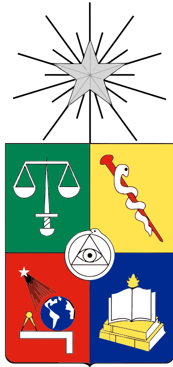


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

EFFECTIVE ASPECTS: A TYPED MONADIC MODEL TO CONTROL AND REASON ABOUT ASPECT INTERFERENCE

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS, MENCION COMPUTACIÓN
EN COTUTELA CON LA ÉCOLE DES MINES DE NANTES

ISMAEL JOSÉ FIGUEROA PALET

PROFESOR GUÍA:
ÉRIC TANTER

PROFESOR CO-GUÍA:
NICOLAS TABAREAU

MIEMBROS DE LA COMISIÓN:
JOHAN FABRY
MARIO SÜDHOLT
ERIK ERNST
BRUNO C. D. S. OLIVEIRA

Este trabajo ha sido parcialmente financiado por CONICYT y los Equipos Asociados INRIA RAPIDS y REAL

SANTIAGO DE CHILE
2014

Resumen

La Programación Orientada a Aspectos (AOP) apunta a mejorar la modularidad y reusabilidad en sistemas de software al ofrecer un mecanismo de abstracción para manejar *crosscutting concerns*. Sin embargo, en la mayoría de los lenguajes orientados a aspectos; los aspectos tienen poder casi sin restricciones, lo que eventualmente entra en conflicto con las metas anteriores. En este trabajo presentamos *EffectiveAspects*: un nuevo enfoque para incorporar el modelo AOP de pointcut/advice en un lenguaje funcional estáticamente tipeado como Haskell.

Como primera contribución, definimos una incorporación completa del modelo de pointcut/advice al lenguaje, usando mónadas. La coherencia de tipos se garantiza explotando el sistema de tipos subyacente, en particular *phantom types* y una nueva *type class* que implemente un algoritmo de anti-unificación. Los aspectos son de primera clase, pueden ser desplegados dinámicamente, y el lenguaje de pointcuts es extensible; por lo tanto combina la flexibilidad de lenguajes de aspectos dinámicamente tipeados con las garantías de un sistema de tipos estático. Las mónadas nos permiten razonar directamente sobre los efectos tanto en los aspectos como en los programas base mediante técnicas monádicas tradicionales. Con esto, extendemos la noción de Open Modules propuesta por Aldrich con efectos y con pointcuts protegidos, que son interfaces que restringen la aplicación externa de advice. Estas restricciones son enforzadas estáticamente usando el sistema de tipos. También, adaptamos las técnicas de EffectiveAdvice para razonar y enforzar propiedades del flujo de control; así como también adaptamos su enfoque basado en parametricidad para controlar la interferencia de efectos. Luego de mostrar que este último enfoque no es suficiente en presencia de múltiples aspectos, proponemos un nuevo enfoque basado en *monad views*, una nueva técnica para manejar mónadas, desarrollada por Schrijvers y Olivera.

Nuestra segunda contribución se basa en un poderoso modelo para razonar sobre la composición de componentes basados en mixins que incorporan efectos computacionales. Este modelo se basa en razonamiento ecuacional, parametricidad y leyes algebráicas de las mónadas. Nuestra contribución es mostrar cómo razonar sobre interferencia en la presencia de cuantificación sin restricción, a través de pointcuts. Mostramos que el razonamiento global puede ser composicional, lo que es clave para la escalabilidad de nuestro enfoque en el contexto de grandes sistemas que evolucionan. Demostramos un teorema general de equivalencia que se basa en algunas condiciones que pueden ser establecidas, reutilizadas y adaptadas por separado a medida que el sistema evoluciona. El teorema está definido en términos de un modelo abstracto de AOP monádico.

Este trabajo desarrolla técnicas de razonamiento de efectos, basadas en tipos, para el modelo pointcut/advice, en un modelo que es expresivo y extensible; y que permite el desarrollo de aplicaciones orientadas a aspecto robustas y la experimentación con nuevas semánticas de AOP.

Abstract

Aspect-oriented programming (AOP) aims to enhance modularity and reusability in software systems by offering an abstraction mechanism to deal with crosscutting concerns. However, in most general-purpose aspect languages aspects have almost unrestricted power, eventually conflicting with these goals. In this work we present EffectiveAspects: a novel approach to embed the pointcut/advice model of AOP in a statically-typed functional programming language like Haskell. Our work comprises two main contributions.

First, we define a monadic embedding of the full pointcut/advice model of AOP. Type soundness is guaranteed by exploiting the underlying type system, in particular phantom types and a new anti-unification type class. In this model aspects are first-class, can be deployed dynamically, and the pointcut language is extensible, therefore combining the flexibility of dynamically-typed aspect languages with the guarantees of a static type system. Monads enable us to directly reason about computational effects both in aspects and base programs using traditional monadic techniques. Using this we extend Aldrich's notion of Open Modules with effects, and also with protected pointcut interfaces to external advising. These restrictions are enforced statically using the type system. Also, we adapt the techniques of EffectiveAdvice to reason about and enforce control flow properties. Moreover, we show how to control effect interference using the parametricity-based approach of EffectiveAdvice. We show that this approach falls short in the presence of multiple aspects and propose a different approach using *monad views*, a novel technique for handling the monad stack, developed by Schrijvers and Oliveira. Then, we exploit the properties of our model to enable the modular construction of new semantics for aspect scoping and weaving.

Our second contribution builds upon a powerful model to reason about mixin-based composition of effectful components and their interference, based on equational reasoning, parametricity, and algebraic laws about monadic effects. Our contribution is to show how to reason about interference in the presence of unrestricted quantification through pointcuts. We show that global reasoning can be compositional, which is key for the scalability of the approach in the face of large and evolving systems. We prove a general equivalence theorem that is based on a few conditions that can be established, reused, and adapted separately as the system evolves. The theorem is defined for an abstract monadic AOP model; we illustrate its use with a simple version of the model just described.

This work brings type-based reasoning about effects for the first time in the pointcut/advice model, in a framework that is both expressive and extensible. The framework is well-suited for development of robust aspect-oriented systems as well as being a research tool for experimenting and reasoning about new aspect semantics.

A mi futura familia

To my future family

Agradecimientos

En primer lugar, agradezco a Dios por su apoyo durante todos estos años así como por todos los milagros que han conducido al término de esta tesis.

De vuelta en la Tierra, hay muchas personas a las cuales estoy agradecido. Estaré siempre en deuda con mis guías Éric Tanter y Nicolas Tabareau. Gracias por mostrarme el mundo de la investigación profesional, con todos sus desafíos, ventajas y desventajas. Gracias por enseñarme a pensar como un investigador y por obligarme a superar mi inercia y pereza. Aún más importante, gracias por dejarme fallar cuando era necesario y por apoyarme a pesar de mis muchos errores. Sería un privilegio poder seguir trabajando con ustedes.

También quiero agradecer a Tom Schrijvers por su apoyo y colaboración durante este trabajo. Luego de olvidar contestar uno de sus correos durante más de un año, él tuvo la amabilidad de invitarme a la Universidad de Ghent para comenzar en lo que ahora es la última parte de este trabajo. Gracias Tom por el privilegio, ojalá este sea el comienzo de una colaboración estable y fructífera.

Estoy muy agradecido de mi país, el cual a través de CONICYT me dio el financiamiento necesario para mis estudios. Similarmente, agradezco a INRIA por el financiamiento para muchas pasantías y viajes a Francia. Estoy agradecido de todas las personas del Departamento de Ciencia de la Computación (DCC) de la Universidad de Chile. Por favor perdonénme si alguien no está incluido en este listado no exhaustivo. Agradezco a mis compañeros y amigos: Milton Inostroza, Héctor Ferrada, Jorge Jara, Daniel Moreno, Teresa Bracamonte, Cristobal Navarro, entre muchos otros. Estoy también agradecido de los profesores y personal administrativo, en particular de Angélica Aguirre y Sandra Gaez, gracias a todos por su apoyo! Mirando al pasado, estoy muy agradecido de Rubén Carvajal-Schiaffino por inspirarme a continuar mis estudios más allá del pregrado. En el lado francés estoy también agradecido de mucha gente en la École des Mines de Nantes: Jacques Noyé, Mario Südholt, Rémi Douence, Ismael Mejías, Jurgen Van Ham, Diana Allam, Guilhem Jaber, Vincent Armant, Cécile Derouet, Pierre Salmon, Benoit Querniard; y de mis amigos chilenos en Francia, Gabriel Rodríguez, Alejandra Ramos e Ignacio Salas.

Al último, pero no por ello menos importante, agradezco a mis padres y a mi familia por apoyarme todos estos años—incluso cuando no entendían por qué decidí seguir estudiando. Finalmente, quiero agradecer especialmente a Eliana y Amanda por todo su amor, apoyo, comprensión y motivación. A ustedes dedico este trabajo.

Contents

List of Figures viii

1 Introduction 1

2 Preliminaries 6
- 2.1 Aspect-Oriented Programming . . . 6
- 2.2 Basics of Haskell Programming . . . 8
  - 2.2.1 Values and Types . . . 8
  - 2.2.2 Functions and Pattern Matching . . . 11
  - 2.2.3 Lexical Scoping and Local Identifiers . . . 12
  - 2.2.4 Type Classes and Ad Hoc Polymorphism . . . 12
  - 2.2.5 Bounded Polymorphism . . . 13
  - 2.2.6 `newtype` Declarations . . . 14
- 2.3 Monadic Programming in a Nutshell . . . 14
  - 2.3.1 Plain Monadic Programming . . . 15
  - 2.3.2 Polymorphism on the Monad Stack . . . 18

I Design and Type Safety 20

3 Introducing Aspects 21
- 3.1 Join Point Model . . . 22
- 3.2 Aspect Deployment . . . 24
- 3.3 Aspect Weaving . . . 25

4 Type Safety 28
- 4.1 Typing Aspects, Informally . . . 28
  - 4.1.1 Typing Pointcuts . . . 28
  - 4.1.2 Typing Aspects . . . 31
- 4.2 Typing Aspects, Formally . . . 32
  - 4.2.1 Type Substitutions . . . 32
  - 4.2.2 Statically Computing Least General Types . . . 33
  - 4.2.3 Pointcut Safety . . . 35
  - 4.2.4 Advice Safety . . . 36
  - 4.2.5 Safe Aspects . . . 37

5 Discussion About the Model 39

5.1 Quantification . . . 39
5.1.1 Approximating Equality on Functions . . . 39
5.1.2 Tagged Function Applications . . . 40
5.2 Aspects and Bounded Polymorphism . . . 41
5.3 Obliviousness . . . 42
5.4 Technical Requirements of our Model . . . 43

6 Related Work, Part I **44**

II Controlling Effects **47**

7 Open and Protected Modules, with Effects **48**
7.1 Background: Open Modules . . . 48
7.2 A Simple Example . . . 49
7.3 Protected Pointcuts . . . 49
7.4 Enforcing Control Flow Properties . . . 50

8 Controlling Effect Interference **53**
8.1 Distinguishing Aspect and Base Computation . . . 53
8.2 Interference Between Multiple Aspects . . . 56
8.3 Background: Monad Views . . . 58
8.4 Beyond the Aspect/Base Distinction . . . 60

9 Modular Language Extensions **63**
9.1 Control Flow Pointcut . . . 63
9.2 Secure Weaving . . . 65
9.3 Privileged Aspects . . . 65
9.4 Execution Levels . . . 66
9.5 Reasoning about Language Extensions . . . 68
9.6 Other Approaches to Modular AOP Language Extensions . . . 69

III Compositional Reasoning About Aspect Interference **71**

10 The Challenge of Compositional Reasoning **72**
10.1 Compositional Reasoning, Informally . . . 73
10.2 Background: Monadic Reasoning in a Nutshell . . . 75
10.2.1 Equational Reasoning and Observational Equivalence . . . 76
10.2.2 Monad Laws . . . 76

11 Compositional Reasoning, Formally **78**
11.1 Abstracting Monadic AOP . . . 78
11.1.1 Join Point Model . . . 78
11.1.2 Necessary Properties of $\mathbb{A}_{\mathrm{T}}$ . . . 79
11.1.3 Running Example in Monadic Style . . . 80
11.2 Compositional Harmlessness Theorem . . . 81
11.2.1 System Decomposition . . . 83

11.2.2 Compositional Weaving . . . 83
11.2.3 Compositional Projection . . . 84
11.2.4 Contextual Harmlessness . . . 86
11.2.5 Local Harmlessness . . . 87

12 A Simple Monadic AOP Model **88**
12.1 An Embedding of Open Applications . . . 88
12.2 Running $\mathbb{A}_\mathrm{T}$ Computations . . . 90
12.3 Aspect Weaving . . . 90
12.4 Properties of $\mathbb{A}_\mathrm{T}$ . . . 90

13 Local Harmlessness **93**
13.1 AOP-MRI Translation . . . 93
13.2 Background: the MRI Framework . . . 94
13.3 Connecting MRI to AOP . . . 95
13.4 Harmlessness of Logging . . . 97
13.5 Harmlessness of Memoization . . . 98

14 Related Work, Part III **100**
14.1 Approaches to Modular Reasoning in AOP . . . 100
14.1.1 Protecting Modules from Aspects . . . 100
14.1.2 Limiting the Scope of Aspects . . . 104
14.2 Reasoning about Interference in AOP . . . 107

IV Conclusions **111**

15 Contributions **112**

16 Perspectives **114**

Bibliography **117**

Appendix A Proofs of the Properties of the Simple Monadic AOP Model **124**
A.1 Monad Laws . . . 124
A.1.1 Left Identity . . . 124
A.1.2 Right Identity . . . 124
A.1.3 Associativity of $\gg\!=_{\mathbb{A}_\mathrm{T}}$ . . . 125
A.2 Monad Transformer Laws . . . 126
A.2.1 Identity Preservation . . . 126
A.2.2 Composition Preservation . . . 126
A.3 $run\mathbb{A}_\mathrm{T}$ is a Monad Morphism . . . 127
A.3.1 Identity preservation . . . 127
A.3.2 Compositionality . . . 127
A.3.3 $run\mathbb{A}_\mathrm{T}$ is left inverse of $lift$ . . . 127

List of Figures

4.1 The $LeastGen'$ type class. An instance holds if c is the least general type of a and b. 33

7.1 Fibonacci module. 49
7.2 Memoized Fibonacci module. 50
7.3 Replacement, augmentation and narrowing advice combinators (adapted from (Oliveira et al., 2010)). 51
7.4 Memoization as a narrowing advice (adapted from (Oliveira et al., 2010)). 52

8.1 Fibonacci with error. 55
8.2 Applying structural masks to the monad stack S_1. 61

9.1 Execution levels monad transformer and level-shifting operations 66
9.2 Redefining aspect deployment for execution levels semantics. An aspect is made level-aware by transforming its pointcut and advice. 67
9.3 A program that loops unless execution levels are used. 67

10.1 State and Writer monads transformers: constructors, evaluation and projection functions. 76

11.1 Logging and memoization advice in monadic style 81

12.1 $\mathbb{A}_\mathrm{T}$ instances for the $Monad$ and $MonadTrans$ type classes. 89
12.2 Proof of the second monad morphism law for $run\mathbb{A}_\mathrm{T}$. 91

13.1 Fibonacci function. Left: in the simple pointcut/advice model of Chapter 12. Right: in the MRI setting (taken from (Oliveira et al., 2012)) 97

14.1 Execution levels in action: pointcut and advice are evaluated at level 1, `proceed` goes back to level 0 (from (Tanter, 2010)) 106

Chapter 1

Introduction

"Formal methods will never have a significant impact until they can be used by people that don't understand them" Attributed to Tom Melham

"They [types] are the world's best lightweight formal method!" Pierce (2012)

Aspect-oriented programming languages support the modular definition of crosscutting concerns through a join point model (Kiczales et al., 1997). In the pointcut/advice mechanism, crosscutting is supported by means of pointcuts, which quantify over join points, in order to implicitly trigger advice (Wand et al., 2004). Such a mechanism is typically integrated in an existing programming language by modifying the language processor, may it be the compiler (either directly or through macros), or the virtual machine. In a typed language, introducing pointcuts and advices also means extending the type system, if type soundness is to be preserved. For instance, AspectML (Dantas et al., 2008) is based on a specific type system in order to safely apply advice. AspectJ (Kiczales et al., 2001) does not substantially extend the type system of Java and suffers from soundness issues. StrongAspectJ (De Fraine et al., 2008) addresses these issues with an extended type system. In both cases, proving type soundness is rather involved because a whole new type system has to be dealt with.

In functional programming, the traditional way to tackle language extensions, mostly for embedded languages, is to use monads (Moggi, 1991; Wadler, 1992). Early work on AOP suggests a strong connection to monads. De Meuter (1997) proposed to use them to lay down the foundations of AOP, and Wand et al. (2004) used monads in their denotational semantics of pointcuts and advice. Recently, Tabareau (2012) proposed a weaving algorithm that supports monads in the pointcut and advice model, which yields benefits in terms of extensibility of the aspect weaver, although in this work the weaver itself was not monadic but integrated internally in the system.

In general terms, this thesis aims to give a concrete answer to the following research question:

How can we use types to control and reason about interference of computational effects, such as mutable state or exceptions, in aspect-oriented languages?

To provide such a concrete answer, and considering the above, we chose to use monads as the mechanism to directly manipulate and reason about computational effects. This in turn led us to us-

ing the Haskell programming language, given its extraordinary support for monadic programming. In particular, the thesis presents two main contributions under the name of EffectiveAspects: the first is a **lightweight, full-fledged embedding of aspects** in Haskell, that is typed and monadic. The second is a **simplified monadic framework** which allows *compositional reasoning* about aspect interference in the presence of unrestricted quantification through pointcuts.

With respect to the full-fledged model of AOP, by *lightweight* we mean that aspects are provided as a small standard Haskell library. The embedding is *full-fledged* because it supports dynamic deployment of first-class aspects with an extensible pointcut language—as is usually found only in dynamically-typed aspect languages like AspectScheme (Dutchyn et al., 2006) and AspectScript (Toledo et al., 2010).

By *typed*, we mean that in the embedding, pointcuts, advices, and aspects are all statically typed, and pointcut/advice bindings are proven to be safe. Type soundness is directly derived by relying on the existing type system of Haskell (type classes (Wadler and Blott, 1989), phantom types (Leijen and Meijer, 1999), and some recent extensions of the Glasgow Haskell Compiler). Specifically, we define a novel type class for anti-unification (Plotkin, 1970; Reynolds, 1970), which is key to define safe aspects.

Because the embedding is *monadic*, we derive two notable advantages over ad-hoc approaches to introducing aspects in an existing language. First, we can directly reason about aspects and effects using traditional monadic techniques. In short, we can generalize the interference combinators of EffectiveAdvice (Oliveira et al., 2010) in the context of pointcuts and advice. And also we can use non-interference analysis techniques such as those from EffectiveAdvice, and from other advanced mechanisms, in particular *monad views* (Schrijvers and Oliveira, 2011). Second, because we embed a monadic weaver, we can modularly extend the aspect language semantics. We illustrate this with several extensions and show how type-based reasoning can be applied to language extensions.

