very useful. In addition, the *implicit dependency* problem presented in Section 3.3.7.6, which causes that some links *come in* while others *come out* of the reification tree during its evaluation, must be further studied.

Appendix A

Introduction to Haskell

This appendix presents a short introduction to the Haskell language and to monadic programming using Haskell.

This appendix is organized as follows. Section A.1 presents the Haskell language and review its main constructions. Section A.2 introduces monadic programming and presents two main monads used in this thesis, namely the *exception* and the *exception and state* monads.

A.1 The Haskell Language

Haskell is a general purpose, purely functional programming language [Tho99, HF92]. By being purely functional, function evaluation is *side-effects* free. The language supports recursive functions and algebraic data types, as well as lazy evaluation.

Algebraic data types definition are introduced by the keyword `data` followed by the name of the type, an equal symbol and then the constructors of the type being defined. The name of both the type and the constructors must begin with capital letters. Constructors may receive parameters and they define a function that can be used to create instances of the defined type. The Listing A.1 shows the definition of a data type for a list of integers.

```
data ListInt = Nil |
               Cons Int ListInt
```

Listing A.1: Haskell Algebraic Data Types <Haskell Code>

The type definition provides two constructors, `Nil` and `Cons`. `Nil` is a constructor that returns an empty list. `Cons` is a constructor that given an integer and a list builds a new list. Notice that the type is defined using recursion. Haskell also incorporates *polymorphic types*, i.e. types that are universally qualified in some way over all types. This allows the definition of families of types, rather than a single type, which can be used later to define general functions over those families. For instance, we may define the family of types for lists of `a`, see Listing A.2.

```
data List a = Nil |
              Cons a (List a)

type ListInt = List Int
```

Listing A.2: Haskell Polymorphic Types <Haskell Code>

In the definition, the type name, `List`, is followed by a *type variable* which represents the type of the elements of the list. In the general case of polymorphic type definition, the name of the type may be followed by a list of type variables. Those type variables can be used in the type constructors, e.g. `Cons`. The declaration of the polymorphic type `List a` defines a *type constructor*, named `List`, which can be used to construct types of the family of types it defines. The last line of the Listing A.2 defines the type "list of integers", by fixing the type variable to the type `Int`. In addition, it uses the keyword `type` to define a synonym, named `ListInt`, for the type of lists of integer.

Functions in Haskell are usually defined by a series of equations, which in turn are defined using *pattern matching* and *guards* on the input values of the function. For instance, consider the factorial function in Listing A.3 defined using two equations. The first, using pattern matching, states the value of `fac 0`. The second, using pattern matching and a guard, states how the `fac n` is calculated recursively. Notice that the name of the functions must begin with a lower-case letter.

```
fac :: Integer -> Integer
fac 0 = 1
fac n | n>0 = n * fac (n-1)
```

Listing A.3: Haskell Function Definition <Haskell Code>

The first line in the example declares the type signature of the function. This declaration is not mandatory, since Haskell can infer its signature; but it is a good practice to include it for documentation purposes. The functions defined over polymorphic types are called *polymorphic functions*. Functions like `map`, `foldl` and `foldr` over lists that define standard recursion patterns are examples of polymorphic functions. The Listing A.4 shows the implementation of the `map` function. The polymorphic type for Lists in Haskell is specified using the syntax `[a]` to represent `List a`. The `[]` represents the empty list and the infix operator ":" to add its first argument to its second argument (a list). The list `1:(2:(3:[]))` can be specified using the shorthand `[1,2,3]`. The `map` function takes a function (from `a->b`) and a list (of `a`) as arguments and applies the function to all the elements in the list, returning a list of `b`.

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

inc n = n + 1

map inc [1,2,3] => [2,3,4]
```

Listing A.4: Haskell Polymorphic Function <Haskell Code>

`map` can be used over lists of any type `a`. For instance, in the example, it is used to increment all the elements of a list of integers. In addition, functions can be defined as anonymous, using the *lambda notation*, e.g. the `inc` can be defined as
`n -> n + 1`.

Haskell provides *modules* as an aid to modularization. A module consists of a number of definitions (e.g. types, functions), with a clearly defined interface stating what the module exports to other modules.

A.2 Monadic Programming

Monads came into the functional programming world as a convenient way to model side-effects in a purely functional programming language. In Haskell, one of the main applications of the concept of monad is to model the IO System, where interaction with the outside world is required.

A monad is a way to structure computations in terms of values and sequences of computations using those values. They are a useful tool for structuring functional programs, which is the precise reason why we will use them in the development of the machines. In order to introduce them we will present the development of the *exception monad* which encapsulates computations that may throw exceptions.

A monad is defined by the following triple `(M, return, >>=)` where `M` is a type constructor for the monad and, `return` and `>>=` are two functions defined over the monad. In the case of the exception monad, the type constructor for the monad is `Exc` as shown in Listing A.5. `Exc a` represents the computations that may return a value of type `a`, represented by the constructor `Val`, or that may end up with an error, represented by the constructor `Error`. The constructor `Error` encapsulate a `String` that describes the exception that occurs.