The second main contribution of this thesis is a formal framework to reason about aspect interference in the presence of quantification. To do this we bridge the gap between our model and the MRI framework developed recently by Oliveira et al. (2012). MRI, which stands for *Modular Reasoning about Interference*, is a purely functional model of incremental programming with effects. MRI enables modular reasoning about non-interference of aspects using techniques like equational reasoning and parametricity. The main results from MRI are two theorems about harmlessness of mixins.

Because MRI is the successor of EffectiveAdvice, on which our full-fledged aspect model is based, we wanted to bring the reasoning power of MRI to the setting of AOP. The main difficulty was that MRI does not address quantification: advices are mixins which are applied explicitly. The lack of quantification greatly simplifies modular reasoning because it is enough to study a single function and mixin in isolation. In addition, MRI only focuses on step-wise applications of mixins, in which the composition of a base component with a mixin can then be treated as a new base component for a subsequent mixin application. In contrast, in the pointcut/advice model of AOP, several aspects live in an aspect environment and are all woven at each join point.

Therefore, and to be more specific, our second main contribution is that we demonstrate, in a simplified model of monadic AOP, that while unrestricted quantification hampers *modular* reasoning, it is amenable to *compositional* reasoning: global harmlessness results can be obtained

through the composition of smaller proofs. This compositionality makes it possible to evolve an aspect-oriented system and *reuse* previously-established results. In particular, we formulate a general behavioral equivalence theorem between a given aspect-oriented system run with respect to two different aspect environments, modulo projection of additional side-effects. This general theorem is proven assuming four sufficient conditions that have to be established separately. When an aspect-oriented system evolves, only some of these conditions may need to be re-established in order to preserve the general theorem.

Organization of the Thesis

The remainder of the thesis is organized as follows. We start with a chapter that presents some preliminary background, followed by three parts that address the main developments of this work: design and type safety, controlling effects, and compositional reasoning about aspect interference. Finally, we conclude in a fourth part by summarizing the contributions of our work and highlighting potential lines of future work. Note that we split the discussion of related work into two parts: the first is presented at the end of Part I, while the second is presented at the end of Part III.

Preliminaries

Chapter 2 presents preliminar background needed specially in the first part of the thesis. Further background is introduced gradually as it is required in order to present our work. In particular this chapter presents a brief introduction to aspect-oriented programming, a tutorial-style presentation of Haskell programming, and an overview of monadic programming in Haskell. Readers proficient with any of these topics may safely skip the corresponding section, or the whole chapter.

Part I: Design and Type Safety

The first part describes the design of the full-fledged model of monadic AOP in Haskell, as well as the proof that the model is statically safe. In particular:

Chapter 3 defines the particular join point model used in our approach. That is, it defines how join points, pointcuts, advices and aspects are implemented. It also describes the mechanisms for aspect deployment and aspect weaving.

Chapter 4 addresses the challenge of how to reuse the type system of Haskell in order to prove that the pointcut/advice are statically safe. That is, to guarantee that in a well-typed program aspects will never be applied incorrectly, with respect to the types of their arguments.

Chapter 5 discusses several design issues about the model, in particular with respect to quantification, obliviousness and bounded polymorphism.

Chapter 6 discusses work directly related to the model proposed in this part. In particular, it shows previous connections between monads and AOP and compares our model to other functional aspect languages.

Part II: Controlling Effects

The second part describes how to reuse the results from EffectiveAdvice and Open Modules in order to enforce restrictions on aspects through the use of *protected pointcuts*. It also shows how to implement modular language extensions through the use of monad transformers. In particular:

Chapter 7 introduces the notion of protected pointcuts and advice combinators, and how that generalizes the control flow combinators from EffectiveAdvice.

Chapter 8 describes two approaches to control effect interference. The first uses the parametricity-based approach of EffectiveAdvice. After highlighting the limitations of this approach, this chapters describes a novel approach using *monad views*, a technique recently developed by Schrijvers and Oliveira (2011).

Chapter 9 describes several modular extensions to the aspect semantics. In particular it shows how to implement a user-defined control flow pointcut, the semantics of *secure, priviledged and protected weaving*, as well as the semantics of execution levels (Tanter et al., 2014).

Part III: Compositional Reasoning About Aspect Interference

In the third part we step back from the full-fledged embedding of aspects in order to present a general theorem of compositional aspect harmlessness. This theorem is proven with respect to an abstract monadic model. In particular:

Chapter 10 presents the challenges of compositional reasoning by giving some examples in a fictitious ML-like language. It also introduces some additional background on monadic reasoning.

Chapter 11 first introduces the abstract monadic model of AOP, and then presents the formal development of the compositional harmlessness theorem. One of the preconditions of the theorem is *local harmlessness*, which is specific to each particular monadic AOP model.

Chapter 12 defines a simple monadic AOP model that fulfills the formal requirements of the compositional harmlessness theorem. The model is a simplification of the one developed in Chapter 3.

Chapter 13 illustrates how to prove local harmlessness in the simple monadic AOP model developed in the previous chapter. In particular it illustrates that local harmlessness can be proven by reusing the results of MRI.

Chapter 14 finishes the discussion of related work, started in Chapter 6, by summarizing several approaches to modular reasoning in AOP and for reasoning about aspect interference.

Part IV: Conclusions

In the final part of the thesis, Chapter 15 summarizes the contributions of this work while Chapter 16 describes potential directions for future work. Finally, Appendix A shows the proofs regarding some properties of the simple monadic model of Chapter 12.

Related Publications and Implementations

The results presented in Parts I and II were initially published by Tabareau et al. (2013) and later extended by Figueroa et al. (2014). The framework for compositional reasoning, *i.e.* the result of Part III, was recently published by Figueroa et al. (2014).

Earlier work on a monadic weaver implemented in Typed Racket was published by Figueroa et al. (2012). Regarding modular language extensions (Chapter 9) Figueroa et al. (2013) presented a modular implementation of the *programmable membranes* semantics of AOP (Tanter et al., 2012). Tangentially related to this thesis, the author also co-authored an extension to execution levels (Figueroa and Tanter, 2011), as well as the extended version of that work (Tanter et al., 2014).

The implementation of the full-fledged model is available at http://pleiad.cl/effectiveaspects. The model for compositional reasoning is available at http://pleiad.cl/research/cri.

Chapter 2

Preliminaries

This chapter presents the preliminary concepts used throughout the thesis, in particular those used in Part I. To not overload the reader we start with this minimal background, and then, as our presentation advances, we will progressively introduce the concepts required for specific subsections of the thesis.

We start by briefly describing aspect-oriented programming (Section 2.1); our summary is indeed very short because we assume that readers will be familiar with this topic. However, based on our experience in submitting our work to the community of aspect-oriented researchers, the same cannot be said about Haskell programming in general, and monadic programming in particular. This is why we provide a tutorial-like introduction to Haskell programming (Section 2.2), and specifically to the basics of monadic programming in Haskell (Section 2.3). Readers already familiar with any of these topics may safely skip the corresponding sections (or the entire chapter).

2.1 Aspect-Oriented Programming

Separation of concerns (Parnas, 1972) is a design principle for software systems where the complexity is addressed by separating the problem into individual and manageable concerns or components. In addition to the reduced complexity, the main benefit of this approach is allowing simpler and independent evolution of the software. The potential for separating concerns is greatly influenced by the design of the programming language in use; in particular, traditional paradigms such as procedural, object-oriented, or functional programming only allow developers to decompose a system under a single decomposition mechanism (procedures, objects, or functions), an issue that is informally known as *the tyranny of the dominant decomposition* (Tarr et al., 1999).

Aspect-oriented programming (AOP) is a programming paradigm proposed by Kiczales et al. (1996) as an advanced mechanism to modularize *crosscutting concerns*. These concerns are said to *crosscut* the code of a system because they cannot be properly modularized in the dominant decomposition mechanism of the language—therefore they are either *scattered* (appear in multiple modules), *tangled* (several concerns are implemented in the same method or function), or both. Typical examples of crosscutting concerns are persistence, monitoring, security and error handling,

among others. This general definition of AOP does not specify how to implement the means for modular crosscutting behavior; instead, the particular support for crosscutting in an aspect-oriented language lies in its *join point model* (Masuhara et al., 2003). Indeed, there are several mechanisms or models to implement AOP, including at least those described by Masuhara and Kiczales (2003): pointcut/advice, open classes, traversal specifications, and class composition.

In our work we focus only on the pointcut/advice model, which is arguably the most emblematic AOP mechanism to date, given its use in mainstream aspect languages like AspectJ (Kiczales et al., 2001) and several languages used in research, like *e.g.* AspectScript (Toledo et al., 2010), AspectScheme (Dutchyn et al., 2006), AspectML (Dantas et al., 2008) or Aspectual Caml (Masuhara et al., 2005). In the pointcut/advice model crosscutting is supported by means of *pointcuts*, which are predicates to quantify over specific (dynamic) points in program execution, called *join points*, in order to implicitly trigger the execution of *advice*. An aspect is essentially just a pointcut/advice pair. Although aspects provide a source-level modularization of the code, the crosscutting behavior declared in aspects must be inserted in the proper points of execution such that it is reflected at runtime. This is done by a *weaving process*, which can be static (*e.g.* the AspectJ compiler), dynamic (*e.g.* the AspectScheme interpreter) or a combination thereof (*e.g.* what is done in DynamicAspectJ (Assaf and Noyé, 2008)).

Illustration in AspectJ To briefly illustrate these concepts, consider the following implementation of a simple logging aspect in AspectJ:

```
aspect Logging {

  pointcut callGetConfiguration : call(Integer getConfiguration())

  Object around() : callGetConfiguration {
    logger.append("Calling getConfiguration");
    return proceed();
  }
}
```

Similar to Java classes, an aspect is declared using the aspect keyword. Pointcuts can be anonymous or named, as in our example. The callGetConfiguration pointcut matches all calls to the getConfiguration method, which returns an integer and takes no arguments. Finally, we define an around advice, which performs the action indicated in its body upon join points matched by callGetConfiguration. An around advice executes in place of a join point matched by its corresponding pointcut. The special method proceed, which is only available inside the body of around advice, can be used to resume the computation of the matched join point. In our example, the advice first writes to the logger object and then resumes the advised method by calling proceed.

Quantification and obliviousness Filman and Friedman (2000) proposed that *quantification* and *obliviousness* were the essential characteristics of an aspect-oriented language:

> "AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.'
>
> Filman and Friedman (2000)

However this characterization may be too strong, as wild quantification has been shown to hamper (modular) reasoning of aspect-oriented systems (Aldrich, 2005; Kiczales and Mezini, 2005; Sullivan et al., 2010). Indeed, some proposals in the AOP community have explicit, rather than oblivious, mechanisms or interfaces to identify join points of interest (Bodden, 2011; Hoffman and Eugster, 2007; Steimann et al., 2010; **?**). As we explain in Chapter 3 and discuss later in Section 5.3, our model of aspect-oriented programming uses explicit emission of join points.

2.2 Basics of Haskell Programming

We now present a brief tutorial introduction to the basics of Haskell programming, while the next section overviews monadic programming in Haskell. We believe this introduction should suffice for readers not already proficient in Haskell. Nevertheless, for a deeper introduction to the language we recommend the *Gentle Introduction to Haskell*[1] (on whose structure this section is based on), complemented with the *Try Haskell*[2] site, which allows one to run Haskell directly from a web browser. We also recommend the excellent *Learn You a Haskell* book and website[3]. Note this is not an introduction to functional programming *per se*, but rather to the specific syntax and semantics of Haskell.

According to its website[4], Haskell is a programming language that is: *polymorphically statically typed*, *lazy*, and *purely functional*. Its implementation is based on the polymorphic lambda calculus, and is typically used as a convenient source language for System F_ω (*e.g.* (Figueroa et al., 2014; Oliveira et al., 2012)). We now describe in more detail what this means, from a developer point of view.

2.2.1 Values and Types

Being purely functional means that all computations are performed on syntactical expressions which yield values. Also, being statically typed implies that every expression and value has a type. Haskell has a few built-in types, including: Int, $Char$, $Bool$, function types, lists, tuples and the unit type[5] denoted as (). In addition, developers can define their own types. In Haskell all values are first-class, but types are not. For instance, the following are valid Haskell values:

```
1       :: Int
'h'     :: Char
```

[1] http://www.haskell.org/tutorial

[2] http://tryhaskell.org

[3] http://learnyouahaskell.com/

[4] http://www.haskell.org/haskellwiki/Introduction

[5] The unit type () is inhabit only by the unit value, also denoted as (). It can be regarded as the void type in Java.

$$
\begin{array}{ll}
(\lambda n \rightarrow n > 0) & :: Int \rightarrow Bool \\
[1,2,3] & :: [Int] \\
(\texttt{'a'}, 1) & :: (Char, Int)
\end{array}
$$

Here the :: annotation means "has type", that is, 1 has type Int, $\texttt{'h'}$ has type $Char$, the anonymous lambda expression $(\lambda n \rightarrow n > 0)$ has a function type $Int \rightarrow Bool$, $[1,2,3]$ is a list of integers with type $[Int]$, and finally, $(\texttt{'a'}, 1)$ is a 2-tuple with type $(Char, Int)$. Note that functions, lists and tuples have a special notation, but are not essentially different from user-defined types.

User-defined types Users can define their own data types using a $\mathbf{data}$ declaration. For example, a type representing a point with two integer coordinates is defined as:

$$\mathbf{data}\ IntPoint = IntPt\ Int\ Int$$

This declaration defines the $IntPoint$ type and the $IntPt$ function—known as the *data constructor*—used to construct values of such type. For types with a single data constructor, the typical convention is to use the same name for both the type and the data constructor, hence the definition would be:

$$\mathbf{data}\ IntPoint = IntPoint\ Int\ Int$$

Record syntax User-defined types can also be expressed using *record syntax*, this means that each field of a data constructor is given a name, which can be used as an accessor function, for instance we can define:

$$\mathbf{data}\ IntPoint = IntPoint\ \{getX :: Int, getY :: Int\}$$

then evaluating $getX\ (IntPoint\ 3\ 2)$ yields 3 and evaluating $getY\ (IntPoint\ 3\ 2)$ yields 2.

Parametric Polymorphism Haskell features parametric polymorphism (also known as *generics* in Java). This means that a type can be universally quantified or parametrized by other types. For instance, although lists in Haskell must be homogeneous (*e.g.* all values of the same type), there is a single parametrized type constructor (and data constructor) for lists. Lists are constructed using the : function and the empty list value $[\,]$, whose types are[6]:

$$
\begin{array}{ll}
(:) & :: a \rightarrow [a] \rightarrow [a] \\
[\,] & :: [a]
\end{array}
$$

These types are a shorthand notation of the more explicit universal quantification:

[6]Note that (:) means that : is an infix operator, just like $+$

$$(:) :: \forall a.a \rightarrow [\,a\,] \rightarrow [\,a\,]$$
$$[\,] \;\; :: \forall a.[\,a\,]$$

In these type signatures $[\,]$ denotes the list type constructor and a (and lower-case letters in general) denotes a type variable. This means that given a concrete instantiation of a, *e.g.* Int, the list type constructor will yield a concrete type, *e.g.* $[\,Int\,]$.

Polymorphic type constructors As a consequence of parametric polymorphism, **data** declarations can define type constructors with any number of type variables. For instance, points need not have only integer coordinates, but rather any single arbitrary type:

$$\mathbf{data}\ Point\ a = Point\ a\ a$$

or even more, a different type for each axis:

$$\mathbf{data}\ Point'\ a\ b = Point'\ a\ b$$

These declarations define two type constructors: $Point$ and $Point'$, which take types as arguments to produce new types. They also define the data constructors with types $a \rightarrow a \rightarrow Point\ a$ and $a \rightarrow b \rightarrow Point'\ a\ b$, respectively. For example, the value $Point\ 1\ 3$ has type $Point\ Int\ Int$, and the value $Point'\ 3$ `'z'` has type $Point'\ Int\ Char$.

Variants and recursive types A user-defined data type can encompass several *variants*, constructed by different data constructors, under the same, potentially recursive, type. For example, consider the type of a polymorphic binary tree:

$$\mathbf{data}\ BinTree\ a = Leaf\ a \mid Node\ a\ (BinTree\ a)\ (BinTree\ a)$$

This declaration defines the $BinTree$ type constructor, and the $Leaf$ and $Node$ data constructors. A binary tree with type $BinTree\ a$ is either a $Leaf$ with a value of type a, or a node with a value of the same type and two subtrees of type $BinTree\ a$.

Type synonyms Observe that we did not mention strings as part of the built-in data types. The reason is that, just like in C, a string is defined as a list of characters. Haskell provides *type synonyms* to introduce a new name for a type:

$$\mathbf{type}\ String = [\,Char\,]$$

Note that there is no data constructor—this declaration just instructs the compiler to perform a simple text substitution from $String$ to $[\,Char\,]$ when processing a source file.

2.2.2 Functions and Pattern Matching

A function is defined by giving its type signature and its implementation, for instance:

$$\begin{aligned}&\mathit{add1} :: \mathit{Int} \rightarrow \mathit{Int}\\ &\mathit{add1}\ x = x + 1\end{aligned}$$

The $\mathit{add1} :: \mathit{Int} \rightarrow \mathit{Int}$ expression is the *type signature* of function $\mathit{add1}$; and the $\mathit{add1}\ x = x + 1$ is its actual implementation, meaning that it takes argument x and returns the value $x + 1$. Functions can have polymorphic types as well, and in most simple cases explicit type signatures can be omitted thanks to the type inference mechanism of Haskell.

Pattern matching is a core language feature used to discriminate or deconstruct values according to their data constructors (either built-in or user-defined). For example, consider a function $\mathit{binTreeDepth}$ that computes the depth of a binary tree. The result of the function depends on whether the value is a Leaf or a Node. To this end, Haskell provides two ways to pattern match on values. First, the **case** expression explicitly matches a value against a set of possible patterns:

$$\begin{aligned}&\mathit{binTreeDepth} :: \mathit{BinTree}\ a \rightarrow \mathit{Int}\\ &\mathit{binTreeDepth}\ t = \mathbf{case}\ t\ \mathbf{of}\\ &\quad \mid \mathit{Leaf}\ a \qquad\qquad\quad \rightarrow 1\\ &\quad \mid \mathit{Node}\ a\ \mathit{left}\ \mathit{right} \rightarrow 1 + \mathit{max}\ (\mathit{binTreeDepth}\ \mathit{left}, \mathit{binTreeDepth}\ \mathit{right})\end{aligned}$$

The second option is specific to function definitions, and it allows to define a function by several different *equations* (note the use of the _ pattern, which signals that the value is of no interest):

$$\begin{aligned}&\mathit{binTreeDepth} :: \mathit{BinTree}\ a \rightarrow \mathit{Int}\\ &\mathit{binTreeDepth}\ (\mathit{Leaf}\ _) \qquad\qquad\quad = 1\\ &\mathit{binTreeDepth}\ (\mathit{Node}\ _\ \mathit{left}\ \mathit{right}) = 1 + \mathit{max}\ (\mathit{binTreeDepth}\ \mathit{left}, \mathit{binTreeDepth}\ \mathit{right})\end{aligned}$$

The special syntax of lists and tuples can be used for pattern matching. As an example of the former, consider the length function, which computes the size of an arbitrary list (we illustrate matching on tuples below). A list is either empty, with pattern $[\,]$, or is composed of head x with tail xs, with pattern $(x : xs)$:

$$\begin{aligned}&\mathit{length} :: [a] \rightarrow \mathit{Int}\\ &\mathit{length}\ [\,] \qquad\ = 0\\ &\mathit{length}\ (_ : xs) = 1 + \mathit{length}\ xs\end{aligned}$$

Finally, patterns can match on literal values and can be arbitrarily nested. As a contrived example, consider a function to check whether a list starts with the $(1, [\texttt{'a'}, \texttt{'b'}, \texttt{'c'}], \mathit{True})$ tuple:

$$\begin{aligned}&\mathit{startsWithTuple} :: [(\mathit{Int}, [\mathit{Char}], \mathit{Bool})] \rightarrow \mathit{Bool}\\ &\mathit{startsWithTuple}\ ((1, [\texttt{'a'}, \texttt{'b'}, \texttt{'c'}], \mathit{True}) : _) = \mathit{True}\\ &\mathit{startsWithTuple}\ _ \qquad\qquad\qquad\qquad\qquad\quad = \mathit{False}\end{aligned}$$

This last example highlights an important point: patterns are tried from top to bottom, and only the body of the first matched pattern is evaluated. A fatal runtime error is raised if no pattern matches the arguments.

Function composition and chaining Two operators are specially relevant when working with functions, and are extensively used throughout this document. Consider two functions f and g; the first operator is function composition:

$$(\circ) :: (b \to c) \to (a \to b) \to a \to c$$

which is used just like in mathematics, that is, $(f \circ g)\ x$ is equivalent to $f\ (g\ x)$. The second one is the dollar sign operator:

$$(\$) :: (a \to b) \to a \to b$$

which is typically used to avoid parentheses due to its higher precedence with respect to other functions. Essentially, $f\ \$\ e$ applies function f to the result of evaluation expression e, *e.g.* $f\ (g\ x)$ can be written as $f\ \$\ g\ x$.

2.2.3 Lexical Scoping and Local Identifiers

Identifiers in Haskell are always lexically scoped; they can be introduced either at the top level, or locally using a **let** expression or a **where** clause. For instance consider a function that sums the even numbers in a list of integers:

$$\begin{array}{l} sumEven :: [Int] \to Int \\ sumEven\ l = \mathbf{let}\ evens = filter\ even\ l\ \mathbf{in}\ sum\ evens \end{array}$$

Using **let** we define the local identifier $evens$, bound to a list with the even numbers in l. We then simply sum these numbers. Alternatively, we can write the function using a **where** clause instead of **let**:

$$\begin{array}{l} sumEven\ l = sum\ evens \\ \quad \mathbf{where}\ evens = filter\ even\ l \end{array}$$

In general choosing between **let** and **where** is a matter of programming style[7]; and we tend to prefer **where** over **let**.

2.2.4 Type Classes and Ad Hoc Polymorphism

In addition to parametric polymorphism, Haskell supports another kind of polymorphism called *ad hoc polymorphism*, also known as *overloading*. The idea is to use the same *function name* to refer to *different implementations* that are identified by the types of their arguments and return value. The classical example in Haskell is the equality function, denoted as $\equiv$. Ideally we want to use the same operator to compare equality of integers, booleans, characters, lists, or user-defined

[7] Although there are subtle differences, as discussed in http://www.haskell.org/haskellwiki/Let_vs._Where

types (note however that function equality is undecidable in general). This is achieved using *type classes* (Wadler and Blott, 1989): a type class, similar to a class in object-oriented languages, defines an abstract interface of *methods* and is parametrized by one or more types. Following our example, the Eq type class defines the $\equiv$ binary operator for equality comparisons:

$$
\begin{aligned}
&\textbf{class } Eq\ a \textbf{ where} \\
&\quad (\equiv) :: a \to a \to Bool
\end{aligned}
$$

This declaration can be read as: "if a concrete type T is registered as an *instance* of type class Eq, then it is statically known that there exists a function $(\equiv_T) :: T \to T \to Bool$". For example, to define equality comparisons for $IntPoint$s we register the type as an instance of Eq:

$$
\begin{aligned}
&\textbf{instance } Eq\ IntPoint \textbf{ where} \\
&\quad IntPoint\ x_1\ y_1 \equiv IntPoint\ x_2\ y_2 = x_1 \equiv x_2 \wedge y_1 \equiv y_2
\end{aligned}
$$

This definition relies on the fact that Int is already an instance of Eq, thus allowing us to evaluate $x_1 \equiv x_2$ and $y_1 \equiv y_2$. Most built-in types, with the notable exception of functions, are already instance of this class. Now consider the equality of binary trees:

$$
\begin{aligned}
&\textbf{instance } Eq\ a \Rightarrow BinTree\ a \textbf{ where} \\
&\quad Leaf\ x \equiv Leaf\ y = x \equiv y \\
&\quad Node\ x\ l_1\ r_1 \equiv Node\ y\ l_2\ r_2 = x \equiv y \wedge l_1 \equiv l_2 \wedge r_1 \equiv r_2
\end{aligned}
$$

Binary tree equality is defined by two equations, one for each data constructor: two $Leaf$s are equal if they hold the same value; similarly, two $Node$s are equal if they have the same value and their subtrees are equal. However, this declaration is predicated on the fact that we can check the equality of the elements in the tree. This restriction is reflected by the $Eq\ a \Rightarrow$ expression, which is a *type class constraint* (or *class constraint* or simply *constraint*). In other words, given a concrete type T, we can only declare an $Eq\ (BinTree\ T)$ instance if there is already an $Eq\ T$ instance.

Finally, it is important to remark that the type checker statically rejects any program where a type class method is used on arguments whose types are not registered instances of the class.

2.2.5 Bounded Polymorphism

A notorious characteristic of parametric polymorphism is that a function cannot do any meaningful operation on the arguments on which it is polymorphic. *Bounded polymorphism* (Cardelli and Wegner, 1985) is another form of polymorphism that restricts the universal quantification of parametric polymorphism to a set of types that are known to implement a particular interface—therefore, the operations specific to that interface can be safely used inside a bounded polymorphic function. In Haskell this is implemented using type classes and constraints; for example, the $elem$ function only requires an equality comparison to check whether a value belongs to a list:

$$
\begin{aligned}
&elem :: Eq\ a \Rightarrow a \to [a] \to Bool \\
&elem\ _\ [\,] = False \\
&elem\ a\ (x : xs) = a \equiv x \vee elem\ a\ xs
\end{aligned}
$$

The type of $elem$ can be read as: "for all types a such that a is instance of Eq, the function takes a value of type a, a list of elements of the same type, and returns a boolean". Because it is statically known that, regardless of its particular instantiation, the type a is instance of Eq, we can safely use the $\equiv$ operation in the definition of the function. This would not possible if the type were only $a \to [a] \to Bool$.

2.2.6 newtype Declarations

In addition to **data** declarations and **type** synonyms, Haskell provides a third way to define types: **newtype** declarations. A **newtype** declaration defines a type just like **data**, but is restricted to types with *exactly one data constructor with one field in it*. There are two main uses of **newtype**, the first is to allow data abstraction without having duplicate implementations. For instance:

$$\textbf{newtype}\ CharPoint = CharPoint\ (Point\ Char\ Char)$$

allows developers to export the (opaque) type $CharPoint$ while using internally $Point\ Char\ Char$.[8] The second use is to redefine the type classes on which a type is registered. The reason is that a **newtype** does not inherit the instance declarations of the underlying type. This technique is commonly used in monadic programming; we use it in Section 3.3 to define a custom monad transformer based on a standard one.

2.3 Monadic Programming in a Nutshell

> *1990 - [...]. Haskell gets some resistance due to the complexity of using monads to control side effects. Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem?"*
>
> **Fictional** quotation attributed to Philip Wadler by James Iry[9]

Monads (Moggi, 1991; Wadler, 1992) are a denotational approach to embed and reason about computational effects such as mutable state, I/O, or exception-handling in purely functional languages like Haskell. Monad transformers (Liang et al., 1995) allow the modular construction of monads that combine several effects. A monad transformer is a type constructor used to create a *monad stack* where each layer represents an effect. Monadic programming in Haskell is provided by the standard *Monad transformers library* (known as mtl), which defines a set of monad transformers that can be flexibly composed together.

A monad is defined by a type constructor m and functions $\gg=$ (called $bind$) and $return$. At the type level a monad is a regular type constructor, although conceptually we distinguish a value

[8]The restrictions on **newtype** allow the GHC compiler to optimize away the extra data constructor such that at runtime both expressions are isomorphic

[9]A Brief, Incomplete, and Mostly Wrong History of Programming Languages http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

of type a from a *computation in monad* m of type $m\ a$. Monads provide a uniform interface for computational effects, as specified in the $Monad$ type class:

$$
\begin{array}{l}
\mathbf{class}\ Monad\ m\ \mathbf{where} \\
\quad return :: a \rightarrow m\ a \\
\quad (\gg\!=)\ \ :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b
\end{array}
$$

Here $return$ promotes a value of type a into a computation of type $m\ a$, and $\gg\!=$ is a pipeline operator that takes a computation, extracts its value, and applies an action to produce a new computation. The precise meanings for $return$ and $\gg\!=$ are specific to each monad. The computational effect of a monad is "hidden" in the definition of $\gg\!=$, which imposes a sequential evaluation where the effect is performed at each step. To avoid cluttering caused by using $\gg\!=$ Haskell provides the **do**-notation, which directly translates to chained applications of $\gg\!=$. The $x \leftarrow k$ expression binds identifier x with the value extracted from performing computation k for the rest of a **do** block.[10]

The simplest monad is the *identity monad*, denoted as $\mathbb{I}$, which has no computational effect:

$$
\begin{array}{l}
\mathbf{newtype}\ \mathbb{I}\ a = \mathbb{I}\ a \\
\mathbf{instance}\ Monad\ \mathbb{I}\ \mathbf{where} \\
\quad return\ a \quad\ = \mathbb{I}\ a \\
\quad (\mathbb{I}\ a) \gg\!= f = f\ a
\end{array}
$$

A monad transformer is defined by a type constructor t and the $lift$ operation, as specified in the $MonadTrans$ type class:

$$
\begin{array}{l}
\mathbf{class}\ MonadTrans\ t\ \mathbf{where} \\
\quad lift :: m\ a \rightarrow t\ m\ a
\end{array}
$$

The purpose of $lift$ is to promote a computation from an inner layer of the monad stack, of type $m\ a$, into a computation in the monad defined by the complete stack, with type $t\ m\ a$. Each transformer t must declare in an effect-specific way how to make $t\ m$ an instance of the $Monad$ class.

2.3.1 Plain Monadic Programming

To illustrate monadic programming we first describe the use of the state monad transformer $StateT$, denoted as $\mathbb{S}_{\mathrm{T}}$, whose computational effect is to thread a value with read-write access.

$$
\begin{array}{l}
\mathbf{newtype}\ \mathbb{S}_{\mathrm{T}}\ s\ m\ a = \mathbb{S}_{\mathrm{T}}\ (s \rightarrow m\ (a, s)) \\
eval\mathbb{S}_{\mathrm{T}} :: \mathbb{S}_{\mathrm{T}}\ s\ m\ a \rightarrow s \rightarrow m\ a
\end{array}
$$

A $\mathbb{S}_{\mathrm{T}}\ s\ m\ a$ computation is a function that takes an initial state of type s and returns a computation in the underlying monad m with a pair containing the resulting value of type a, and a

[10] $x \leftarrow k$ performs the effect in k, while **let** $x = k$ does not (the **in** keyword is omitted when using **let** inside a **do** expression).

potentially modified state of type s. The $eval\mathbb{S}_{\mathrm{T}}$ function evaluates a $\mathbb{S}\ s\ m\ a$ computation using an initial state s and yields only the returning computation $m\ a$. In addition, functions $get\mathbb{S}_{\mathrm{T}}$ and $put\mathbb{S}_{\mathrm{T}}$ allow to retrieve and update the state inside a computation, respectively[11].

$$
\begin{aligned}
&get\mathbb{S}_{\mathrm{T}} :: Monad\ m \Rightarrow \mathbb{S}_{\mathrm{T}}\ s\ m\ s \\
&get\mathbb{S}_{\mathrm{T}} \quad = \mathbb{S}_{\mathrm{T}}\ \$\ \lambda s \rightarrow return\ (s, s) \\
&put\mathbb{S}_{\mathrm{T}} :: Monad\ m \Rightarrow s \rightarrow \mathbb{S}_{\mathrm{T}}\ s\ m\ () \\
&put\mathbb{S}_{\mathrm{T}}\ s' = \mathbb{S}_{\mathrm{T}}\ \$\ \lambda_ \rightarrow return\ ((), s')
\end{aligned}
$$

Note that both functions get the current state from some previous operation ($\gg\!=$ or $eval\mathbb{S}_{\mathrm{T}}$). The difference is that $get\mathbb{S}_{\mathrm{T}}$ returns this value and keeps the previous state unchanged, whereas $put\mathbb{S}_{\mathrm{T}}$ replaces the previous state with its argument.

Example Application Consider a mutable queue of integers with operations to enqueue and dequeue its elements. To implement it we will define a monad stack M_1, which threads a list of integers using the $\mathbb{S}_{\mathrm{T}}$ transformer on top of the identity monad. We also define $runM_1$, which initializes the queue with an empty list, and returns only the resulting value of a computation in M_1.

$$
\begin{aligned}
&\mathbf{type}\ M_1 = \mathbb{S}_{\mathrm{T}}\ [Int]\ \mathbb{I} \\
&runM_1 :: M_1\ a \rightarrow a \\
&runM_1\ c = run\mathbb{I}\ \$\ eval\mathbb{S}_{\mathrm{T}}\ c\ [\,]
\end{aligned}
$$

The implementation of the queue operations using M_1 is simple, we just enqueue elements at the end of the list and dequeue elements from the beginning.

$$
\begin{aligned}
&enqueue_1 :: Int \rightarrow M_1\ () \\
&enqueue_1\ n = \mathbf{do}\ queue \leftarrow get\mathbb{S}_{\mathrm{T}} \\
&\qquad\qquad\qquad put\mathbb{S}_{\mathrm{T}}\ \$\ queue +\!\!+ [n] \\
&dequeue_1 :: M_1\ Int \\
&dequeue_1 = \mathbf{do}\ queue \leftarrow get\mathbb{S}_{\mathrm{T}} \\
&\qquad\qquad\quad put\mathbb{S}_{\mathrm{T}}\ \$\ tail\ queue \\
&\qquad\qquad\quad return\ \$\ head\ queue
\end{aligned}
$$

Handling Error Scenarios The above implementation of $dequeue_1$ terminates with a runtime error if it is performed on an empty queue, because $tail$ fails when applied on an empty list. To provide an error-handling mechanism we use the error monad transformer $ErrorT$, denoted as $\mathbb{E}_{\mathrm{T}}$.

$$
\begin{aligned}
&\mathbf{newtype}\ \mathbb{E}_{\mathrm{T}}\ e\ m\ a = \mathbb{E}_{\mathrm{T}}\ m\ (Either\ e\ a) \\
&run\mathbb{E}_{\mathrm{T}} :: Monad\ m \Rightarrow \mathbb{E}_{\mathrm{T}}\ e\ m\ a \rightarrow m\ (Either\ e\ a)
\end{aligned}
$$

The type $Either\ e\ a$ represents two possible values: a $Left\ e$ value or a $Right\ a$ value. In this case the convention is that a $Left\ e$ value is treated as an error, while a $Right\ a$ value is considered a

[11]Note the use of \$, here and throughout the rest of the paper, to avoid extra parentheses.

successful operation. Then, the $throw\mathbb{E}_T$ and $catch\mathbb{E}_T$ operations can be defined to raise and handle exceptions.

$$throw\mathbb{E}_T :: Monad\ m \Rightarrow e \rightarrow \mathbb{E}_T\ e\ m\ a$$
$$throw\mathbb{E}_T\ e = \mathbb{E}_T\ \$\ return\ (Left\ e)$$

$$catch\mathbb{E}_T :: Monad\ m \Rightarrow \mathbb{E}_T\ e\ m\ a \rightarrow (e \rightarrow \mathbb{E}_T\ e\ m\ a) \rightarrow \mathbb{E}_T\ e\ m\ a$$

$$\begin{aligned} m\ \text{`}catch\mathbb{E}_T\text{`}\ h = \mathbb{E}_T\ \$\ \mathbf{do}\ & a \leftarrow run\mathbb{E}_T\ m \\ & \mathbf{case}\ a\ \mathbf{of} \\ & \quad Left\ err \ \ \rightarrow run\mathbb{E}_T\ (h\ err) \\ & \quad Right\ val \rightarrow return\ (Right\ val) \end{aligned}$$

Observe that $catch\mathbb{E}_T$ is intended to be used as an infix operator, where the first argument is the protected expression that would be inside a *try* block in Java, while the second argument is the exception handler.

Combining State and Error-Handling Effects To implement a queue with support for exceptions we first define a new monad stack M_2 that combines both effects (using *String*s as error messages):

$$\mathbf{type}\ M_2 = \mathbb{S}_T\ [Int]\ (\mathbb{E}_T\ String\ \mathbb{I})$$
$$runM_2\ c = run\mathbb{I}\ \$\ run\mathbb{E}_T\ \$\ eval\mathbb{S}_T\ c\ [\,]$$

Then we define the $enqueue_2$ operation as before, but using M_2:

$$enqueue_2 :: Int \rightarrow M_2\ ()$$
$$\begin{aligned} enqueue_2\ n = \mathbf{do}\ & queue \leftarrow get\mathbb{S}_T \\ & put\mathbb{S}_T\ \$\ queue +\!\!+ [n] \end{aligned}$$

However, the straightforward definition of $dequeue_2$ fails with a typing error:

$$dequeue_2 :: M_2\ Int$$
$$\begin{aligned} dequeue_2 = \mathbf{do}\ & queue \leftarrow get\mathbb{S}_T \\ & \mathbf{if}\ null\ queue \\ & \quad \mathbf{then}\ throw\mathbb{E}_T\ \texttt{"Queue is empty"} \quad \text{-- typing error} \\ & \quad \mathbf{else}\ \ \mathbf{do}\ put\mathbb{S}_T\ \$\ tail\ queue \\ & \qquad\qquad\quad return\ \$\ head\ queue \end{aligned}$$

The problem is that $throw\mathbb{E}_T$ returns a computation whose type is $(\mathbb{E}_T\ String\ \mathbb{I})\ Int$, but the return type of $dequeue_2$ is $(\mathbb{S}_T\ [Int]\ (\mathbb{E}_T\ String\ \mathbb{I}))\ Int$.

Explicit Lifting in the Monad Stack Using *lift* we can reuse a function intended for an inner layer on the stack, like $throw\mathbb{E}_T$. The number of *lift* calls corresponds to the distance between the top of the stack and the inner layer of the stack. Hence for $dequeue_2$ we need only one call to *lift*:

$dequeue_2 :: M_2 \; Int$
$dequeue_2 = \textbf{do} \; queue \leftarrow get\mathbb{S}_T$
$\quad \textbf{if} \; null \; queue$
$\quad\quad \textbf{then} \; (lift \circ throw\mathbb{E}_T)$ `"Queue is empty"`
$\quad\quad \textbf{else do} \; put\mathbb{S}_T \; \$ \; tail \; queue$
$\quad\quad\quad return \; \$ \; head \; queue$

Although we managed to implement a queue with support for both effects, this is not satisfactory from a software engineering point of view. The reason is that plain monadic programming and explicit liftings produce a strong coupling between functions and particular monad stacks, hampering reusability and maintainability of the software.