```
data Exc a = Error String |
             Val a

instance Monad Exc where
   return = Val
   (Error x) >>= _ = (Error x)
   (Val x) >>= f = f x

raiseE :: String -> Exc a
raiseE str = Error str
```

Listing A.5: Exception Monad <Haskell Code>

In Haskell, the functions that apply over the monadic computation are defined as a type class, called `Monad`. Therefore, any user defined monad must[1] be an instance of the class `Monad`, providing the implementation for those operators. The two main functions that must be defined over monads are:

- The function `return`, which receives a value of type `a` and returns computation that, once executed, returns that value. In the Listing A.5, `return` is defined with the constructor `Val`, that given a value it returns `Exc a`. This operator does not perform any other action.

- The infix operator `>>=`, also known as the binding operator. The signature of the operator is `M a -> (a -> M b) -> M b`. This operator allows to specify how two computations are composed in a monad. In the Listing A.5, the operator receives computation `Exc a` and a function `a -> Exc b`. On the one hand, if the computation is an `Error` the error should be propagated, so it returns the `Error`. On the other hand, if the computation does not return an error, it should apply the function, consequently returning the new `Exc b` computation.

Haskell provides the *do notation* in order to simplify programming with monads. In short, the do notation allows to write monadic computations using a pseudo-imperative style with named variables. The Listing A.6 shows the implementation of an interpreter for a tiny language to make integer divisions. In line 6 the keyword `do` specifies that we want to perform a sequence of monadic computations. The

[1]Actually, defining a monad does not require the definition of an instance of the `Monad` class, but it is convenient to have it, in order take advantage of various Haskell features such as the *do notation* that we will introduce shortly.

sequence is conformed by three computations: two `evalExc` (lines 6 and 7) and the `if` expression (lines 8-10). The operator `<-` used in lines 6 and 7 automatically binds the result of the evaluation of the terms `t` and `u`, to the variables `x` and `y` respectively. Then the `if` expression at lines 8-10 checks if the divisor is 0, in which case it raises an exception (using the `raiseE` function defined in Listing A.5), otherwise it performs the division and returns the value.

```
 1  data Term = Con Int | Div Term Term
 2
 3  evalExc  :: Term -> Exc Int
 4  evalExc (Con x)   = return x
 5  evalExc (Div t u) =
 6            do  x <- evalExc t
 7                y <- evalExc u
 8                if y == 0
 9                  then raiseE "division by zero"
10                  else return (div x y)
```

Listing A.6: Example of the Exception Monad <Haskell Code>

The *do notation* is nothing else than a shortcut to the use of the `>>=` operator to bind the computations. The do expression at Listing A.6 can be specified using the bind operator as follows:

```
evalExc t >>=
    \x -> evalExc u >>=
              \y -> if y == 0
                      then raiseE "division by zero"
                      else return (div x y)
```

Note that using the do notation greatly simplifies programming with monads. Also note that the binding operator of the exception monad encapsulates all the exception propagation logic. Consequently, we obtain a cleaner code in the interpreter implementation.

Now that we have introduced the basic concepts behind monads, we will present a more complex monad, the *State and Exception monad*, which models the computations that maintain state and that may throw exceptions. This monad is crucial for the implementation of the machines. The Listing A.7 shows the implementation of the monad. The polymorphic type defined for this monad is `StE s a`, where `s` is a type variable representing the type of the state maintained and `a` is the type of values returned by the computations. `StE s a` has a unique constructor containing a function that given the initial state it performs the monad computation and may return an exception or a pair with the resultant value and state.

The `return` function preserves the state and inserts the new returned value. The

```haskell
data StE s a = MkStE (s -> Exc (a, s))

funOfStE:: StE s a -> (s -> Exc (a, s))
funOfStE (MkStE f) = f

instance Monad (StE s) where
   return x = MkStE (\s -> return (x, s))
   (MkStE f) >>= g =
           MkStE (\s -> (f s) >>=
                       (\(v, s') -> (funOfStE (g v)) s'))

raiseStE :: String -> StE s a
raiseStE str = MkStE (\_ -> raiseE str)

getStE :: StE s s
getStE = MkStE (\s -> return(s, s))

putStE :: s -> StE s ()
putStE st = MkStE (\_ -> return((), st))
```

Listing A.7: State and Exception Monad <Haskell Code>

`>>=` operator builds a function that sequentially apply the functions `f:: s -> Exc (a, s)` and `g:: a-> StE s a`. So, `>>=` applies `f` over the initial state (`s`) returning a new pair `(v, s')`; the value is passed to `g` and evaluated in the state `s'`. Thus composing both computations.

In addition to the monad definition, the Listing A.7 shows the following functions: `raiseStE` that throws an exception in the `StE` monad, `getStE` that returns the state maintained by the monad and `putStE` that sets the state maintained with the monad. Note that the user of the monad `StE` does not have to deal with state maintenance and exception propagation logic, the binding operator handles it. When the user requires accessing or updating the state, it simply uses the `getStE` and `putStE` functions.

Appendix B

Concrete Syntax

This appendix presents the concrete syntax of the three languages described in this thesis, namely BASE, Kernel and PA languages.

This appendix is organized as follows. Section B.1 presents the concrete syntax for the BASE language. Section B.2 and B.3 present two extensions to the BASE syntax in order to define the Kernel and PA langauge syntaxes, respectively.

B.1 BASE Concrete Syntax

The description of the concrete syntax of the BASE language is specified using Backus-Naur Form (BNF). The terminals, i.e. keywords and symbols, are underlined in order to improve clarity. For any symbol α, we write: (α)? to say that α is optional, (α)* to denote 0 or more repetitions of α, and (α)+ to denote 1 or more repetitions of α.