2.3.2 Polymorphism on the Monad Stack

To address the coupling of functions with particular monad stacks and to expand the notion of monads as a uniform interface for computational effects, the mtl defines a set of type classes associated to particular effects. This way, monadic functions can impose constraints in the monad stack using these type classes instead of relying on a specific stack. These class constraints can be seen as *families of monads*, making a function polymorphic with respect to the concrete monadic stack used to evaluate it.

State Operations The $MonadState$ type class, denoted as $\mathbb{S}_M$, defines the interface for state-related operations, and $\mathbb{S}_T$ is the canonical instance of this class.[12]

$\textbf{class} \; Monad \; m \Rightarrow \mathbb{S}_M \; s \; m \mid m \rightarrow s \; \textbf{where}$
$\quad get :: m \; s$
$\quad put :: s \rightarrow m \; ()$

Error-Handling Operations The $MonadError$ type class, denoted as $\mathbb{E}_M$, defines the standard interface for error-handling operations, with $\mathbb{E}_T$ as its canonical instance.

$\textbf{class} \; Monad \; m \Rightarrow \mathbb{E}_M \; e \; m \mid m \rightarrow e \; \textbf{where}$
$\quad throwError :: e \rightarrow m \; a$
$\quad catchError :: m \; a \rightarrow (e \rightarrow m \; a) \rightarrow m \; a$

Implicit Lifting in the Monad Stack Going back to our example of the integer queue, the implementation using class contraints now is as follows:

[12] Expression $m \rightarrow s$ denotes a *functional dependency* (Jones, 2000), which means that the type of m determines the type of s, allowing a more precise control of type inference.

$$
\begin{array}{l}
\mathit{enqueue} :: (\mathit{Monad}\ m, \mathbb{S}_{\mathrm{M}}\ [\mathit{Int}]\ m) \Rightarrow \mathit{Int} \rightarrow m\ ()\\
\mathit{enqueue}\ n = \mathbf{do}\ \mathit{queue} \leftarrow \mathit{get}\\
\qquad\qquad\qquad\quad \mathit{put}\ \$\ \mathit{queue} +\!\!+ [n]\\
\mathit{dequeue} :: (\mathit{Monad}\ m, \mathbb{S}_{\mathrm{M}}\ [\mathit{Int}]\ m, \mathbb{E}_{\mathrm{M}}\ \mathit{String}\ m) \Rightarrow m\ \mathit{Int}\\
\mathit{dequeue} = \mathbf{do}\ \mathit{queue} \leftarrow \mathit{get}\\
\qquad\qquad\quad \mathbf{if}\ \mathit{null}\ \mathit{queue}\\
\qquad\qquad\qquad \mathbf{then}\ \mathit{throwError}\ \texttt{"Queue is empty"}\\
\qquad\qquad\qquad \mathbf{else\ do}\ \mathit{put}\ \$\ \mathit{tail}\ \mathit{queue}\\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ \$\ \mathit{head}\ \mathit{queue}
\end{array}
$$

Observe that the functions are defined in terms of an abstract monad m, which is required to be an instance of $\mathbb{S}_{\mathrm{M}}$, for insertions; and both $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$ for retrieving values. Also note that *lift* is not required to use *throwError* in *dequeue*. The reason is that using type classes, like $\mathbb{S}_{\mathrm{M}}$ or $\mathbb{E}_{\mathrm{M}}$, an operation is automatically routed to *the first layer of the monad stack that is instance of the respective class*. The mtl defines implicit liftings between its transformers, by defining several class instances for each of them. Because of this, M_2 is instance of both $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$.

The major limitation of implicit liftings is that it *only* chooses the first layer of a given effect. Consequently, when more than one instance of the same effect are used, *e.g.* two state transformers to hold the state of a queue and a stack, the parts of the program that access inner layers must use explicit lifting.

Explicit and implicit lifting are the standard mechanism in Haskell to handle the monad stack. The mechanism used to handle the monad stack directly determines the expressiveness of the type-based reasoning techniques, and other properties like modularity and reusability of components. This is discussed in detail in Chapter 7 and Chapter 8; in particular we show that the standard mechanism falls short to deal with interference of multiple aspects. Then we use *monad views*, a novel mechanism for managing the monad stack recently developed by Schrijvers and Oliveira Schrijvers and Oliveira (2011), to propose another approach to address this situation.

Part I

Design and Type Safety

Chapter 3

Introducing Aspects

The fundamental premise for aspect-oriented programming in functional languages is that function applications need to be subject to aspect weaving. We introduce the term *open application* to refer to a function application that generates a join point, and consequently, can be woven.

Open Function Applications Opening all function applications in a program or only a few selected ones is both a language design question and an implementation question. At the design level, this is the grand debate about *obliviousness* in aspect-oriented programming. Opening all applications is more flexible, but can lead to fragile aspects and unwanted encapsulation breaches. At the implementation level, opening all function applications requires either a preprocessor or runtime support.

For now, we focus on *quantification*—through pointcuts—and opt for a conservative design in which open applications are realized *explicitly* using the $\#$ operator: $f \# 2$ is the same as $f\ 2$, except that the application generates a join point that is subject to aspect weaving. We will come back to obliviousness in Section 5.3, showing how different answers can be provided within the context of our proposal.

Monadic Setting Our approach to introduce aspects in a pure functional programming language like Haskell can be realized without considering effects. Nevertheless, most interesting applications of aspects rely on computational effects (*e.g.* tracing, memoization, exception handling, etc.). We therefore adopt a monadic setting from the start. Also, as we describe in Part II of the thesis, this setting allows us to exploit the approach of EffectiveAdvice (Oliveira et al., 2010) and other monadic reasoning mechanisms in order to perform type-based reasoning about effects in presence of aspects.

Illustration As a basic example, recall the $enqueue$ function (Section 2.3.2) and consider the $uniqueAdv$ advice, which enforces that the argument is only passed to $proceed$ if it is not already present in the underlying list l (*e.g.* to avoid repeated elements when representing a set using a list);

$$
\begin{aligned}
uniqueAdv\ proceed\ arg = \mathbf{do}\ & l \leftarrow get \\
& \mathbf{if}\ elem\ arg\ l \\
& \quad \mathbf{then}\ return\ () \\
& \quad \mathbf{else}\ \ proceed\ arg
\end{aligned}
$$

Then, in $program$ we $deploy$ an $aspect$ that reacts to applications of $enqueue$. This is specified using the pointcut $pcCall\ enqueue$.

$$
\begin{aligned}
program\ n\ m = \mathbf{do}\ & deploy\ (aspect\ (pcCall\ enqueue)\ uniqueAdv) \\
& enqueue \# n \\
& enqueue \# m \\
& showQueue
\end{aligned}
$$

Evaluating $program\ \ 1\ \ 2$ returns a string representation `"[1,2]"` with both elements, whereas $program\ \ 1\ \ 1$ returns `"[1]"` with only one element. Indeed, both results are as expected. As shown in this example, aspects consist of a pointcut/advice pair and are created with $aspect$, and deployed with $deploy$.

Our development of AOP simply relies on defining aspects (pointcuts, advices), the underlying aspect environment together with the operations to deploy and undeploy aspects, and open function application. The remainder of this chapter presents these elements. Then, Chapter 4 concentrates on the main challenge: properly typing pointcuts and ensuring type soundness of pointcut/advice bindings. We conclude the first part of the thesis with a general discussion about the model in Chapter 5 and a review of related work in Chapter 6.

3.1 Join Point Model

The support for crosscutting provided by an aspect-oriented programming language lies in its *join point model* (Masuhara et al., 2003). A join point model is composed by three elements: *join points* that represents the (dynamic) steps in the execution of a program that aspects can affect, a *means of identifying* join points—here, pointcuts—and a *means of effecting* at join points—here, advices.

Join Points Join points are function applications. A join point JP contains a function of type $a \rightarrow m\ b$, and an argument of type a. The monad m denotes the underlying computational effect stack. Note that this means that only functions that are properly lifted to a monadic context can be advised. In addition, in order for pointcuts to be able to reason about the type of advised functions, we require the functions to be $PolyTypeable$.[1]

$$
\mathbf{data}\ JP\ m\ a\ b = (Monad\ m, PolyTypeable\ (a \rightarrow m\ b)) \Rightarrow JP\ (a \rightarrow m\ b)\ a
$$

[1] Haskell provides the $Typeable$ class to introspect monomorphic types. $PolyTypeable$ is an extension that supports both monomorphic and polymorphic types.

From now on we omit the type constraints related to $PolyTypeable$ (the $PolyTypeable$ constraint on a type is required each time the type has to be inspected dynamically; exact occurrences of this constraint can be found in the implementation).

Pointcuts A pointcut is a predicate on the current join point. It is used to identify join points of interest. A pointcut simply returns a boolean to indicate whether it matches the given join point.

$$\textbf{data}\ PC\ m\ a\ b = Monad\ m \Rightarrow PC\ (\forall a'\ b'.m\ (JP\ m\ a'\ b' \rightarrow m\ Bool))$$

A pointcut is represented as a value of type $PC\ m\ a\ b$. Types a and b are used to ensure type safety, as discussed in Section 4.1.1. The predicate itself is a function with polymorphic type $\forall a'\ b'.m\ (JP\ m\ a'\ b' \rightarrow m\ Bool)$, meaning it has access to the monad stack. The $\forall$ declaration quantifies on type variables a' and b' (using rank-2 types) because a pointcut should be able to match against any join point, regardless of the specific types involved (we come back to this in Section 4.1.1).

Pointcut Language We provide two basic pointcut designators, $pcCall$ and $pcType$, as well as logical pointcut combinators, $pcOr$, $pcAnd$, and $pcNot$. A pointcut $pcType\ f$ matches all open applications to functions that have a type compatible with f (see Section 4.1.1 for a precise definition), and a pointcut $pcCall\ f$ matches all open applications to f.

$$\begin{aligned}
&pcType\ f = PC\ (typePred\ (polyTypeOf\ f))\\
&\quad \textbf{where}\ typePred\ t\ \ = return\ \$\ \lambda jp \rightarrow return\ (compareType\ t\ jp)\\
&pcCall\ f = PC\ (callPred\ f\ (polyTypeOf\ f))\\
&\quad \textbf{where}\ callPred\ f\ t = return\ \$\ \lambda jp \rightarrow return\ (compareFun\ f\ jp \wedge compareType\ t\ jp)
\end{aligned}$$

In both cases we use the $polyTypeOf$ function (provided by $PolyTypeable$) to obtain the type representation of function f, and compare it to the type of the function in the join point using $compareType$. Additionally, to implement $pcCall$ we require a notion of function equality[2]. This is used in $compareFun$ to compare the function in the join point with the given function f. Note that in $pcCall$ we also need to perform a type comparison, using $compareType$. This is because under the choosen notion of equality a polymorphic function whose type variables are instantiated in one way is equal to the same function but with type variables instantiated in some other way (*e.g.* $id :: Int \rightarrow Int$ is equal to $id :: Float \rightarrow Float$).

Users can define their own pointcut designators. For instance, we can define control-flow pointcuts like AspectJ's $cflow$ (described in Section 9.1), data flow pointcuts (Masuhara and Kawauchi, 2003), pointcuts that rely on the trace of execution (Douence et al., 2005) (Section 8.1), etc.

Advice An advice is a function that executes in place of a join point matched by a pointcut. This replacement is similar to open recursion in EffectiveAdvice (Oliveira et al., 2010). An advice

[2]For this notion of function equality, we use the $StableNames$ API, which relies on pointer comparison. See Section 5.1 for discussion on the issues of this approach.

receives a function (known as the *proceed* function) and returns a new function of the same type (which may or may not apply the original *proceed* function internally). We introduce a type alias for advice:

$$\mathbf{type}\ \mathit{Advice}\ m\ a\ b = (a \to m\ b) \to a \to m\ b$$

For instance, the type $\mathit{Monad}\ m \Rightarrow \mathit{Advice}\ m\ \mathit{Int}\ \mathit{Int}$ is a synonym for the type $\mathit{Monad}\ m \Rightarrow (\mathit{Int} \to m\ \mathit{Int}) \to \mathit{Int} \to m\ \mathit{Int}$. For a given advice of type $\mathit{Advice}\ m\ a\ b$, we call $a \to m\ b$ the ***advised type*** of the advice.

Aspects An aspect is a first-class value binding together a pointcut and an advice. Supporting first-class aspects is important: it makes it possible to support aspect factories, separate creation and deployment/undeployment of aspects, exporting opaque, self-contained aspects as single units, etc. We introduce a data definition for aspects, parametrized by a monad m (which has to be the same in the pointcut and advice):

$$\mathbf{data}\ \mathit{Aspect}\ m\ a\ b\ c\ d = \mathit{Aspect}\ (\mathit{PC}\ m\ a\ b)\ (\mathit{Advice}\ m\ c\ d)$$

We defer the detailed definition of Aspect with its type class constraints to Section 4.1.2, when we address the issue of safe pointcut/advice binding.

3.2 Aspect Deployment

Aspect Environment The list of aspects that are deployed at a given point in time is known as the *aspect environment*. To be able to define the type $\mathit{AspectEnv}$ as an heterogenous list of aspects, we use an existentially-quantified[3], data $\mathit{EAspect}$ that hides the type parameters of Aspect:[4]

$$\mathbf{data}\ \mathit{EAspect}\ m = \forall a\ b\ c\ d.\mathit{EAspect}\ (\mathit{Aspect}\ m\ a\ b\ c\ d)$$
$$\mathbf{type}\ \mathit{AspectEnv}\ m = [\mathit{EAspect}\ m]$$

This environment can be either fixed initially and used globally (Masuhara et al., 2003), as in AspectJ, or it can be handled dynamically, as in AspectScheme (Dutchyn et al., 2006). Different scoping strategies are possible when dealing with dynamic deployment (Tanter, 2008). Because we are in a monadic setting, we can pass the aspect environment implicitly using a monad. An open function application can then trigger the set of currently-deployed aspects by retrieving these aspects from the underlying monad.

There are a number of design options for the aspect environment, depending on the kind of aspect deployment that is desired. Following the *Reader* monad, we can provide a fixed aspect

[3]In Haskell an existentially-quantified data type is declared using $\forall$ before the data constructor

[4]Because we cannot anticipate a fixed set of class constraints for deployed aspects, we left the type parameters unconstrained. Aspects with ad-hoc polymorphism have to be instantiated before deployment to statically solve each remaining type class constraint (see Section 5.2 for more details).

environment, and add the ability to deploy an aspect for the dynamic extent of an expression, similarly to the $local$ method of the $Reader$ monad. We can also adopt a state-like monad, in order to support dynamic aspect deployment and undeployment with global scope. Without loss of generality, we go for the latter.

The $\mathbb{A}_{\mathrm{T}}$ Monad Transformer Because we are interested in using arbitrary computational effects in programs, we define the aspect environment through a *monad transformer* (Section 2.3), which allows the programmer to construct a monad stack of effects. The $\mathbb{A}_{\mathrm{T}}$ monad transformer is defined as follows:

$$\mathbf{newtype}\ \mathbb{A}_{\mathrm{T}}\ m\ a = \mathbb{A}_{\mathrm{T}}\ (\mathbb{S}_{\mathrm{T}}\ (\mathit{AspectEnv}\ (\mathbb{A}_{\mathrm{T}}\ m))\ m\ a)\ \mathbf{deriving}\ (\mathit{Monad})$$

To define the $\mathbb{A}_{\mathrm{T}}$ transformer we reuse the $\mathbb{S}_{\mathrm{T}}$ data constructor, because the $\mathbb{A}_{\mathrm{T}}$ transformer is essentially a state transformer (Section 2.3.1) that threads the aspect environment. Using the $\mathit{GeneralizedNewtypeDeriving}$ extension of GHC, we can automatically derive $\mathbb{A}_{\mathrm{T}}$ as an instance of Monad. We also define a proper instance of $\mathit{MonadTrans}$ (not shown here), and implicit liftings for the standard monad transformers of the MTL.[5] Observe that the aspect environment is bound to the same monad $\mathbb{A}_{\mathrm{T}}\ m$, in order to provide aspects with access to open applications.

Dynamic Aspect Deployment We now define the functions for dynamic deployment, which simply add and remove an aspect from the aspect environment:

$$\begin{aligned}
&\mathit{deploy}, \mathit{undeploy} :: \mathit{EAspect}\ (\mathbb{A}_{\mathrm{T}}\ m) \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ ()\\
&\mathit{deploy}\ \mathit{asp}\ \ \ \ = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda \mathit{aenv} \rightarrow \mathit{return}\ ((), \mathit{asp} : \mathit{aenv})\\
&\mathit{undeploy}\ \mathit{asp} = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda \mathit{aenv} \rightarrow \mathit{return}\ ((), \mathit{deleteAsp}\ \mathit{asp}\ \mathit{aenv})
\end{aligned}$$

Finally, in order to extract the computation of the underlying monad from an $\mathbb{A}_{\mathrm{T}}$ computation we define the $\mathit{run}\mathbb{A}_{\mathrm{T}}$ function, with type $\mathit{Monad}\ m \Rightarrow \mathbb{A}_{\mathrm{T}}\ m\ a \rightarrow m\ a$ (similar to $\mathit{eval}\mathbb{S}_{\mathrm{T}}$ in the state monad transformer), that runs a computation in an empty initial aspect environment. For instance, in the initial example of the $\mathit{enqueue}$ function, we can define a client as follows:

$$\mathit{client}\ n\ m = \mathit{run}\mathbb{I}\ (\mathit{run}\mathbb{A}_{\mathrm{T}}\ (\mathit{program}\ n\ m))$$

3.3 Aspect Weaving

Aspect weaving is triggered by open applications, that is, applications performed with the $\#$ operator, for instance $f \# x$.

[5] In the rest of this presentation we use the same technique to define our custom monad transformers, hence we omit the **deriving** clauses and standard instance definitions, like $\mathit{MonadTrans}$.

Open Applications We introduce a type class *OpenApp* that declares the # operator. This makes it possible to overload # in certain contexts, and it can be used to declare constraints on monads to ensure that the operation is available in a given context.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{OpenApp}\ m\ \textbf{where} \\
\quad (\#) :: (a \to m\ b) \to a \to m\ b
\end{array}
$$

The # operator takes a function of type $a \to m\ b$ and returns a (woven) function with the same type. Any monad composed with the $\mathbb{A}_{\mathrm{T}}$ transformer has open application defined:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{OpenApp}\ (\mathbb{A}_{\mathrm{T}}\ m)\ \textbf{where} \\
\quad f \# a = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda \mathit{aenv} \to \textbf{do} \\
\qquad\qquad (\mathit{woven_f}, \mathit{aenv}') \leftarrow \mathit{weave}\ f\ \mathit{aenv}\ \mathit{aenv}\ (\mathit{newjp}\ f\ a) \\
\qquad\qquad \mathit{run}\ (\mathit{woven_f}\ a)\ \mathit{aenv}'
\end{array}
$$

An open application results in the creation of a join point, $\mathit{newjp}\ f\ a$, that represents the application of f to a. The join point is then used to determine which aspects in the environment match, produce a new function that combines all the applicable advices, and apply that function to the original argument.

Weaving The function to use at a given join point is produced by the *weave* function:

$$
\begin{array}{l}
\mathit{weave} :: \mathit{Monad}\ m \Rightarrow (a \to \mathbb{A}_{\mathrm{T}}\ m\ b) \to \mathit{AspectEnv}\ (\mathbb{A}_{\mathrm{T}}\ m) \to \\
\quad \mathit{AspectEnv}\ (\mathbb{A}_{\mathrm{T}}\ m) \to \mathit{JP}\ (\mathbb{A}_{\mathrm{T}}\ m)\ a\ b \to m\ (a \to \mathbb{A}_{\mathrm{T}}\ m\ b, \mathit{AspectEnv}\ (\mathbb{A}_{\mathrm{T}}\ m)) \\
\mathit{weave}\ f\ [\,]\ \mathit{fenv}\ _ \qquad\qquad = \mathit{return}\ (f, \mathit{fenv}) \\
\mathit{weave}\ f\ (\mathit{asp} : \mathit{asps})\ \mathit{fenv}\ \mathit{jp} = \textbf{case}\ \mathit{asp}\ \textbf{of}\ \mathit{EAspect}\ (\mathit{Aspect}\ \mathit{pc}\ \mathit{adv}) \to \\
\quad \textbf{do}\ (\mathit{match}, \mathit{fenv}') \leftarrow \mathit{apply_pc}\ \mathit{pc}\ \mathit{jp}\ \mathit{fenv} \\
\qquad \mathit{weave}\ (\textbf{if}\ \mathit{match}\ \textbf{then}\ \mathit{apply_adv}\ \mathit{adv}\ f\ \textbf{else}\ f)\ \mathit{asps}\ \mathit{fenv}'\ \mathit{jp}
\end{array}
$$

The *weave* function is defined recursively on the aspect environment. For each aspect, it applies the pointcut to the join point. It then uses either the partial application of the advice to f if the pointcut matches, or f otherwise[6], to keep on weaving on the rest of the aspect list. This definition is a direct adaptation of AspectScheme's weaving function (Dutchyn et al., 2006), and is also a *monadic weaver* (Tabareau, 2012) that supports modular language extensions (in Chapter 9 we show how to exploit this feature).

Applying Advice As we have seen, the aspect environment has type $\mathit{AspectEnv}\ m$, meaning that the type of the advice function is hidden. Therefore, advice application requires *coercing* the advice to the proper type in order to apply it to the function of the join point:

[6] *apply_pc* checks whether the pointcut matches the join point and returns a boolean and a potentially modified aspect environment. Note that *apply_pc* is evaluated in the full aspect environment *fenv*, instead of the decreasing $(\mathit{asp} : \mathit{asps})$ argument.

$$apply_adv :: Advice\ m\ a\ b \rightarrow t \rightarrow t$$
$$apply_adv\ adv\ f = (unsafeCoerce\ adv)\ f$$

The operation *unsafeCoerce* of Haskell is (unsurprisingly) unsafe and can yield segmentation faults or arbitrary results. To recover safety, we could insert a runtime type check with *compareType* just before the coercion. We instead make aspects type safe such that we can prove that the particular use of *unsafeCoerce* in *apply_adv* is *always* safe.

To summarize, the join point model described in this chapter is relatively simple: join points represent open function applications and their arguments, pointcuts are predicates over arbitrary join points, advices are functions that receive the original join point computation as their *proceed* argument, and aspects are pointcut/advice pairs. Upon an open application the weaving function constructs and applies a function that combines all applicable advices at the corresponding join point. Unfortunately the use of existential quantification, in order to provide an heterogeneous aspect environment, forces us to perform an unsafe cast operation that—unless proven otherwise—breaks the type soundness of the language. The following chapter describes how we deal with the challenge of ensuring type soundness in the presence of aspects. The key concept of our solution consists in using an anti-unification algorithm at the level of types to check that an advice is always applied at join points of the proper type.

Chapter 4

Type Safety

In this chapter we describe how to ensure type soundness in the presence of aspects. We start with an informal discussion (Section 4.1), highlighting the issues that arise from a lack of soundness and giving an intuitive presentation of our solution. Then we formally prove the safety of our approach (Section 4.2).

4.1 Typing Aspects, Informally

Ensuring type soundness in the presence of aspects consists in ensuring that an advice is always applied at a join point of the proper type. Note that by "the type of the join point", we refer to the type of the function being applied at the considered join point.

4.1.1 Typing Pointcuts

The intermediary between a join point and an advice is the pointcut, whose proper typing is therefore crucial. The type of a pointcut as a predicate over join points does not convey any information about the types of join points it matches. To keep this information, we use *phantom type variables* a and b in the definition of PC:

$$\mathbf{data}\ PC\ m\ a\ b = Monad\ m \Rightarrow PC\ (\forall a'\ b'.m\ (JP\ m\ a'\ b' \rightarrow m\ Bool))$$

A phantom type variable is a type variable that is not used on the right hand-side of the data type definition. The use of phantom type variables to type embedded languages was first introduced by Leijen and Meijer to type an embedding of SQL in Haskell (Leijen and Meijer, 1999); it makes it possible to "tag" extra type information on data. In our context, we use it to add the information about the type of the join points matched by a pointcut: $PC\ m\ a\ b$ means that a pointcut can match applications of functions of type $a \rightarrow m\ b$. We call this type the *matched type* of the pointcut. Pointcut designators are in charge of specifying the matched type of the pointcuts they produce.

Least General Types Because a pointcut potentially matches many join points of different types, the matched type must be a *more general type*. For instance, consider a pointcut that matches applications of functions of type $Int \to m\ Int$ and $Float \to m\ Int$. Its matched type is the parametric type $a \to m\ Int$. Note that this is in fact the *least general type* of both types.[1] Another more general candidate is $a \to m\ b$, but the least general type conveys more precise information. As a concrete example, below is the type signature of the $pcCall$ pointcut designator:

$$pcCall :: Monad\ m \Rightarrow (a \to m\ b) \to PC\ m\ a\ b$$

Comparing Types The type signature of the $pcType$ pointcut designator is the same as that of $pcCall$:

$$pcType :: Monad\ m \Rightarrow (a \to m\ b) \to PC\ m\ a\ b$$

However, suppose that f is a function of type $Int \to m\ a$. We want the pointcut $pcType\ f$ to match applications of functions of more specific types, such as $Int \to m\ Int$ and $Int \to m\ Char$. This means that $compareType$ actually checks that the matched type of the pointcut is *more general* than the type of the join point. In other words, the type of a join point is *compatible* with f (as stated in Section 3.1) if it is less general than the matched type of $pcType\ f$.

Logical Combinators We use type constraints in order to properly specify the matched type of logical combinations of pointcuts. The intersection of two pointcuts matches join points that are most precisely described by the *principal unifier* of both matched types. Since Haskell supports this unification when the same type variable is used, we can simply define $pcAnd$ as follows:

$$pcAnd :: Monad\ m \Rightarrow PC\ m\ a\ b \to PC\ m\ a\ b \to PC\ m\ a\ b$$

For instance, a control flow pointcut matches any type of join point, so its matched type is $a \to m\ b$. Consequently, if f is of type $Int \to m\ a$, the matched type of $pcAnd\ (pcCall\ f)\ (pcCflow\ g)$ is $Int \to m\ a$. This is because, for any function g, the matched type of $pcCflow\ g$ is the most general type $a \to m\ b$, which does not increase the specificity of the principal unifier.

Dually, the union of two pointcuts relies on *anti-unification* (Plotkin, 1970; Reynolds, 1970), that is, the computation of the least general type of two types. Haskell does not natively support anti-unification. We exploit the fact that multi-parameter type classes can be used to define relations over types, and develop a novel type class $LeastGen$ (for *least general*) that can be used as a constraint to compute the least general type t of two types t_1 and t_2 (defined in Section 4.2):

$$\begin{aligned} &pcOr :: (Monad\ m, LeastGen\ (a \to b)\ (c \to d)\ (e \to f)) \Rightarrow \\ &\quad PC\ m\ a\ b \to PC\ m\ c\ d \to PC\ m\ e\ f \end{aligned}$$

In this definition the $LeastGen$ constraint indicates that $e \to f$ is the least general type of $a \to b$ and $c \to d$, and therefore is the matched type of the resulting pointcut. For instance, if f is of type

[1] The term *most specific generalization* is also valid, but we stick here to Plotkin's original terminology (Plotkin, 1970).

$Int \to m\ a$ and g is of type $Int \to m\ Float$, the matched type of $pcOr\ (pcCall\ f)\ (pcCall\ g)$ is $Int \to m\ a$. Implementing $LeastGen$ as a type class is a crucial element of our design because it ensures that any failure to perform anti-unification will be reported statically as a compile-time error. Moreover, it allows us to rely on the standard Haskell type class resolution system, instead of developing our own external analysis tool.

Finally, the negation of a pointcut can match join points of any type because no assumption can be made on the matched join points:

$$pcNot :: Monad\ m \Rightarrow PC\ m\ a\ b \to PC\ m\ a'\ b'$$

Observe that the type of $pcNot$ is quite restrictive. In fact, the advice of any aspect with a single $pcNot$ pointcut must be completely generic because the matched type corresponds to fresh type variables. The matched type of $pcNot$ can be made more specific using $pcAnd$ to combine it with other pointcuts with more specific types. For example, if f is of type $Int \to m\ a$ and g is of type $Int \to m\ Float$, the pointcut $pcAnd\ (pcType\ f)\ (pcNot\ (pcCall\ g))$ matches the application of all functions whose type is compatible with f, except g. The matched type of this pointcut is $Int \to m\ a$, which is more specific than the matched type of a stand-alone $pcNot$ pointcut.

Open Pointcut Language The set of pointcut designators in our language is open. User-defined pointcut designators are however responsible for properly specifying their matched types. If the matched type is incorrect or too specific, soundness is lost. For example, in Section 9.1, in order to implement the $pcCflow$ pointcut, we define an auxiliary pointcut that matches any join point:[2]

$$pcAny = PC\ (return\ (\lambda jp \to return\ True))$$

The matched type of $pcAny$ must be $a \to m\ b$ to maintain soundness. Any other type, like *e.g.* $Int \to m\ a$, is ill-typed and will eventually lead to runtime errors.

Constraining Pointcuts to Specific Types A pointcut cannot make any type assumption about the type of the join point it receives as argument. The reason for this is again the homogeneity of the aspect environment: when deploying an aspect, the type of its pointcut is hidden. At runtime, then, a pointcut is expected to be applicable to any join point. The general approach to make a pointcut safe is therefore to perform a runtime type check, as was illustrated in the definition of $pcCall$ and $pcType$ in Section 3.1. However, certain pointcuts are meant to be conjoined with others pointcuts that will first apply a sufficient type condition.

In order to support the definition of pointcuts that *require* join points to be of a given type, we provide the $RequirePC$ type:

$$\mathbf{data}\ RequirePC\ m\ a\ b = Monad\ m \Rightarrow RequirePC\ (\forall a'\ b'.m\ (JP\ m\ a'\ b' \to m\ Bool))$$

The definition of $RequirePC$ is similar to that of PC, with two important differences. First, the matched type of a $RequirePC$ is interpreted as a type *requirement*. Second, a $RequirePC$ is not a

[2]We present a simplified version of $pcAny$. In Section 9.1, this pointcut pushes the current join point into a stack, which is eventually inspected by $pcCflow$.

valid stand-alone pointcut: it has to be combined with a standard PC that enforces the proper type upfront. To safely achieve this, we overload $pcAnd$[3]:

$$\begin{aligned} & pcAnd :: (Monad\ m, LessGen\ (a \to b)\ (c \to d)) \Rightarrow \\ & \qquad PC\ m\ a\ b \to RequirePC\ m\ c\ d \to PC\ m\ a\ b \end{aligned}$$

In this case $pcAnd$ yields a standard PC pointcut and checks that the matched type of the PC pointcut is *less general* than the type expected by the $RequirePC$ pointcut. This is expressed using the constraint $LessGen$, which, as we will see in Section 4.2, is based on $LeastGen$.

To illustrate, let us define a pointcut designator $pcArgGT$ for specifying pointcuts that match when the argument at the join point is greater than a given n (of type a instance of the Ord type class):

$$\begin{aligned} & pcArgGT :: (Monad\ m, Ord\ a) \Rightarrow a \to RequirePC\ m\ a\ b \\ & pcArgGT\ n = RequirePC\ \$\ return\ \$\ \lambda jp \to return\ (unsafeCoerce\ (getJpArg\ jp) \geqslant n) \end{aligned}$$

The use of $unsafeCoerce$ to coerce the join point argument to the type a forces us to declare the Ord constraint on a when typing the returned pointcut as $RequirePC\ m\ a\ b$ (with a fresh type variable b). To get a proper pointcut, we use $pcAnd$, for instance to match all calls to $enqueue$ where the argument is greater than 10:

$$pcCall\ enqueue\ `pcAnd`\ pcArgGT\ 10$$

The $pcAnd$ combinator guarantees that a $pcArgGT$ pointcut is always applied to a join point with an argument that is indeed of a proper type: no runtime type check is necessary within $pcArgGT$, because the coercion is always safe.

4.1.2 Typing Aspects

The main typing issue we have to address consists in ensuring that a pointcut/advice binding is type safe, so that the advice application does not fail. A first idea to ensure that the pointcut/advice binding is type safe is to require the matched type of the pointcut and the advised type of the advice to be the same (or rather, unifiable):

```
   -- wrong!
data Aspect m a b = Aspect (PC m a b) (Advice m a b)
```

This approach can however yield unexpected behavior. Consider this example:

$$\begin{aligned} & idM\ x = return\ x \\ & adv :: Monad\ m \Rightarrow Advice\ (Char \to m\ Char) \end{aligned}$$

[3] The constraint is different from the previous constraint on $pcAnd$. This is possible thanks to the recent $ConstraintKinds$ extension of GHC.

$$
\begin{aligned}
&adv\ proceed\ c = proceed\ (toUpper\ c)\\
&program = \mathbf{do}\ deploy\ (aspect\ (pcCall\ idM)\ adv)\\
&\qquad x \leftarrow idM \# \texttt{'a'}\\
&\qquad y \leftarrow idM \# [True, False, True]\\
&\qquad return\ (x, y)
\end{aligned}
$$

The matched type of the pointcut $pcCall\ \ idM$ is $Monad\ \ m\ \Rightarrow\ a\ \rightarrow\ m\ \ a$. With the above definition of $Aspect$, $program$ passes the type checker because it is possible to unify a and $Char$ to $Char$. However, when evaluated, the behavior of $program$ is undefined because the advice is unsafely applied with an argument of type $[Bool]$, for which $toUpper$ is undefined.

The problem is that during type checking, the matched type of the pointcut and the advised type of the advice can be unified. Because unification is symmetric, this succeeds even if the advised type is more specific than the matched type. In order to address this, we again use the type class $LessGen$ to ensure that the matched type is less general than the advice type:

$$
\begin{aligned}
\mathbf{data}\ Aspect\ m\ a\ b\ c\ d = &(Monad\ m, LessGen\ (a \rightarrow m\ b)\ (c \rightarrow m\ d)) \Rightarrow\\
&Aspect\ (PC\ m\ a\ b)\ (Advice\ m\ c\ d)
\end{aligned}
$$

This constraint ensures that pointcut/advice bindings are type safe: the coercion performed in $apply_adv$ (Section 3.3) always succeeds. We formally prove this in the following section.

4.2 Typing Aspects, Formally

We now formally prove the safety of our approach. We start briefly summarizing the notion of type substitutions and the *is less general* relation between types. Note that we do not consider type class constraints in the definition. Then we describe a novel anti-unification algorithm implemented with type classes, on which the type classes $LessGen$ and $LeastGen$ are based. We finally prove pointcut and aspect safety, and state our main safety theorem.

4.2.1 Type Substitutions

In this section we summarize the definition of type substitutions and introduce formally the notion of least general type in a Haskell-like type system (without ad-hoc polymorphism). Thus, we have types $t ::= Int, Char, \ldots, t_1\ \rightarrow\ t_2, T\ t_1\ \ \ldots\ \ t_m$, which denote primitive types, functions, and m-ary type constructors, in addition to user-defined types. We consider a typing environment $\Gamma = (x_{\mathrm{i}} : t_{\mathrm{i}})_{\mathrm{i}\in\mathbb{N}}$ that binds variables to types.

Definition 1 (Type Substitution, from (Pierce, 2002)) A type substitution σ is a finite mapping from type variables to types. It is denoted $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$, where $\mathrm{d}om(\sigma)$ and $range(\sigma)$ are the sets of types appearing in the left-hand and right-hand sides of the mapping, respectively. It is possible for type variables to appear in $range(\sigma)$.

```
1  class LeastGen′ a b c σ_in σ_out | a b c σ_in → σ_out
2    -- Inductive case: The two type constructors match,
3    -- recursively compute the substitution for type arguments a_i,b_i.
4  instance (LeastGen′ a_1 b_1 c_1 σ_0     σ_1, . . . ,
5            LeastGen′ a_n b_n c_n σ_{n−1} σ_n,
6            T c_1 . . . c_n ∼ c)
7    ⇒ LeastGen′ (T a_1 . . . a_n) (T b_1 . . . b_n) c σ_0 σ_n
8    -- Default case: The two type constructors don't match, c has to be a variable,
9    -- either unify c with c′ if c′ ↦ (a, b), or extend the substitution with c ↦ (a, b)
10 instance (Analyze c (TVar c),
11           MapsTo σ_in c′ (a, b),
12           VarCase c′ (a, b) c σ_in σ_out)
13   ⇒ LeastGen′ a b c σ_in σ_out
```

Figure 4.1: The $LeastGen'$ type class. An instance holds if c is the least general type of a and b.

Substitutions are always applied simultaneously on a type. If σ and γ are substitutions, and t is a type, then $\sigma \circ \gamma$ is the composed substitution, where $(\sigma \circ \gamma)t = \sigma(\gamma t)$. Application of substitution on a type is defined inductively on the structure of the type.

Substitution is extended pointwise for typing environments in the following way: $\sigma(x_\mathrm{i} : t_\mathrm{i})_{\mathrm{i}\in\mathbb{N}} = (x_\mathrm{i} : \sigma t_\mathrm{i})_{\mathrm{i}\in\mathbb{N}}$. Also, applying a substitution to a term t means to apply the substitution to all type annotations appearing in t.

Definition 2 (Less General Type) We say type t_1 *is less general* than type t_2, denoted $t_1 \preceq t_2$, if there exists a substitution σ such that $\sigma t_2 = t_1$. Observe that $\preceq$ defines a partial order on types (modulo α-renaming).

Definition 3 (Least General Type) Given types t_1 and t_2, we say type t is the *least general type* iff t is the supremum of t_1 and t_2 with respect to $\preceq$.

4.2.2 Statically Computing Least General Types

In an aspect declaration, we statically check the type of the pointcut and the type of the advice to ensure a safe binding. To do this we encode an anti-unification algorithm at the type level, exploiting the type class mechanism. A multi-parameter type class $R\ \ t_1 \dots t_n$ can be seen as a *relation* R on types $t_1 \dots t_n$, and **instance** declarations as ways to inductively define this relation, in a manner very similar to logic programming.