```
<program>       ⟶   (<class>)*
<class>         ⟶   class <ident> (extends <ident>)?
                    { (<member>)* }
<member>        ⟶   <field> | <method>
<field>         ⟶   <type> <ident> ;
<method>        ⟶   <type> <ident> ( <parameters> ) { <exprBlock> }
<parameters>    ⟶   (<parameter> (,<parameter>)?)*
<parameter>     ⟶   <type> <ident>
<exprBlock>     ⟶   (<expr> ;)*
```

```
<expr>            ⟶  <literal> | <primitive> | <if> | <let> |
                     <varGet> | <varSet> | <fieldGet> |
                     <fieldSet> | <call> | <superCall> |
                     <newObject> | <instanceOf> | <cast> |
                     <newArray> | <newArrayInit> | <getArrayByIdx> |
                     <setArrayByIdx> | ( <expr> )
<literal>         ⟶  " <characters> " | <number> | True | False |
                     NULL
<primitive>       ⟶  <expr> <opArith> <expr> |
                     <expr> <opBoolBin> <expr> | <opLgcUni> <expr> |
                     write <expr> | newLine | $ <expr> |
                     <expr> ++ <expr>
<opArith>         ⟶  + | - | * | /
<opBoolBin>       ⟶  == | < | > | <opLgcBin>
<opLgcBin>        ⟶  || | &&
<opLgcUni>        ⟶  !
<if>              ⟶  if ( <expr> ) { <exprBlock> }
                     (else { <exprBlock> })?
<let>             ⟶  let ( <variables> ) in { <exprBlock> }
<variables>       ⟶  (<type> <ident> := <expr> ;)+
<varGet>          ⟶  # <ident>
<varSet>          ⟶  # <ident> := <expr>
<fieldGet>        ⟶  <ident>
<fieldSet>        ⟶  <ident> := <expr>
<call>            ⟶  <ident> ( <arguments> ) |
                     <expr>.<ident> ( <arguments> )
<arguments>       ⟶  (<expr> (,<expr>)?)*
<superCall>       ⟶  super.<ident> ( <arguments> )
<newObject>       ⟶  new <ident>
<instanceOf>      ⟶  <expr> instanceOf <type>
<cast>            ⟶  ( <type> ) <expr>
<newArray>        ⟶  new <ident> ([])+ [ <expr> ]
<newArrayInit>    ⟶  new <ident> ([])+ ( <arguments> )
<getArrayByIdx>   ⟶  <expr> [ <expr> ]
<setArrayByIdx>   ⟶  <expr> [ <expr> ] := <expr>
<type>            ⟶  void | int | bool | string | <ident> |
                     <ident> ([])+
<ident>           ⟶  <letter> (<letter> | <digit>)*
```

B.2 Kernel Language

The concrete syntax of the Kernel language is described by extending BASE language concrete syntax. The ⟶ arrow represents a new production. The ↪ arrow represents the extension of a BASE language production, by adding new cases. The ⟼ arrow represents a production that is overwritten.

```
<program>         ⟼   (<class> | <selector> | <hookset> |
                      <calldescriptor> | <link>)*
<selector>        ⟶   selector <ident> ( <parameters> ) {
                      <selectorbody> }
<selectorbody>    ⟶   on <operType> <ident> ,
                      when { <exprBlock> }
<hookset>         ⟶   <primitive> | <composite>
<primitive>       ⟶   hookset <kIdent> using { <selectorApply> } |
                      hookset <kIdent> { <selectorBody> }
<composite>       ⟶   hookset <kIdent> combine { <combineExpr> }
<combineExpr>     ⟶   <combineExpr> <opLgcBin> <combineExpr> |
                      <opLgcUni> <combineExpr> | <kIdent> |
                      ( <combineExpr> )
<calldescriptor>  ⟶   call <kIdent> {
                      from <operType> <ident> ,
                      to <classType> <ident> ,
                      do { <exprBlock> } }
<link>            ⟶   link <kIdent> {
                      (<callSpec> ,)+
                      to { <exprBlock> } }
<callSpec>        ⟶   from <kIdent> on <control> with <kIdent>
<control>         ⟶   BEFORE | AROUND | AFTER
<expr>            ↪   <selectorApply> | <selectorEval> | <proceed> |
                      <linkMO>
<selectorApply>   ⟶   <kIdent> ( <arguments> )
<selectorEval>    ⟶   <expr> : ( <expr> )
<proceed>         ⟶   proceed | proceed { <exprBlock> }
<linkMO>          ⟶   <kIdent> #MO
<kIdent>          ⟶   % <ident>
<operType>        ⟶   <ident>
```

B.3 PA Language

The concrete syntax of the PA language is described by extending the BASE language, as explained in the previous section.

```
<program>         ⟼   (<class> | <pointcut> | <advice>)*
<pointcut>        ⟶   pointcut <ident> <parameters> : <pcd> ;
<pcd>             ⟶   call ( <methodPattern> ) |
                      execute ( <methodPattern> ) |
                      aexecute () |
                      new (<ident>) |
                      get (<ident>) |
                      set (<ident>) |
                      within (<type>) |
                      withincode (<ident>) |
                      this (<ident>) |
                      target (<ident>) |
                      args (<idents>) |
                      <ident> (<idents>) |
                      cflow (<pcd>) |
                      cflowbelow (<pcd>) |
                      <pcd> <opLgcBin> <pcd> |
                      <opLgcUni> <pcd>
<methodPattern>   ⟶   <type> <ident> (<types>)
<idents>          ⟶   (<ident> (,<ident>)?)*
<types>           ⟶   (<type> (,<type>)?)*
<advice>          ⟶   before (<parameters>) : <pcd> { <exprBlock> } |
                      <type> around (<parameters>) : <pcd>
                      { <exprBlock> } |
                      after (<parameters>) : <pcd> { <exprBlock> }
<expr>            ↪   proceed (<arguments>)
```

Appendix C

Test Environment

This appendix describes the testing environment provided by the Reflex Sandbox. Through this environment, case-based testing can be performed in order to check that the machines behaves as expected. The tests can be performed on a per-machine or an inter-machine basis. The latter is oriented to test on a relationship that exists between two machines, e.g. testing the compilation between the PA machine to the Kernel machine.

This appendix is organized as follows. Section C.1 introduces an abstract representation of the machines on which the test environment bases its definition. Sections C.2 and C.3 describe the testing environment to perform single machine and inter-machine testing, respectively.

C.1 Machine Abstraction

In the test environment, tests are defined over an abstract representation of a machine, which corresponds to the data type `Machine`, shown in Listing C.1. `Machine` is a polymorphic data type defined over three type variables: `p` is the type it uses represent a program, `s` is the type it uses to represent the state and `v` is the type it uses to represent the values. A `Machine` consists of:

- **a name** used by the tests environment to render the output of a test suite execution.
- **an initial state** in which each test shall start its execution.

- **a parser** used by the test environment to parse the programs to be executed in the machine. It is represented by the function type `String -> Exc p`.
- **an interpreter** that given a program and an initial state, evaluates the program an returns it result.

```
data Machine p s v = CMachine Name s (String -> Exc p)
                              (p -> s -> Exc ((v, Output), s))

class (Show p, Show s, Show v) => IMachine p s v where
  getMachineName :: (Machine p s v) -> Name
  initialState :: (Machine p s v) -> s
  parse :: (Machine p s v) -> String -> Exc p
  execute :: (Machine p s v) -> String -> s -> Exc ((v, Output), s)
  executeAST :: (Machine p s v) -> p -> s -> Exc ((v, Output), s)
```

Listing C.1: Machine Abstraction <Haskell Code>

In addition, a Haskell type class (`IMachine`) is defined enclosing the most relevant functions to manipulate the abstract representation of the machine. Those functions provide direct access to the elements that compose the machine, except for the `execute` function that, given the program text, first parses the program, and then executes it.

C.2 Per-Machine Testing

The test environment provides the data type `TestSuite`, shown in Listing C.2, to represent a collection of tests that must be carried on over a single machine. `TestSuite` is also a polymorphic type defined over those type variables of the abstract machine being tested. A `TestSuite` consists of: a machine over which the test shall be performed and a list of *test cases*. A `TestCase` essentially specifies a program, given through the name of the file that contains it, and a test condition. The test environment, based on a `TestCase`, parses the program, executes it and tests the validity of the condition in order to determine whether the test succeeds or fails. There are three different kinds of tests. The `OutputTC` test case checks that the program ends its execution normally (i.e. without exceptions), and also, that the output and returned value are as expected. That is to say, the output of the execution is equal to its component element `Output` and that the function `v -> Bool` returns `True` when it is evaluated with the returned value. The `ExceptionTC` test case checks that the program ends its execution with an exception and that the `String` of the exception returns `True` once tested with the function `String -> Bool`. The `OutStateTC` test case is similar to `OutputTC`, but it also imposes a condition over the state of the machine, through the function `s -> Bool`.

```
data TestSuite p s v = CSuite (Machine p s v) [TestCase p s v]

data TestCase p s v =
       OutputTC (Machine p s v) FileName Output (v -> Bool)  |
       ExceptionTC (Machine p s v) FileName (String -> Bool) |
       OutStateTC (Machine p s v) FileName Output (v -> Bool) (s->Bool)
```

Listing C.2: Per-Machine TestSuite <Haskell Code>

The `ITestSuite` type class defines the functions in order to build a test suite and to run it. The function `makeTestSuite` receives a machine and returns a test suite for it, with an empty set of test cases. The three `add*Test` functions allow to add new test cases to a given `TestSuite`. Finally, the `runTestSuite` function, given the `TestSuite`, executes each of the test cases for that machine, outputting the results of the tests in the standard output.

```
 class (IMachine p s v) => ITestSuite p s v where
   makeTestSuite :: (Machine p s v) -> (TestSuite p s v)
   addOutputTest :: (TestSuite p s v) -> FileName -> Output ->
                    (v -> Bool) -> (TestSuite p s v)
   addExceptionTest :: (TestSuite p s v) -> FileName ->
                       (String -> Bool) -> (TestSuite p s v)
   addOutStateTest :: (TestSuite p s v) -> FileName -> Output ->
                      (v -> Bool) -> (s -> Bool) -> (TestSuite p s v)
   runTestSuite :: (TestSuite p s v) -> IO ()
```

Listing C.3: Per-Machine TestSuite - Type Class <Haskell Code>

Note that the `ITestSuite` type class provides the implementation of the five functions shown in the listing, however we do not show them here.

C.2.1 Using the Test Environment

Listing C.4 illustrates the construction of a test suite for the Kernel machine. In order to be able to test a machine, an abstract representation of it must be built. This requires defining the appropriate instance of the type class `IMachine` and building the `Machine` data value that represents the machine. For instance, the first line declares an instance of `IMachine` for the Kernel machine. Note that since the type class `IMachine` provides a default implementation for all its functions (not shown here), the instance does not require to implement any. The function `getKernelMachine` builds the data value for the Kernel machine, using its initial state, parser and interpreter.