The type classes $LessGen$ and $LeastGen$ used in Section 4.1 are defined as particular cases of the more general type class $LeastGen'$, shown in Figure 4.1. This class is defined in line 1 and is parametrized by types a, b, c, σ_in and σ_{out}. Note that σ_{out} is functionally dependent on a, b, c and σ_{in}; and that there is no **where** keyword because the class declares no operations. Both σ_in

and σ_{out} denote substitutions encoded at the type level as a list of mappings from type variables to *pairs* of types. We use pairs of types in substitutions because we have to simultaneously compute substitutions from c to a and from c to b.

To be concise, lines 4-7 present a single definition parametrized by the type constructor arity but in practice, there needs to be a different instance declaration for each type constructor arity.

Proposition 1 If $LeastGen'\ a\ b\ c\ \sigma_{\text{in}}\ \sigma_{out}$ holds, then substitution σ_{out} extends σ_{in} and $\sigma_{out}c = (a, b)$.

PROOF. By induction on the type representation of a and b.

A type can either be a type variable, represented as $TVar\ a$, or an n-ary type constructor T applied to n type arguments[4]. The rule to be applied depends on whether the type constructors of a and b are the same or not.

(*i*) If the constructors are the same, then the rule defined in lines 4-7 computes $(T\ c_1 \ldots c_n)$ using the induction hypothesis that $\sigma_{\text{i}}c_{\text{i}} = (a_{\text{i}}, b_{\text{i}})$, for $\text{i} = 1 \ldots n$. The component-wise application of constraints is done from left to right, starting from substitution σ_0 and extending it to the resulting substitution σ_n. The type equality constraint $(T\ c_1 \ldots c_n) \sim c$ checks that c is unifiable with $(T\ c_1 \ldots c_n)$ and, if so, unifies them. Then, we can check that $\sigma_n c = (a, b)$.

(*ii*) If the type constructors are not the same the only possible generalization is a type variable. In the rule defined in lines 10-13 the goal is to extend σ_{in} with the mapping $c \mapsto (a, b)$ such that $\sigma_{out}c = (a, b)$, while preserving the injectivity of the substitution (see next proposition). □

Proposition 2 If σ_{in} is an injective function, and $LeastGen'\ a\ b\ c\ \sigma_{\text{in}}\ \sigma_{out}$ holds, then σ_{out} is an injective function.

PROOF. By construction $LeastGen'$ introduces a binding from a fresh type variable to (a, b), in the rule defined in lines 10-13, only if there is no type variable already mapping to (a, b)—in which case σ_{in} is not modified.

To do this, we first check that c is actually a type variable ($TVar\ c$) by checking its representation using $Analyze$. Then in relation $MapsTo$ we bind c' to the (possibly inexistent) type variable that maps to (a, b) in σ_{in}. In case there is no such mapping, then c' is $None$.

Finally, relation $VarCase$ binds σ_{out} to σ_{in} extended with $\{c \mapsto (a, b)\}$ in case c' is $None$, otherwise $\sigma_{out} = \sigma_{\text{in}}$. It then unifies c with c'. In all cases c is bound to the variable that maps to (a, b) in σ_{out}, because it was either unified in rule $MapsTo$ or in rule $VarCase$. The hypothesis that σ_{in} is injective ensures that any preexisting mapping is unique. □

Proposition 3 If σ_{in} is an injective function, and $LeastGen'\ a\ b\ c\ \sigma_{\text{in}}\ \sigma_{out}$ holds, then c is the least general type of a and b.

[4] We use the $Analyze$ type class from $PolyTypeable$ to get a type representation at the type level. For simplicity we omit the rules for analyzing type representations.

PROOF. By induction on the type representation of a and b.

(*i*) If the type constructors are different the only generalization possible is a type variable c.

(*ii*) If the type constructors are the same, then $a = T\ a_1 \ldots a_n$ and $b = T\ b_1 \ldots b_n$. By Proposition 1, $c = T\ c_1 \ldots c_n$ generalizes a and b with the substitution σ_{out}. By induction hypothesis c_i is the least general type of $(a_\mathrm{i}, b_\mathrm{i})$.

Now consider a type d that also generalizes a and b, *i.e.* $a \preceq \mathrm{d}$ and $b \preceq \mathrm{d}$, with associated substitution α. We prove c is less general than d by constructing a substitution τ such that $\tau\mathrm{d} = c$.

Again, there are two cases, either d is a type variable, in which case we set $\tau = \{\mathrm{d} \mapsto c\}$, or it has the same outermost type constructor, *i.e.* $\mathrm{d} = T\ \mathrm{d}_1 \ldots \mathrm{d}_n$. Thus $a_\mathrm{i} \preceq \mathrm{d}_\mathrm{i}$ and $b_\mathrm{i} \preceq \mathrm{d}_\mathrm{i}$; and because c_i is the least general type of a_i and b_i, there exists a substitution τ_i such that $\tau_\mathrm{i}\mathrm{d}_\mathrm{i} = c_\mathrm{i}$, for $\mathrm{i} = 1 \ldots n$.

Now consider a type variable $x \in \mathrm{d}om(\tau_\mathrm{i}) \cap \mathrm{d}om(\tau_j)$. By definition of α, we know that $\sigma_{out}(\tau_\mathrm{i}(x)) = \alpha(x)$ and $\sigma_{out}(\tau_j(x)) = \alpha(x)$. Because σ_{out} is injective (by Proposition 2), we deduce that $\tau_\mathrm{i}(x) = \tau_j(x)$ so there are no conflicting mappings between τ_i and τ_j, for any i and j. Consequently, we can define $\tau = \bigcup \tau_\mathrm{i}$ and check that $\tau\mathrm{d} = c$. □

Definition 4 ($\mathit{LeastGen}$ type class) To compute the least general type c for a and b, we define:

$$\mathit{LeastGen}\ a\ b\ c \triangleq\ \mathit{LeastGen'}\ a\ b\ c\ \sigma_{\mathit{empty}}\ \sigma_{\mathit{out}}$$

where σ_{empty} is the empty substitution and σ_{out} is the resulting substitution.

Definition 5 ($\mathit{LessGen}$ type class) To establish that type a is less general than type b, we define:

$$\mathit{LessGen}\ a\ b \triangleq\ \mathit{LeastGen}\ a\ b\ b$$

4.2.3 Pointcut Safety

We now establish the safety of pointcuts with relation to join points.

Definition 6 (Pointcut match) We define the relation $matches(pc, jp)$, which holds iff applying pointcut pc to join point jp in the context of a monad m yields a computation $m\ True$.

Definition 7 (Safe user-defined pointcut) Given a join point term jp and type environment Γ, a user-defined pointcut pc is safe if:

$$\begin{array}{l}\Gamma \vdash pc : \mathit{PC}\ m\ a\ b \\ \Gamma \vdash jp : \mathit{JP}\ m\ a'\ b' \\ \Gamma \vdash matches(pc, jp)\end{array}$$

implies that

$$a' \rightarrow m\ b' \preceq a \rightarrow m\ b$$

Now we prove that the matched type of a given pointcut is more general than the join points matched by that pointcut.

Proposition 4 Given a join point term jp and a pointcut term pc, and type environment Γ; and that if pc is user-defined, then it is safe (according to Definition 7). Then,

$$\Gamma \vdash pc : PC\ m\ a\ b$$
$$\Gamma \vdash jp : JP\ m\ a'\ b'$$
$$\Gamma \vdash matches(pc, jp)$$

implies that

$$a' \to m\ b' \preceq a \to m\ b$$

PROOF. By induction on the matched type of the pointcut.

- Case $pcCall$: By construction the matched type of a $pcCall\ f$ pointcut is the type of f. Such a pointcut matches a join point with function g if and only if: f is equal to g, and the type of f is less general than the type of g. (On both $pcCall$ and $pcType$ this type comparison is performed by $compareType$ on the type representations of its arguments.)
- Case $pcType$: By construction the matched type of a $pcType\ \ f$ pointcut is the type of f. Such a pointcut only matches a join point with function g whose type is less general than the matched type.
- Case $pcAnd$ on $PC\ \ PC$: Consider pc_1 ‘$pcAnd$‘ pc_2. The matched type of the combined pointcut is the *principal unifier* of the matched types of the arguments—which represents the intersection of the two sets of join points. The property holds by the induction hypothesis applied to pc_1 and pc_2.
- Case $pcAnd$ on $PC\ \ RequirePC$: Consider pc_1 ‘$pcAnd$‘ pc_2. The matched type of the combined pointcut is the type of pc_1 and it is checked that the type required by pc_2 is *more general* so the application of pc_2 will not yield an error. The property holds by induction hypothesis on pc_1.
- Case $pcOr$: Consider pc_1 ‘$pcOr$‘ pc_2. The matched type of the combined pointcut is the *least general type* of the matched types of the argument, computed by the $LeastGen$ constraint—which represents the union of the two sets of join points. The property holds by induction hypothesis on pc_1 and pc_2.
- Case $pcNot$: The matched type of a pointcut constructed with $pcNot$ is a fresh type variable, which by definition is more general than the type of any join point.

□

4.2.4 Advice Safety

If an aspect is well-typed, then the advised type of the advice is more general than the matched type of the pointcut:

Proposition 5 Given a pointcut term pc, an advice term adv, and a type environment Γ, if

$$\Gamma \vdash pc : PC\ m\ a\ b$$
$$\Gamma \vdash \mathrm{adv} : Advice\ m\ c\ \mathrm{d}$$
$$\Gamma \vdash (aspect\ pc\ \mathrm{adv}) : Aspect\ m\ a\ b\ c\ \mathrm{d}$$

then

$$a \rightarrow m\ b \preceq c \rightarrow m\ \mathrm{d}$$

PROOF. Using the definition of $Aspect$ (Section 4.1.2) and because it holds that $\Gamma \vdash aspect\ pc\ \mathrm{adv} : Aspect\ m\ a\ b\ c\ \mathrm{d}$, we know that the constraint $LessGen$ is satisfied, so by Definitions 4 and 5, and Proposition 1, we can check that $a \rightarrow m\ b \preceq c \rightarrow m\ \mathrm{d}$. □

4.2.5 Safe Aspects

We now show that if an aspect is well-typed, then the advised type of the advice is more general than the type of join points matched by the corresponding pointcut:

Theorem 1 (Safe Aspects) Given the terms jp, pc and adv representing a join point, a pointcut and an advice respectively, given a type environment Γ; and assuming that if pc is a user-defined pointcut, then it is safe (according to Definition 7). Then,

$$\Gamma \vdash pc : PC\ m\ a\ b$$
$$\Gamma \vdash \mathrm{adv} : Advice\ m\ c\ \mathrm{d}$$
$$\Gamma \vdash aspect\ pc\ \mathrm{adv} : Aspect\ m\ a\ b\ c\ \mathrm{d}$$
$$\Gamma \vdash jp : JP\ m\ a'\ b'$$
$$\Gamma \vdash matches(pc, jp)$$

implies that

$$a' \rightarrow m\ b' \preceq c \rightarrow m\ \mathrm{d}$$

PROOF. By Proposition 4 and 5 and the transitivity of $\preceq$. □

Corollary 1 (Safe Weaving) The coercion of the advice in $apply_adv$ is safe.

PROOF. Recall $apply_adv$ (Section 3.3):

$$apply_adv :: Advice\ m\ a\ b \rightarrow t \rightarrow t$$
$$apply_adv\ adv\ f = (unsafeCoerce\ adv)\ f$$

By construction, $apply_adv$ is used only with a function f that comes from a join point that is matched by a pointcut associated to adv. Using Theorem 1, we know that the join point has type

$JP\ m\ a'\ b'$ and that $a' \rightarrow m\ b' \preceq a \rightarrow m\ b$. We note σ the associated substitution. Then, by compatibility of substitutions with the typing judgement (Pierce, 2002), we deduce $\sigma\Gamma \vdash \sigma adv : Advice\ m\ a'\ b'$. Therefore $unsafeCoerce\ adv$ corresponds exactly to σadv, and is safe.

□

In this chapter we have shown that our particular use of the unsafe cast operation in the context of applying advices is always safe. Consequently, in a well-typed program advices "do not go wrong", that is, they are always applied to join points of the proper type. We rely on a partial order on types, namely in the *is less general relation*. After proving that the $LeastGen'$ type class statically computes or verifies this relation for two types, the rest of the proofs are straightforward. An aspect is well-typed iff the matched type of its pointcut is less general than the type of the advice. In turn, a pointcut is safely defined iff the type of the join points accepted by the pointcut is less general than the matched type of the pointcut, encoded using phantom type variables.

Finally, observe that the model described in Chapters 3 and 4 reflects a specific point in the design space of a typed monadic embedding of aspects. The next chapter concludes with the first part of this thesis, discussing a number of issues and design decisions that guided the development of our model.

Chapter 5

Discussion About the Model

We now discuss a number of issues related to our approach regarding quantification and obliviousness, how to deal with overloaded functions, and the technical requirements of our model.

5.1 Quantification

Pointcuts quantify about join points, and a major element of the join point is the function being applied. In existing AOP languages there are many ways by which pointcuts select advised entities: for instance, by name (*e.g.* method names in AspectJ (Kiczales et al., 2001), function names in AspectML (Dantas et al., 2008)), by reference equality (*e.g.* AspectScheme (Dutchyn et al., 2006), AspectScript (Toledo et al., 2010)), by their type (*e.g.* AspectJ), using a mechanism to explicitly attach tags or types to join points (*e.g.* Ptolemy (Rajan and Leavens, 2008), JPIs (**?**)), etc.

In our presentation we have developed the $pcType$ and $pcCall$ pointcuts as a means to quantify based on the type and identity of functions. The $pcType$ designator relies on type comparison, implemented using the $PolyTypeable$ type class in order to obtain representations for polymorphic types. The $pcCall$ is more problematic, as it relies on function equality, but Haskell does not provide an operator like $eq?$ in Scheme. Indeed, the general problem of determining function equality is undecidable, thus an approximation is required. We now discuss two approaches to implement this pointcut designator.

5.1.1 Approximating Equality on Functions

A first workaround is to implement a pointer comparison operator like $eq?$ in Scheme, to define an approximate notion of function equality.

StableNames The $StableNames$ pointer-comparison API is provided by default in the GHC compiler; unfortunately it is fragile. $StableNames$ equality is safe in the sense that it does not equate

two functions that are not the same, but two functions that are equal can be seen as different. The problem becomes even more systematic when it comes to bounded polymorphism. Indeed, each time a function with constraints is used, a new closure is created by passing the current method dictionary of type class instances. Even with optimized compilation (*e.g.* ghc -O), this (duplicated) closure creation is unavoidable and so $\mathit{StableNames}$ will consider different any two constrained functions, even if the passed dictionary is the same.

Tagged Functions Another approach, which is basically a manually-implemented version of $\mathit{StableNames}$, is to define a special class of tagged functions whose equality is defined as the equality of their tags:

$$
\begin{array}{l}
\mathbf{type}\ \mathit{FunctionTag} = \ldots \\
\mathbf{data}\ \mathit{Function}\ a\ b = \mathit{Function}\ (a \to b)\ \mathit{FunctionTag} \\
\mathbf{instance}\ \mathit{Eq}\ (\mathit{Function}\ a\ b)\ \mathbf{where} \\
\quad \mathit{Function}\ _\ t_1 \equiv \mathit{Function}\ _\ t_2 = t_1 \equiv t_2
\end{array}
$$

This approach allows us to have a somewhat robust notion of function equality. However such a solution is cumbersome and hardly scalable, because it would force us to duplicate all operations on functions (*e.g.* $\mathit{compose}$, map, etc.). An additional problem of this solution is the need to properly tag functions, using either a preprocessor or some unique supply monad.

5.1.2 Tagged Function Applications

A different approach, similar to that of Ptolemy or JPIs, is to attach tags to *function applications* rather than to functions themselves (which implies modifying the join point model accordingly). In this approach the $\#$ operator takes a tag as an additional argument and emits a tagged join point. For example $f\ \#^t\ x$ applies function f with tag t to argument x. Instead of defining a pcCall pointcut based on function identity, we can define a pcTag pointcut as follows:

$$
\begin{array}{l}
\mathit{pcTag} :: \mathit{Monad}\ m \Rightarrow \mathit{FunctionTag} \to \mathit{PC}\ m\ a'\ b' \\
\mathit{pcTag}\ t = \mathit{PC}\ \$\ \mathit{return}\ \$\ \lambda \mathit{jp} \to \mathit{return}\ (\mathit{getJpTag}\ \mathit{jp} \equiv t)
\end{array}
$$

Observe that in contrast to pcCall, the matched type of pcTag consists of fresh type variables, just like pcNot. This is because tagged applications can now represent a family of functions to be advised. To be practical, the matched type must be refined using pcAnd. Crucially, this approach maintains the safety of aspects because functions with the right tag but an incompatible type will not be advised. Tagged applications are orthogonal to the implementation of pcType, which may or may not take them into account.

The model of explicitly tagged applications appears to be the most applicable in practice, and it is also amenable to formal reasoning, as we will describe later in Part III. However, the original development of EffectiveAspects (Figueroa et al., 2014; Tabareau et al., 2013), presented in this first part of the thesis, considered only the approaches of Section 5.1.1.

5.2 Aspects and Bounded Polymorphism

From a programmer's point of view, it can be interesting to advise an overloaded function (that is, the application of all the possible implementations) with a single aspect. However, deploying aspects in the general case of bounded polymorphism is problematic because of the resolution of class constraints. For example, consider the Log type class as follows:

$$\begin{array}{l}\textbf{class}\ Show\ a \Rightarrow Log\ a\ \textbf{where}\\ \quad log :: Monad\ m \Rightarrow a \rightarrow m\ String\end{array}$$

Now it may be desirable to advise all applications of log, for instance to define a custom text format:

$$\begin{array}{l}formatLogAdv :: (Log\ a, Monad\ m) \Rightarrow Advice\ m\ a\ String\\ formatLogAdv\ proceed\ arg = \textbf{do}\ s \leftarrow proceed\ arg\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad return\ \$\ "\ [Log : " +\!\!+\ s +\!\!+ "]\ "\\ program\ n = \textbf{do}\ deploy\ (aspect\ (pcCall\ log)\ formatLogAdv)\\ \qquad\qquad\qquad log\ \#\ n\end{array}$$

But $program$ fails to compile. The particular error given by GHC 7.4 is:

```
No instance for (Show c1) arising from a use of 'log'
    The type variable 'c1' is ambiguous
    Possible fix: add a type signature that fixes
    these type variable(s)
    Note: there are several potential instances:
      instance Show Double -- Defined in 'GHC.Float'
      instance Show Float -- Defined in 'GHC.Float'
      instance (Integral a, Show a) => Show (GHC.Real.Ratio a)
        -- Defined in 'GHC.Real'
      ...plus 45 others
```

Recall that in order to be able to type the aspect environment, we existentially hide the matched and advised types of an aspect (Section 3.2). This means that all type class constraints must be solved statically at the point an aspect is deployed. If the matched and advised types are both bounded polymorphic types, type inference cannot gather enough information to statically solve the constraints. So advising all possible implementations requires repeating deployment of the same aspect with different type annotations, one for each instance of the involved type classes. In our example it means having to deploy for each instance of Log (for some unspecified monad m):

$$\begin{array}{l}program\ n = \textbf{do}\ deploy\ (aspect\ (pcCall\ (log :: Integer \rightarrow m\ String)\ formatLogAdv)\\ \qquad\qquad\qquad deploy\ (aspect\ (pcCall\ (log :: Float \rightarrow m\ String)\quad formatLogAdv)\\ \qquad\qquad\qquad \ldots\\ \qquad\qquad\qquad log\ \#\ n\end{array}$$

To alleviate this problem, we developed a macro using TemplateHaskell (Sheard and Jones, 2002). Thus the resulting program can be written as:

$$
\begin{aligned}
&\textit{program}\ n = \textbf{do}\ \$\ (\textit{deployOnMethod}\ \texttt{"Log"}\ \texttt{"log"}\ [|\ \textit{formatLogAdv}\ |])\\
&\qquad\qquad\qquad \textit{log}\ \#\ n
\end{aligned}
$$

which will expand into the annotated deployment for each instance, described above. The macro requires the name of the type class and the method, as well as the advice.