```
instance IMachine Program State Value

getKernelMachine :: Machine Program State Value
getKernelMachine = CMachine "Kernel Machine" initialState
                            parseProgram runInState

instance ITestSuite Program State Value

testMachine :: IO ()
testMachine = runTestSuite (addTests initial)
              where
                initial = makeTestSuite getKernelMachine

addTests::TestSuite Program State Value -> TestSuite Program State Value
addTests ts = foldl add ts tests
   where
     add suite (fn, o, f) = addOutputTest suite fn o f
     tests = [((rootDirectory ++ "Fib.rsb"),
              "Before executing Fib with 3\n" ++
              "Before executing Fib with 2\n" ++
              "Before executing Fib with 1\n" ++
              "Before executing Fib with 0\n" ++
              "Before executing Fib with 1\nfib(3) is 2", isVoid),..]
```

Listing C.4: Per-Machine TestSuite - Example <Haskell Code>

In addition, the instance for the `ITestSuite` must also be defined for the Kernel machine. Again, no function must be implemented by the instance (`ITestSuite` provides a default implementation for all its functions, not shown here). The function `testMachine` builds the test suite, using `makeTestSuite` and `getKernelMachine`. Also, it adds the test cases using `addTests` and runs the tests. `addTests` adds a list of `OutputTC` to the test suite. The test case shown in the listing corresponds to a program that calculates the *fibonacci* numbers and defines a link to log all the calls to the method `fib`. The test case is specified by giving the name of the file containing the program, along with the expected output and return value. In conclusion, once `testMachine` gets executed, all the tests would be performed and the output would be printed in the standard output.

C.3 Inter-Machine Testing

The inter-machine testing facility offers the possibility of testing a relationship that exists between two machines. As shown in Figure C.1, such a relationship is specified through two functions: compilation and comparison. The compilation functions transforms a program in one machine, the so-called *origin machine*, into a program in the other, the so-called *destination machine*. Once both programs gets executed in their respective machine, the comparison functions checks that the relationship holds between the results of the execution.

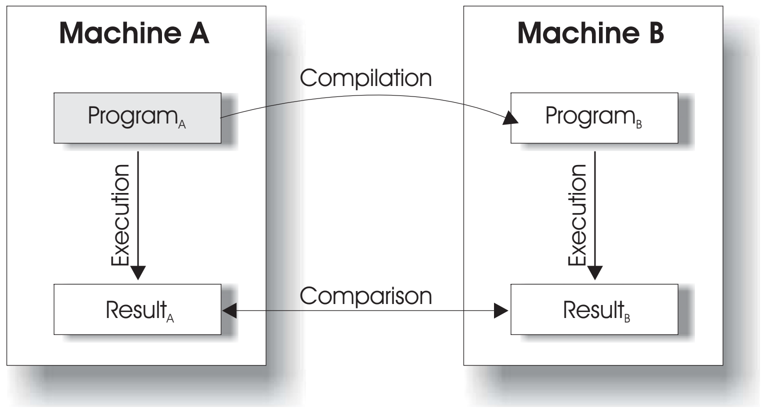


Figure C.1: Inter-Machine Testing

This testing facility is particulary useful to test the compilers that transforms aspect-oriented programs into Kernel programs. For instance, the compiler that transforms PA programs to Kernel programs, presented in Chapter 5, has been tested using this facility. In addition, it can be also used, for instance to test the relationship between two implementations of the Kernel machine, which may define two alternative semantics or even have different features. By performing inter-machine testings between the two Kernel machines, we can test that they behaves as expected, that is to say, in some test cases they have the same semantic effects while in other they differ.

The next section presents how this relationship is abstractly modeled in Haskell. Then, Section C.3.2 presents the definition of a test suite for inter-machine testing, which allows to define test cases to check the relationship. Finally, Section C.3.3 illustrates the use of this testing facility.

C.3.1 Relationship Abstraction

The relationship is represented using the data type `MachineRel` and the type class `IMachineRel`, both shown in Listing C.5. `MachineRel` is a polymorphic type defined over six type variables, representing the types of the programs, state and value of both machines. The data type encloses the name of the relation and the abstract representations for both machines. The first three functions of `IMachineRel` provide access to those elements of `MachineRel`. `compile` is the compilation function, shown in Figure C.1, which transforms a program of the origin machine (`po`) into a program in the target machine (`pd`). Also, it defines the `compareMachines` function, that given the results of the execution of the two machines compares them and determines whether the test succeeds or fails. If it fails, the function `errorRelMessage` is called in order to generate the error message.