The macro extracts all the constrained variables in the matched type of the pointcut ($Log\ a$ in the example), and generates an annotated deployment for every possible combination of instances that satisfy all constraints. In order to retain safety, the advised type of an aspect must be less constrained than its matched type. This is statically enforced by the Haskell type system after macro expansion.

5.3 Obliviousness

The embedding of aspects we have presented thus far supports quantification through pointcuts, but is not oblivious: open applications are explicit in the code. A first way to introduce more obliviousness without requiring non-local macros or, equivalently, a preprocessor or ad hoc runtime semantics, is to use *partial applications* of $\#$. For instance, the $enqueue$ function can be turned into an implicitly woven function by defining $enqueue' = enqueue\ \#$. It can be sufficient in similar scenarios where quantification is under control. Otherwise, it can yield issues in the definition of pointcuts that rely on function identity, because $enqueue'$ and $enqueue$ are different functions. Also, this approach is not entirely satisfactory with respect to obliviousness because it has to be applied specifically for each function.

De Meuter (1997) proposes to use the binder of a monad to redefine function application. His approach focuses on defining one monad per aspect, but can be generalized to a list of dynamically-deployed aspects as presented in Section 3.2. For this, we can redefine the monad transformer $\mathbb{A}_{\mathrm{T}}$ to make all monadic applications open transparently:

$$
\begin{aligned}
&\textbf{instance}\ \textit{Monad}\ m \Rightarrow \textit{Monad}\ (\mathbb{A}_{\mathrm{T}}\ m)\ \textbf{where}\\
&\quad \textit{return}\ a = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda \textit{aenv} \rightarrow \textit{return}\ (a, \textit{aenv})\\
&\quad k \gg\!= f \quad = \textbf{do}\ x \leftarrow k\\
&\qquad\qquad\qquad\quad f\ \#\ x
\end{aligned}
$$

This presentation improves obliviousness because any monadic application is now an open application, but it suffers from a major drawback: it breaks the *monadic laws*. Indeed, left identity and associativity

-- Left identity:
$return\ x \gg\!= f = f\ x$
-- Associativity:
$(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$

can be invalidated, depending on the current list of deployed aspects. This is not surprising as AOP allows one to redefine the behavior of a function and even to redefine the behavior of a function

depending on its context of execution. Breaking monadic laws is not prohibited by Haskell, but it is very dangerous and fragile; for instance, some compilers exploit the laws to perform optimizations, so breaking them can yield incorrect optimizations.

5.4 Technical Requirements of our Model

The current implementation of Effective Aspect uses several extensions of the GHC Haskell compiler (see the details at http://plead.cl/EffectiveAspects). Nevertheless, we believe that the anti-unification algorithm at the type level (Section 4.1.1) is the essential feature that would be required to make our approach work on other languages. A potential line of work is to port EffectiveAspects to Scala, which has some likeness to Haskell and also has monads, and investigate what kind of issues arise in the process.

We have presented a typed and sound monadic embedding of aspects where pointcuts, advices, and aspects are first-class. Among several issues, a crucial point in the design of our model is the need for a decidable notion of function equality. Additionally, it is not clear how to combine aspects and bounded polymorphism in a way other than deploying one aspect for every combination of instances of the involved type classes. Nevertheless, our model is open to further improvements.

The next chapter concludes the first part of the thesis by discussing work related to functional aspect languages as well as the relation between aspects and monads. Then, the second part of this work builds upon the model here presented to show how, by virtue of using monads, we can reuse and extend several techniques to control the interaction between aspects and computational effects.

Chapter 6

Related Work, Part I

For convenience to the reader, each part of this thesis presents a separate discussion of related work. In this chapter we discuss the connection between monads and AOP, as well as the relation between EffectiveAspects and other functional aspect languages.

Connection between monads and AOP The earliest connection between aspects and monads was established by De Meuter in 1997 (De Meuter, 1997). In that work, he proposes to describe the weaving of a given aspect directly in the binder of a monad. As we have just described above (Section 5.3), doing so breaks the monad laws, and is therefore undesirable.

Wand *et al.* (Wand et al., 2004) formalize pointcuts and advice and use monads to structure the denotational semantics; a monad is used to pass the join point stack and the store around evaluation steps. The specific flavor of AOP that is described is similar to AspectJ, but with only pure pointcuts. The calculus is untyped.

Hofer and Osterman (Hofer and Ostermann, 2007) shed some light on the modularity benefits of monads and aspects, clarifying that they are different mechanisms with quite different features: monads do not support declarative quantification, and aspects do not provide any support for encapsulating computational effects. In this regard, our work does not attempt at unifying monads and aspects, contrary to what De Meuter suggested. Instead, we exploit monads in Haskell to build a flexible embedding of aspects that can be modularly extended. In addition, the fully-typed setting provides the basis for reasoning about monadic effects.

Monadic weaving The notion of *monadic weaving* was described by Tabareau (2012), where he shows that writing the aspect weaver in a monadic style paves the way for modular language extensions. He illustrated the extensibility approach with execution levels (Tanter, 2010) and level-aware exception handling (Figueroa and Tanter, 2011). In Chapter 9 we describe how this extensibility can be exploited in the context of our model. The authors then worked on a practical monadic aspect weaver in Typed Racket (Figueroa et al., 2012). However, the type system of Typed Racket turned out to be insufficiently expressive, and the top type Any had to be used to describe pointcuts and advices. This was the original motivation to study monadic weaving in Haskell. Also in

contrast to this work, prior work on monadic aspect weaving does not consider a base language with monads. In our model, both the base language and the aspect weaver are monadic, combining the benefits of type-based reasoning about effects and modular language extensions (described in Part II and Part III)—including type-based reasoning about language extensions.

AOP in Haskell via type classes Haskell has already been the subject of AOP investigations using the type class system as a way to perform static weaving (Sulzmann and Wang, 2007). AOP idioms are translated to type class instances, and type class resolution is used to perform static weaving. This work only supports simple pointcuts, pure aspects and static weaving, and is furthermore very opaque to modular changes as the translation of AOP idioms is done internally at compile time.

Comparison with dynamically-typed aspect languages The specific flavor of pointcut/advice AOP that we have developed is directly inspired by AspectScheme (Dutchyn et al., 2006) and AspectScript (Toledo et al., 2010): dynamic aspect deployment, first-class aspects, and extensible set of pointcut designators. While we have not yet developed the more advanced scoping mechanisms found in these languages (Tanter, 2008), we believe there are no specific challenges in this regard. The key difference here is that these languages are both dynamically typed, while we have managed to reconcile this high level of flexibility with static typing.

Comparison with statically-typed aspect languages In terms of statically-typed functional aspect languages, the closest proposal to ours is AspectML (Dantas et al., 2008). In AspectML, pointcuts are first-class, but advice is not. The set of pointcut designators is fixed, as in AspectJ. AspectML does not support: advising anonymous functions, aspects of aspects, separate aspect deployment, and undeployment.

AspectML was the first language in which first-class pointcuts were statically typed. The typing rules rely on anti-unification, just like we do in this paper. The major difference, though, is that AspectML is defined as a completely new language, with a specific type system and a specific core calculus. Proving type soundness is therefore very involved (Dantas et al., 2008). In contrast, we do not need to define a new type system and a new core calculus. Type soundness in our approach is derived straightforwardly from the type class that establishes the anti-unification relation. Half of Section 4.2 is dedicated to proving that this type class is correct. Once this is done (and it is a result that is independent from AOP), proving aspect safety is direct. Another way to see this work is as a new illustration of the expressive power of the type system of Haskell, in particular how phantom types and type classes can be used in concert to statically type embedded languages.

Aspectual Caml (Masuhara et al., 2005) is another polymorphic aspect language. Interestingly, Aspectual Caml uses type information to influence matching, rather than for reporting type errors. More precisely, the type of pointcuts is inferred from the associated advices, and pointcuts only match join points that are valid according to these inferred types. We believe this approach can be difficult for programmers to understand, because it combines the complexities of quantification with those of type inference. Aspectual Caml is implemented by modifying the Objective Caml compiler, including modifications to the type inference mechanism. There is no proof of type

soundness.

The advantages of our typed embedding do not only lie within the simplicity of the soundness proof. They can also be observed at the level of the implementation. The AspectML implementation is over 15,000 lines of ML code (Dantas et al., 2008), and the Aspectual Caml implementation is around 24,000 lines of Objective Caml code (Masuhara et al., 2005). In contrast, our implementation, including all extensions from Part II, is only around 1,600 lines of Haskell code. Also, embedding an AOP extension entirely inside a mainstream language has a number of practical advantages, especially when it comes to efficiency and maintainability of the extension.

Part II

Controlling Effects

Chapter 7

Open and Protected Modules, with Effects

In the previous chapters we have shown the design, implementation, and type safety of the EffectiveAspects model. In this chapter we now turn our attention to illustrating how we can exploit the monadic embedding of aspects to encode Open Modules (Aldrich, 2005) extended with effects. Additionally we present the notion of *protected pointcuts*, which are pointcuts whose type places restrictions on admissible advice. We illustrate the use of protected pointcuts to enforce control flow properties of external advice, reusing the approach of EffectiveAdvice (Oliveira et al., 2010).

7.1 Background: Open Modules

Before we start with our development, we give a small background on Open Modules. Aldrich (2005) proposed Open Modules as a module system that intends to be *open* to extension with advice, while still allowing *modular* reasoning with respect to the implementation details of modules. In particular, this means that function calls that are internal to the implementation of a module cannot be advised from outside. That is, external advice can only intercept calls to functions in the public interface of a module as well as pointcuts explicitly exported by a module. Conversely, only internal advice can affect internal calls between functions.

Open Modules is formalized in a functional aspect language called `TinyAspect`, which features "second-class" advices and aspects. There is only one pointcut designator that matches calls to a function name. The name must be specified either in the public or private interface of a module. The language features a type system to enable modular reasoning between module boundaries. The key contribution of Open Modules is the idea that a module must be able to restrict aspect quantification by controlling the join points in which it can be advised.

From our point of view, the principal drawback of Open Modules is that computational effects are not explicitly addressed in its core calculus, despite being used in the code of impure advice. Therefore it is not clear to what extent the notion of Open Modules is compatible with modular reasoning in the presence of computational effects.

$$
\begin{array}{l}
\textbf{module}\ \mathit{Fib}\ (\mathit{fib}, \mathit{pcFib})\ \textbf{where} \\
\textbf{import}\ \mathit{AOP} \\
\mathit{pcFib} = \mathit{pcCall}\ \mathit{fibBase}\ \text{`}\mathit{pcAnd}\text{`}\ \mathit{pcArgGT}\ 2 \\
\mathit{fibBase}\ n = \mathit{return}\ 1 \\
\mathit{fibAdv}\ \mathit{proceed}\ n = \textbf{do}\ f_1 \leftarrow \mathit{fibBase}\ \#\ (n-1) \\
\qquad\qquad\qquad\qquad\quad f_2 \leftarrow \mathit{fibBase}\ \#\ (n-2) \\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ (f_1 + f_2) \\
\mathit{fib} :: \mathit{Monad}\ m \Rightarrow m\ (\mathit{Int} \rightarrow m\ \mathit{Int}) \\
\mathit{fib} = \textbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ \mathit{fibAdv}) \\
\qquad\quad \mathit{return}\ \$\ \mathit{fibBase}\ \ \#
\end{array}
$$

Figure 7.1: Fibonacci module.

7.2 A Simple Example

We first describe a simple example that serves as the starting point. Figure 7.1 describes a Fibonacci module, following the canonical example of Open Modules. The module uses an internal aspect to implement the recursive definition of Fibonacci: the base function, $\mathit{fibBase}$, simply implements the base case; and the fibAdv advice implements recursion when the pointcut pcFib matches. Note that pcFib uses the user-defined pointcut $\mathit{pcArgGT}$ (defined in Section 4.1.1) to check that the call to $\mathit{fibBase}$ is done with an argument greater than 2. The fib function is defined by first deploying the internal aspect, and then partially applying $\#$ to $\mathit{fibBase}$. This transparently ensures that an application of fib is open. The fib function is exported, together with the pcFib pointcut, which can be used by an external module to advise applications of the internal $\mathit{fibBase}$ function. Figure 7.2 presents a Haskell module that provides a more efficient implementation of fib by using a memoization advice. To benefit from memoization, a client only has to import fib from the $\mathit{MemoizedFib}$ module instead of directly from the Fib module.

Note that if we consider that the aspect language only supports the pcCall pointcut designator, this implementation actually represents an open module proper. Preserving the properties of open modules, in particular protecting from external advising of internal functions, in presence of arbitrary quantification (*e.g.* pcType, or an always-matching pointcut) is left for future work. Importantly, just like Open Modules, the approach described here does not ensure anything about the advice beyond type safety. In particular, it is possible to create an aspect that incorrectly calls $\mathit{proceed}$ several times, or an aspect that has undesired computational effects. Fortunately, the type system can assist us in expressing and enforcing specific interference properties.

7.3 Protected Pointcuts

In order to extend Open Modules with effect-related enforcement, we introduce the notion of *protected pointcuts*, which are pointcuts enriched with restrictions on the effects that associated advice

```
module MemoizedFib (fib) where
import qualified Fib
import AOP

memo proceed n =
  do table ← get
     if member n table
       then return (table ! n)
       else do y ← proceed n
               table' ← get
               put (insert n y table')
               return y

fib = do deploy (aspect Fib.pcFib memo)
         Fib.fib
```

Figure 7.2: Memoized Fibonacci module.

can exhibit. Simply put, a protected pointcut embeds a *combinator* that is applied to the advice in order to build an aspect. If the advice does not respect the (type) restrictions expressed by the combinator, the aspect creation expression simply does not type check and hence the aspect cannot be built. A combinator is any function that can produce an advice:

$$\textbf{type}\ Combinator\ t\ m\ a\ b = Monad\ m \Rightarrow t \rightarrow Advice\ m\ a\ b$$

The $protectPC$ function packs together a pointcut and a combinator:

$$protectPC :: (Monad\ m, LessGen\ (a \rightarrow m\ b)\ (c \rightarrow m\ d)) \Rightarrow$$
$$PC\ m\ a\ b \rightarrow Combinator\ t\ m\ c\ d \rightarrow ProtectedPC\ m\ a\ b\ t\ c\ d$$

A protected pointcut, of type $ProtectedPC$, cannot be used with the standard aspect creation function $aspect$. The following $pAspect$ function is the only way to get an aspect from a protected pointcut (the constructor PPC is not exposed):

$$pAspect :: Monad\ m \Rightarrow ProtectedPC\ m\ a\ b\ t\ c\ d \rightarrow t \rightarrow Aspect\ m\ a\ b\ c\ d$$
$$pAspect\ (PPC\ pc\ comb)\ adv = aspect\ pc\ (comb\ adv)$$

The key point here is that when building an aspect using a protected pointcut, the combinator $comb$ is applied to the advice adv. We now show how to exploit this extension of Open Modules to restrict control flow properties, using the proper type combinators. The next chapter describes how to control computational effects.

7.4 Enforcing Control Flow Properties

Rinard *et al.* present a classification of advice in four categories depending on how they affect the control flow of programs (Rinard et al., 2004):

```
type Replace m a b = (a → m b)
replace :: Replace m a b → Advice m a b
replace radv proceed = radv

type Augment a b c m = (a → m c, a → b → c → m ())
augment :: Monad m ⇒ Augment a b c m → Advice m a b
augment (before, after) proceed arg =
  do c ← before arg
     b ← proceed arg
     after arg b c
     return b

type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
narrow :: Monad m ⇒ Narrow m a b c → Advice m a b
narrow (p, aug, rep) proceed x =
  do b ← p x
     if b then replace rep proceed x
          else augment aug proceed x
```

Figure 7.3: Replacement, augmentation and narrowing advice combinators (adapted from (Oliveira et al., 2010)).

- **Combination:** The advice can call *proceed* any number of times.
- **Replacement:** There are no calls to *proceed* in the advice.
- **Augmentation:** The advice calls *proceed* exactly once, and it does not modify the arguments to or the return value of *proceed*.
- **Narrowing:** The advice calls *proceed* at most once, and does not modify the arguments to or the return value of *proceed*.

In EffectiveAdvice (Oliveira et al., 2010), Oliveira and colleagues show a type-based enforcement of these categories, through advice combinators (Figure 7.3). These combinators fit the general *Combinator* type we described in Section 7.3, and can therefore be embedded in protected pointcuts. Observe that no combinator is needed for combination advice, because no interference properties are enforced. Replacement advice is advice that has no access to *proceed*. Augmentation advice is represented by a pair of *before*/*after* advice functions, such that *after* has access to the argument, the return value, and an extra value optionally exposed by the *before* function. A narrowing advice is in fact the combination of both a replacement advice and an augmentation advice, where the choice between both is driven by a runtime predicate.

As an illustration, observe that memoization is a typical example of a narrowing advice: the combination of a replacement advice ("return memoized value without proceeding") and an augmentation advice ("proceed and memoize return value"), where the choice between both is driven by a runtime predicate ("is there a memoized value for this argument?"). Therefore it is now straightforward for the *Fib* module to expose a protected pointcut that restricts valid advice to narrowing advice only:

$$
\begin{aligned}
&\mathit{memo} :: (\mathbb{S}_{\mathbf{M}}\ (\mathit{Map}\ a\ b)\ m, \mathit{Ord}\ a) \Rightarrow \mathit{Narrow}\ a\ b\ ()\ m \\
&\mathit{memo} = (\mathit{pred}, (\mathit{before}, \mathit{after}), \mathit{rep})\ \mathbf{where} \\
&\quad \mathit{pred}\ n \qquad = \mathbf{do}\ \{\mathit{table} \leftarrow \mathit{get}; \mathit{return}\ (\mathit{member}\ n\ \mathit{table})\} \\
&\quad \mathit{before}\ _ \quad = \mathit{return}\ () \\
&\quad \mathit{after}\ n\ r\ _ = \mathbf{do}\ \{\mathit{table} \leftarrow \mathit{get}; \mathit{put}\ (\mathit{insert}\ n\ r\ \mathit{table})\} \\
&\quad \mathit{rep}\ x \qquad = \mathbf{do}\ \{\mathit{table} \leftarrow \mathit{get}; \mathit{return}\ (\mathit{table}\ !\ n)\}
\end{aligned}
$$

Figure 7.4: Memoization as a narrowing advice (adapted from (Oliveira et al., 2010)).

$$
\begin{aligned}
&\mathbf{module}\ \mathit{Fib}\ (\mathit{fib}, \mathit{ppcFib})\ \mathbf{where} \\
&\mathit{ppcFib} = \mathit{protectPC}\ \mathit{pcFib}\ \mathit{narrow} \\
&\ldots
\end{aligned}
$$

The protected pointcut ppcFib embeds the narrow type combinator. Hence, only advice that can be statically typed as narrowing advice can be bound to that pointcut. A valid definition of the memo advice is given in Figure 7.4. Note that the protected pointcut is only restrictive with respect to the control flow effect of the advice, but not with respect to its computational effect: any monad m is accepted.