```
data MachineRel po so vo pd sd vd =
       CMRel Name (Machine po so vo) (Machine pd sd vd)

class (IMachine po so vo, IMachine pd sd vd) =>
       IMachineRel po so vo pd sd vd where
  getMachineRelName :: (MachineRel po so vo pd sd vd) -> Name
  originMachine :: (MachineRel po so vo pd sd vd) ->
                   (Machine po so vo)
  targetMachine :: (MachineRel po so vo pd sd vd) ->
                   (Machine pd sd vd)
  compile :: (MachineRel po so vo pd sd vd) -> po -> Exc pd
  compareMachines :: (MachineRel po so vo pd sd vd) ->
                     Exc ((vo, Output), so) ->
                     Exc ((vd, Output), sd) -> Bool
  errorRelMessage :: (MachineRel po so vo pd sd vd) ->
                     (po, Exc ((vo, Output), so)) ->
                     (pd, Exc ((vd, Output), sd)) -> String
```

Listing C.5: Relationship Abstraction <Haskell Code>

C.3.2 Test Suite

The test suites are represents through the data type `TestSuite`[1] shown in Listing C.6, which consists of a relationship (`MachineRel`) on which the tests shall be performed and a list of test cases. A test case is specified only by giving the program (through a `FileName`) of the origin machine that must be tested, conversely to a per-machine test case that also encloses a condition. Note that the success condition is already specified in the function `compareMachines` presented in the previous section. The `ITestSuite`[2] type class provides the functions: `makeTestSuite` to build a test suite base on a relationship, `addTestFile` to add new test cases to the suite and `runTestSuite` to actually run the test suite. Once `runTestSuite` is executed, it performs the following tasks. In first place, it loads the program and parses it using the `parse` function

[1]Note that the data type's module-level name is equal to the `TestSuite` data type presented in Section C.2. However, the full-qualified names (not shown here) of both data types, and the data types themselves, are different.

[2]This type class is also different from the one presented in Section C.2 (`ITestSuite`), as well as its full-qualified name (not shown here).

of the origin machine (see Section C.1). The result of the parsing is a program of type `po`. In second place, using the `compile` function it transforms the `po` into a `pd` program of the destination machine. In third place, both programs get executed in their respective machines. Finally, the results of the programs are compared using the `compareMachines` to determine if the test was successful.

```
data TestSuite po so vo pd sd vd =
         CSuite (MachineRel po so vo pd sd vd) [FileName]

class (IMachineRel po so vo pd sd vd) =>
      ITestSuite po so vo pd sd vd where
  makeTestSuite :: (MachineRel po so vo pd sd vd) ->
                   (TestSuite po so vo pd sd vd)
  addTestFile :: (TestSuite po so vo pd sd vd) -> FileName ->
                 (TestSuite po so vo pd sd vd)
  runTestSuite :: (TestSuite po so vo pd sd vd) -> IO()
```

Listing C.6: Inter-Machine Test Suite <Haskell Code>

C.3.3 Using the Test Environment

Listings C.7 and C.8 show how the compiler form the PA machine and the Kernel machine is tested using inter-machine testing. In first place, Listing C.7 shows the definition of the relationship between both machines. It defines a Haskell `instance` of the type class `IMachineRel` providing the implementations for the `compile`, `compare-Machines` and `errorRelMessage` functions. The `compile` function calls the compiler presented in Section 5.3, here characterized by the function `compileStEState`, passing the PA machine program. The `applyStE` function executes the compilation process by passing the `initialState` to the monad `StE` (see Section 3.2.2.2 for an explanation of `applyStE`), resulting in the correspondent Kernel program. The `compareMachines` compares the results by checking that if both executions end with exceptions, they must have the same error message, and if both executions end normally, the output and the returned values must correspond. The function `compareResults` (not shown here) performs those comparisons. Finally, if the results do not correspond, the function `errorRelMessage` generates the appropriate string that is used by the test environment to report the error.

Listing C.8 shows the construction of a test suite that tests two programs, one that makes use of a *before* advice and another that makes use of an *around* advice calling `proceed`. In order to use the testing environment, an instance of `ITestSuite` is defined for the relationship. The function `testCompiler` creates the test suite, using `makeTestSuite` and `getMachineRel`. `getMachineRel` generates the `MachineRel` data

```
instance IMachineRel PA.Program PA.State PA.Value
                     Ker.Program Ker.State Ker.Value where
  compile ms pr =
   elimE compilate okBehavior errorBehavior
   where
     compilate = applyStE (compileStEState pr) Compiler.initialState
     okBehavior (pr', _) = return pr'
     errorBehavior msg = raiseE msg

  compareMachines ms outO outD
   | (isErrorE outO) && (isErrorE outD) =
        (getErrorMsgE outO) == (getErrorMsgE outD)
   | (isOkE outO) && (isOkE outD) =
        compareResults (fst (getValueE outO)) (fst (getValueE outD))
   | otherwise = False

  errorRelMessage ms (pO, outO) (pD, outD) =
    (show pD) ++ "\n\n" ++ "PA Machine\n" ++ (showResult outO) ++
                     "\n\nKernel Machine\n" ++ (showResult outD)
    where
      showResult res
         | isErrorE res =
           "Exception result: " ++ (getErrorMsgE res)
         | otherwise =
           "Normal result: \n\n" ++ "Value: \n\n" ++
           (show (fst (fst (getValueE res)))) ++
           "\n\nOutput: \n\n" ++ (snd (fst (getValueE res)))
```

Listing C.7: PA Machine to Kernel Machine - Example <Haskell Code>

value containing the relationship name and the two abstract machines involved in it. `testCompiler` also adds the two test cases and runs them.

```
instance ITestSuite PA.Program PA.State PA.Value
                    Ker.Program Ker.State Ker.Value where

testCompiler :: IO ()
testCompiler = runTestSuite suite
               where
                 suite = foldl addTestFile initial testList
                 initial = makeTestSuite getMachineRel
                 testList = [testBefore, testProceed]
                 testBefore = rootDirectory ++ "Before.rsb"
                 testProceed = rootDirectory ++ "AroundProceed.rsb"

getMachineRel :: MachineRel PA.Program ... Ker.Program ...
getMachineRel = CMSupport "PA to Kernel Compilation"
                          getPAMachine getKernelMachine
```

Listing C.8: PA Machine to Kernel Machine - Example (continued)

Bibliography

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In Peri Tarr, editor, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 87–98, Chicago, USA, March 2005. ACM Press.

[ACK05] Rubén Altman, Alan Cyment, and Nicolás Kicillof. On the need for setpoints. In Kris Gybels, Maja D'Hondt, Istvan Nagy, and Remi Douence, editors, *2nd European Interactive Workshop on Aspects in Software (EIWAS'05)*, September 2005.

[asb] Aspect sandbox website. http://www.cs.ubc.ca/labs/spl/projects/asb.html.

[aspa] Aspectj programming guide. http://www.eclipse.org/aspectj/doc/released/progguide/index.html.

[aspb] Aspectj website. http://eclipse.org/aspectj.

[AWB+93] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1993.

[BA01] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[BAT01] Lodewijk Bergmans, Mehmet Aksit, and Bedir Tekinerdogan. Aspect composition using composition filters. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 357–384. Kluwer Academic Publishers, 2001.

[BGW93] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. Clos in context: the shape of the design space. pages 29–61, 1993.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns (Volume 1)*. Wiley, 1996.

[Bru02] Kim B. Bruce. *Fundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In Vlissides and Schmidt [VS04], pages 331–344.

[CME02] The Concern Manipulation Environment website, 2002. http://www.research.ibm.com/cme.

[CMT04] Denis Caromel, Luis Mateu, and Éric Tanter. Sequential object monitors. In Martin Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086, pages 316–340, Oslo, Norway, June 2004.

[CN03] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In Frank Pfenning and Yannis Smaragdakis, editors, *Proceedings of the 2nd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2003)*, volume 2830, pages 364–376, Erfurt, Germany, September 2003.

[CN04] Shigeru Chiba and Kiyoshi Nakagawa. Josh: An open AspectJ-like language. In Lieberherr [Lie04], pages 102–111.

[DD99] Kris De Volder and Theo D'Hondt. Aspect-oriented logic meta-programming. In Pierre Cointe, editor, *Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection (Reflection 99)*, volume 1616, pages 250–272, Saint-Malo, France, July 1999.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487, pages 173–188, Pittsburgh, PA, USA, October 2002.

[DGH+04] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of aspectj programs. In Vlissides and Schmidt [VS04], pages 150–169.

[Dij68] Edsger W. Dijkstra. The structure of THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[DMS01] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 170–186, London, UK, 2001. Springer-Verlag.

[dRS84] Jim des Rivières and Brian C. Smith. The implementation of procedurally reflective languages. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 331–347, August 1984.

[DS01] Rémi Douence and Mario Südholt. A generic reification technique for object-oriented reflective languages. In *Higher-Order and Symbolic Computation*, volume 14, pages 7–34. Kluwer Academic Publishers, 2001.

[DS02] Rémi Douence and Mario Südholt. A model and a tool for event-based aspect-oriented programming (eaop). Research Report 02/11/INFO, École des Mines de Nantes, 2002.

[DT04] Rémi Douence and Luc Teboul. A pointcut language for control-flow. In Gabor Karsai and Eelco Visser, editors, *Proceedings of the 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286, pages 95–114, Vancouver, Canada, October 2004.

[FF00] 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.

[FW84] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 348–355, August 1984.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley professional computing series, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition, the Java Series.* Sun Microsystems, Inc, 2000.

[HF92] Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *SIGPLAN Notices*, 27(5):T1–T53, 1992.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Lieberherr [Lie04], pages 26–35.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, 1996.

[HO93] William H. Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA*, pages 411–428, 1993.

[HOT02a] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Research, 2002.

[HOT$^+$02b] William H. Harrison, Harold L. Ossher, Peri L. Tarr, Vincent Kruskal, and Frank Tip. CAT: A toolkit for assembling concerns. Technical Report RC22686, IBM Research, 2002.

[Ibr90] Mamdouh H. Ibrahim. Report of the workshop on reflection and metalevel architectures in object-oriented programming. In Norman Meyrowitz, editor, *Proceedings of the 5th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA/ECOOP 90)*, Ottawa, Canada, October 1990. ACM Press. ACM SIGPLAN Notices, 25(10).

[ILG$^+$97] John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. In *Int'l Scientific Computing in Object-Oriented Parallel Environments (ISCOPE)*, volume 1343 of *LNCS*. Springer-Verlag, 1997.

[JCJ$^+$92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, , and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[KAR$^+$93] Gregor Kiczales, J. Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. Metaobject protocols: Why we want them and what else they can do. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 101–118. MIT Press, 1993.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072, pages 327–353, Budapest, Hungary, June 2001.

[Kic96] Gregor Kiczales, editor. *Reflection'96*, San Francisco, CA, USA, April 1996.

[Kic01] Gregor Kiczales. The future of reflection. Invited talk at the Third International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001), September 2001.

[KLLH03] Sergei Kojarski, Karl Lieberherr, David H. Lorenz, and Robert Hirschfeld. Aspectual reflection. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Boston, Massachusetts, March 2003. ACM Press.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[LBS00] Thomas Ledoux and Noury Bouraqadi-Saâdani. Adaptability in mobile agent systems using reflection. RM 2000, Workshop on Reflective Middleware, April 2000.

[Lie04] Karl Lieberherr, editor. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, March 2004. ACM Press.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL-97-010, Palo Alto Research Center, 1997.

[Lop02] Cristina Videira Lopes. Aspect-oriented programming: An historical perspective (what's in a name?). Technical Report UCI-ISR-02-5, Institute for Software Research, University of California, Irvine, 2002.

[LSLX94] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, 1994.

[Mae87a] Pattie Maes. *Computional reflection*. PhD thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium, 1987.

[Mae87b] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, pages 147–155. ACM Press, 1987.

[McA96] Jeff McAffer. Engineering the meta-level. In Kiczales [Kic96], pages 39–61.

[MJD96] Jacques Malenfant, Marco Jacques, and François-Nicolas Demers. A tutorial on behavioral reflection and its implementation. In Kiczales [Kic96], pages 1–20.

[MK03] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895, pages 105–121, November 2003.

[MKD02] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In *AOSD Workshop on Foundations of Aspect-Oriented Languages*, pages 17–26, 2002.

[MKD03a] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622, pages 46–60, 2003.

[MKD03b] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Modeling crosscutting in aspect-oriented mechanisms. In Luca Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 2–28. Springer, 2003.

[MKL97] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL-97-009, Palo Alto Research Center, 1997.

[MMY94] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. An object-oriented concurrent reflective language for dynamic resource management in highly parallel computing. In *IPSJ SIG Notes*, volume 94-PRG-18, 1994.

[NCT04] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed aop. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15, New York, NY, USA, 2004. ACM Press.

[OI94] Hideaki Okamura and Yutaka Ishikawa. Object location control using meta-level programming. *Lecture Notes in Computer Science*, 821:299–319, 1994.

[OT01a] Harold Ossher and Petri Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 821–822, Washington, DC, USA, 2001. IEEE Computer Society.

[OT01b] Harold Ossher and Petri Tarr. The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. *Communications of ACM*, 44(10):43–50, October 2001.

[Par72] David Parnas. On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Ras01] Awais Rashid. A hybrid approach to separation of concerns: The story of sades. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 231–249, London, UK, 2001. Springer-Verlag.

[RFX] Reflex website. http://reflex.dcc.uchile.cl/.

[Riv96] Fred Rivard. Smalltalk: a reflective language. In Kiczales [Kic96], pages 21–38.

[RTN04] Leonardo Rodríguez, Éric Tanter, and Jacques Noyé. Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, Arica, Chile, November 2004. IEEE Computer Society Press.

[SLS03] Macneil Shonle, Karl Lieberherr, and Ankit Shah. Xaspects: an extensible system for domain-specific aspect languages. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37, New York, NY, USA, 2003. ACM Press.

[Smi82] Brian C. Smith. Reflection and semantics in a procedural language. Technical Report 272, MIT Laboratory of Computer Science, 1982.

[Smi84] Brian C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1984.

[Ste94] Patrick Steyaert. *Open Design of Object-Oriented Languages – A Foundation for Specializable Reflective Language Frameworks.* PhD thesis, Vrije Universiteit Brussels, Belgium, 1994.

[Str93] Robert J. Stroud. Transparency and reflection in distributed systems. *ACM Operating System Review*, 22(2):99–103, April 1993.

[Sul01] Gregory Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10), October 2001.

[Tan04] Éric Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming.* PhD thesis, University of Nantes and University of Chile, November 2004.

[Tan05] Éric Tanter. Metalevel facilities for multi-language AOP. In *2nd European Interactive Workshop on Aspects in Software (EIWAS 2005)*, Brussels, Belgium, September 2005.

[Tho99] Simon Thompson. *The Haskell: The Craft of Functional Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[TN04a] Éric Tanter and Jacques Noyé. Motivation and requirements for a versatile AOP kernel. In *1st European Interactive Workshop on Aspects in Software (EIWAS 2004)*, Berlin, Germany, September 2004.

[TN04b] Éric Tanter and Jacques Noyé. Versatile kernels for aspect-oriented programming. Research Report RR-5275, INRIA, July 2004.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Ron Crocker and Guy L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, October 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[TVP02] Éric Tanter, Michaël Vernaillen, and José Piquer. Towards transparent adaptation of migration policies. In *8th ECOOP Workshop on Mobile Object Systems (EWMOS 2002)*, Málaga, Spain, June 2002.

[VS04] John M. Vlissides and Douglas C. Schmidt, editors. *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*. acm, October 2004.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A nonreflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–37, 1988.

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 306–315, New York, NY, USA, 1988. ACM Press.

[Yok92] Yasuhiko Yokote. The ApertOS reflective operating system: The concept and its implementation. In *Proceedings of the 7th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 92)*, pages 414–434, Vancouver, British Columbia, Canada, October 1992. ACM Press. ACM SIGPLAN Notices, 27(10).

[Zim96] Chris Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.