Finally, note that this approach is not limited to the four categories of Rinard *et al.*; custom kinds of advice can be defined in a similar way. For instance, we consider *adaptation* advice as a weaker version of narrowing where the advice is allowed to modify the arguments to *proceed*. The implementation is straightforward:

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Adaptation}\ a\ b\ c\ m = (a \to a, a \to m\ c, a \to b \to c \to m\ ()) \\
&\mathit{adapt} :: \mathit{Adaptation}\ a\ b\ c\ m \to \mathit{Advice}\ m\ a\ b \\
&\mathit{adapt}\ (\mathit{adapter}, \mathit{before}, \mathit{after})\ \mathit{proceed}\ \mathit{arg} = \\
&\quad \mathit{augment}\ (\mathit{before}, \mathit{after})\ \mathit{proceed}\ (\mathit{adapter}\ \mathit{arg})
\end{aligned}
$$

A relevant design choice is whether the $\mathit{adapter}$ function is pure or is allowed to perform effects. This choice affects which properties can be statically checked based on the type of the advice. Allowing effects is more expressive, but it is source of potential interferences, in addition to advices and pointcuts.

In this chapter we have shown how we can directly combine the principles of Open Modules with the control flow combinators of EffectiveAdvice. The combination is fairly straightforward: first, because pointcuts are first-class they can directly be exported in the public interface of a module. Second, by using monads we get to reuse, without modification, the EffectiveAdvice combinators. In the next chapter we further exploit the techniques of EffectiveAdvice in order to control the interference between computational effects intended only for aspects, the base system, or both.

Chapter 8

Controlling Effect Interference

The monadic embedding of aspects also enables reasoning about computational effects. We are particularly interested in reasoning about *effect interference* between components of a system: aspects, base programs, and combinations thereof. To do this, in Section 8.1 we first show how to adapt the non-interference types defined in EffectiveAdvice (Oliveira et al., 2010), which distinguish between aspect and base computation. The essence of this technique is to use parametricity to forbid components from making assumptions about some part of the monad stack. Then, because components must work uniformly over the restricted section of the stack, they can only utilize effects available in the non-restricted section.

However this approach falls short when considering several aspects in a system, because aspects (and base programs) can still interfere between them. In Section 8.2 we show how a refinement of the technique can be used to address this situation, but that unfortunately is impractical because it requires explicit liftings and strongly couples components to particular shapes of the monad stack—hampering modularity and reusability.

Finally, we show in Section 8.4 a different approach to enforce non-interference based on *monad views* (Schrijvers and Oliveira, 2011), a recently developed mechanism for handling the monad stack, which is summarized in Section 8.3.

8.1 Distinguishing Aspect and Base Computation

To illustrate the usefulness of distinguishing between aspect and base computation, consider a Fibonacci module where the internal calls throw an exception when given a negative integer as argument. In that situation, it is interesting to ensure that the external advice bound to the exposed pointcut cannot throw or catch those exceptions.

Following EffectiveAdvice (Oliveira et al., 2010), we can enforce an advice to be parametric with respect to a monad used by base computation, effectively splitting the monad stack into two. To this end we define the $\mathbb{NIA}_{\mathrm{T}}$ ($\mathbb{NI}$ stands for non-interference) type:

$$\mathbf{newtype}\ \mathbb{NIA}_{\mathrm{T}}\ t\ m\ a = \mathbb{NIA}_{\mathrm{T}}\ (\mathbb{S}_{\mathrm{T}}\ (\mathit{AspectEnv}\ (\mathbb{NIA}_{\mathrm{T}}\ t\ m))\ (t\ m)\ a)$$

Observe that $\mathbb{NIA}_{\mathrm{T}}$ splits the monad stack into an upper part t, with the effects available to aspects; and a lower part m, with the effects available to base computation. We extend other definitions (weave, deploy, etc.) accordingly.

Note that $\mathbb{NIA}_{\mathrm{T}}$ is a proper monad, but not a monad transformer. This is because the $\mathit{MonadTrans}$ class is designed for a type constructor t that is applied to some monad m, but $\mathbb{NIA}_{\mathrm{T}}$ takes two types as arguments. We could define the partial application $\mathbb{NIA}_{\mathrm{T}}\ t$ as a monad transformer, but this is inconvenient because explicit lift operations would skip the upper layer of the stack[1]. However, for allowing explicit lifting into $\mathbb{NIA}_{\mathrm{T}}$ we need an operation to transform a computation from $t\ m$ into an $\mathbb{NIA}_{\mathrm{T}}\ t\ m$ computation. To this end we provide the niLift operation as follows:

$$\begin{aligned}
&\mathit{niLift} :: \mathit{Monad}\ (t\ m) \Rightarrow t\ m\ a \rightarrow \mathbb{NIA}_{\mathrm{T}}\ t\ m\ a\\
&\mathit{niLift}\ ma = \mathbb{NIA}_{\mathrm{T}}\ \$\ \mathbb{S}_{\mathrm{T}}\ \$\ \lambda aenv \rightarrow \mathbf{do}\\
&\qquad\qquad a \leftarrow ma\\
&\qquad\qquad \mathit{return}\ (a, aenv)
\end{aligned}$$

Effect Interference and Pointcuts The novelty compared to EffectiveAdvice is that we also have to deal with interferences for pointcuts. But to allow effect-based reasoning on pointcuts, we need to distinguish between the monad used by the base computation and the monad used by pointcuts. Indeed, in the interpretation of the type $PC\ m\ a\ b$, m stands for both monads, which forbids to reason separately about them. To address this issue, we need to interpret $PC\ m\ a\ b$ differently, by saying that the matched type is $a \rightarrow b$ instead of $a \rightarrow m\ b$. In this way, the monad for the base computation (which is implicitly bound by b) does not have to be m at the time the pointcut is defined. To accommodate this new interpretation with the rest of the code, very little changes have to be made[2]. Mainly, the types of pcCall, pcType and the definition of Aspect:

$$\begin{aligned}
&\mathit{pcCall}, \mathit{pcType} :: \mathit{Monad}\ m \Rightarrow (a \rightarrow b) \rightarrow PC\ m\ a\ b\\
&\mathbf{data}\ \mathit{Aspect}\ m\ a\ b\ c\ d = (\mathit{Monad}\ m, \mathit{LessGen}\ (a \rightarrow b)\ (c \rightarrow m\ d)) \Rightarrow\\
&\qquad\qquad\qquad\qquad \mathit{Aspect}\ (PC\ m\ a\ b)\ (\mathit{Advice}\ m\ c\ d)
\end{aligned}$$

Note how the definition of Aspect forces the monad of the pointcut computation to be unified with that of the advice, and with that of the base code. The results of Section 4.2 can straightforwardly be rephrased with these new definitions.

Typing Non-Interfering Pointcuts and Advices Using rank-2 types (Peyton Jones et al., 2007) we can restrict the type of pointcuts and advices. The following types synonyms guarantee that non-interfering pointcuts (NIPC) and advices ($\mathit{NIAdvice}$) only use effects available in t.

$$\mathbf{type}\ \mathit{NIPC}\ t\ a\ b = \forall m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t) \Rightarrow PC\ (\mathbb{NIA}_{\mathrm{T}}\ t\ m)\ a\ b$$

[1] Because we would lift from m to $(\mathbb{NIA}_{\mathrm{T}}\ t)\ m$

[2] The implementation available online uses this interpretation of $PC\ m\ a\ b$.

$$
\begin{array}{l}
\textbf{module } \mathit{FibErr}\ (\mathit{fib}, \mathit{ppcFib})\ \textbf{where} \\
\textbf{import } \mathit{AOP} \\
\mathit{pcFib} = \mathit{pcCall}\ \mathit{fibBase}\ `\mathit{pcAnd}`\ \mathit{pcArgGT}\ 2 \\
\mathit{ppcFib} = \mathit{protectPC}\ \mathit{pcFib}\ \mathit{niAdvice} \\
\mathit{fibBase}\ n = \mathit{return}\ 1 \\
\mathit{fibAdv}\ \mathit{proceed}\ n = \textbf{do}\ f_1 \leftarrow \mathit{errorFib}\ \#\ (n-1) \\
\qquad\qquad\qquad\qquad\quad f_2 \leftarrow \mathit{errorFib}\ \#\ (n-2) \\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ (f_1 + f_2) \\
\mathit{fib} = \textbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ \mathit{fibAdv}) \\
\qquad\quad \mathit{return}\ \mathit{errorFib} \\
\mathit{errorFib} :: (\mathit{MonadTrans}\ t, \mathbb{E}_{\mathrm{M}}\ \mathit{String}\ m) \Rightarrow \mathit{Int} \to \mathbb{NIA}_{\mathrm{T}}\ t\ m\ \mathit{Int} \\
\mathit{errorFib}\ n = \textbf{if}\ n < 0 \\
\qquad\qquad\quad \textbf{then}\ \ (\mathit{niLift} \circ \mathit{lift} \circ \mathit{throwError})\text{``}\mathit{Error} : \mathit{negative\ argument}\text{''} \\
\qquad\qquad\quad \textbf{else}\ \ \mathit{fibBase}\ \#\ n
\end{array}
$$

Figure 8.1: Fibonacci with error.

$$\textbf{type } \mathit{NIAdvice}\ t\ a\ b = \forall m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t) \Rightarrow \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}}\ t\ m)\ a\ b$$

By universally quantifying over the type m of the effects used in the base computation, these types enforce, through the properties of parametricity, that pointcuts or advices cannot refer to specific effects in the base program. We can define aspect construction functions that enforce different (non-)interference patterns, such as non-interfering pointcut NIPC with unrestricted advice Advice, unrestricted pointcut PC with non-interfering advice $\mathit{NIAdvice}$, etc.

Enforcing Non-Interference Coming back to Open Modules and protected pointcuts, to enforce non-interfering advice we need to define a typed combinator that requires an advice of type $\mathit{NIAdvice}$:

$$
\begin{array}{l}
\mathit{niAdvice} :: (\mathit{Monad}\ (t\ m), \mathit{Monad}\ m) \Rightarrow \mathit{NIAdvice}\ t\ a\ b \to \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}}\ t\ m)\ a\ b \\
\mathit{niAdvice}\ \mathit{adv} = \mathit{adv}
\end{array}
$$

Observe that the $\mathit{niAdvice}$ combinator is computationally the identity function, but it does impose a type requirement on its argument. Using this combinator, a module can expose a protected pointcut that enforces non-interference with base effects.

Fibonacci Module with Error Handling We now define a Fibonacci module (Figure 8.1) where base functions $\mathit{fibBase}$ and fibAdv raise an exception when given a negative argument.[3] The exception is raised on monad m that corresponds to base computation, and which is required to be an

[3] We do not use an error-checking aspect on purpose, for the sake of illustration. We use such an aspect in Section 8.2 where we consider the issues of multiple effectful aspects.

instance of $\mathbb{E}_{\mathrm{M}}$. The definition of ppcFib enforces that external advice cannot manipulate exceptions in m, because it uses the $\mathit{niAdvice}$ advice combinator. The drawback is that because we are using an effect in an inner layer of the stack, we need to use explicit lifting to satisfy the expected type.

Non-Interfering Base Computation Symmetrically, we can check that a part of the base code cannot interfere with effects available to aspects by using the type synonym NIBase, which universally quantifies over the type t of effects available to the advice:

$$\mathbf{type}\ \mathit{NIBase}\ m\ a\ b = \forall t.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t) \Rightarrow a \rightarrow \mathbb{NIA}_{\mathrm{T}}\ t\ m\ b$$

Reasoning About Pointcut Interference Another use of effect reasoning can be done at the level of pointcuts. Indeed, in the monadic embedding of aspects, we allow for effectful pointcuts. For example, we can define a sequential pointcut combinator (Douence et al., 2001) $\mathit{pcSeq}\ \mathit{pc}_1\ \mathit{pc}_2$, that matches first pc_1 and then pc_2:

```
pcSeq :: (S_M Bool m) => PC m a b -> PC m c d -> PC m c d
pcSeq (PC mpc1) (PC mpc2) =
  PC $ do pc1 <- mpc1
          pc2 <- mpc2
          return $ \jp -> do b <- get
                             if b then pc2 jp
                                  else do b' <- pc1 jp
                                          put b'
                                          return False
```

As expressed in the $\mathbb{S}_{\mathrm{M}}\ \mathit{Bool}\ m$ constraint, the pointcut requires a boolean state in which to store the current point of its matching behavior: False (resp. True) means pc_1 (resp. pc_2) is to be matched. Consequently, any base program that modifies this state will alter the behavior of the pointcut. This situation can be avoided by using the non-interfering base computation type NIBase, just described above.

8.2 Interference Between Multiple Aspects

$\mathbb{NIA}_{\mathrm{T}}$ only distinguishes between base and aspect computation. Although useful, this implies that interference between aspects is still possible because all of them will share the same upper part of the monad stack. A similar situation happens with base programs and the lower part of the monad stack.

To illustrate this issue, consider a Fibonacci module program that uses the memo advice to improve the performance, and also uses a $\mathit{checkArg}$ advice that throws an exception when given a negative argument (instead of a base code check as in Figure 8.1). In this setting, $\mathit{checkArg}$ could update the cache with incorrect values, either accidentally or intentionally; or conversely, memo could throw arbitrary exceptions, even with a non-negative argument.

Finer-Grained Splitting of the Monad Stack Following the idea used in $\mathbb{NIA}_{\mathrm{T}}$, to enforce non-interference between $memo$ and $checkArg$ we need to split the monad stack into the monad for base computation m, and two upper layers t_1 and t_2. The idea is to assign to each aspect a unique layer in the stack, and to use parametricity to ensure non-interference. To this end we define the $\mathbb{NIA}_{\mathrm{T}_2}$ monad, which splits the monad stack as described. We also consider $niLift_2$, which serves the same role as $niLift$.

$$\mathbf{newtype}\ \mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m\ a = \mathbb{NIA}_{\mathrm{T}_2}\ (\mathbb{S}_{\mathrm{T}}\ (AspectEnv\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m))\ (t_1\ (t_2\ m))\ a)$$

Again, we extend other definitions properly ($weave$, etc.). Using rank-2 types, the following type synonyms guarantee that non-interfering pointcuts and advices access can only access the effect available in the first layer L_1, which corresponds to t_1; or in the second layer L_2, which corresponds to t_2.

$$\begin{aligned}
&\mathbf{type}\ NIPC_{L_1}\ t_1\ a\ b = \forall t_2\ m.(Monad\ m, MonadTrans\ t_1, MonadTrans\ t_2) \Rightarrow \\
&\qquad PC\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b \\
&\mathbf{type}\ NIPC_{L_2}\ t_2\ a\ b = \forall t_1\ m.(Monad\ m, MonadTrans\ t_1, MonadTrans\ t_2) \Rightarrow \\
&\qquad PC\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b \\
&\mathbf{type}\ NIAdvice_{L_1}\ t_1\ a\ b = \forall t_2\ m.(Monad\ m, MonadTrans\ t_1, MonadTrans\ t_2) \Rightarrow \\
&\qquad Advice\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b \\
&\mathbf{type}\ NIAdvice_{L_2}\ t_2\ a\ b = \forall t_1\ m.(Monad\ m, MonadTrans\ t_1, MonadTrans\ t_2) \Rightarrow \\
&\qquad Advice\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b
\end{aligned}$$

Non-Interference Combinators To enforce non-interference properties we need to define advice combinators, as we did with $niAdvice$. Again, we can enforce different non-interference patterns, by defining as many construction functions as required. We describe the advice combinators $niAdvice_{L_1}$ and $niAdvice_{L_2}$ that enforce that aspects work exclusively with the effect provided by the first and second layer, respectively.

$$\begin{aligned}
&niAdvice_{L_1} :: (Monad\ m, MonadTrans\ t_1, MonadTrans\ t_2) \Rightarrow \\
&\qquad NIAdvice_{L_1}\ t_1\ a\ b \to Advice\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b \\
&niAdvice_{L_1}\ adv = adv \\
&niAdvice_{L_2} :: (Monad\ m, MonadTrans\ t_1, MonadTrans\ t_2) \Rightarrow \\
&\qquad NIAdvice_{L_2}\ t_2\ a\ b \to Advice\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b \\
&niAdvice_{L_2}\ adv = adv
\end{aligned}$$

Now we define the monad stack S that provides the state and error-handling effects.

$$\mathbf{type}\ S = \mathbb{NIA}_{\mathrm{T}_2}\ (\mathbb{E}_{\mathrm{T}}\ String)\ (\mathbb{S}_{\mathrm{T}}\ (Map\ Int\ Int))\ \mathbb{I}$$

Then, we define the new fibonacci function using the $checkArg_{L_1}$ and $memo_{L_2}$ advices, which operate on the first and second layer of the monad stack, respectively.

$$\begin{aligned}
&fibMemoErr :: Int \to S\ Int \\
&fibMemoErr\ n = \mathbf{do}\ deploy\ (aspect\ pcFib\ (niAdvice_{L_2}\ memo_{L_2}))
\end{aligned}$$

$$
\begin{array}{l}
f \leftarrow \mathit{fib} \\
\mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ f)\ (\mathit{niAdvice}_{L_1}\ \mathit{checkArg}_{L_1})) \\
f \mathbin{\#} n
\end{array}
$$

The implementation of $\mathit{checkArg}_{L_1}$ is as follows:

$$
\begin{array}{l}
\mathit{checkArg}_{L_1}\ \mathit{proceed}\ \mathit{arg} = \\
\quad \mathbf{if}\ \mathit{arg} < 0 \\
\quad\quad \mathbf{then}\ (\mathit{niLift}_2 \circ \mathit{throwError})\text{``}\mathit{Error : negative\ argument}'' \\
\quad\quad \mathbf{else}\ \mathit{proceed}\ \mathit{arg}
\end{array}
$$

And similarly, we define memo_{L_2}:

$$
\begin{array}{l}
\mathit{memo}_{L_2}\ \mathit{proceed}\ n = \\
\quad \mathbf{do}\ \mathit{table} \leftarrow \mathit{niLift}_2\ \$\ \mathit{lift}\ \$\ \mathit{get} \\
\quad\quad \mathbf{if}\ \mathit{member}\ n\ \mathit{table} \\
\quad\quad\quad \mathbf{then}\ \mathit{return}\ (\mathit{table}\ !\ n) \\
\quad\quad\quad \mathbf{else}\ \ \mathbf{do}\ y \leftarrow \mathit{proceed}\ n \\
\quad\quad\quad\quad\quad\quad \mathit{table}' \leftarrow \mathit{niLift}_2\ \$\ \mathit{lift}\ \$\ \mathit{get} \\
\quad\quad\quad\quad\quad\quad (\mathit{niLift}_2 \circ \mathit{lift} \circ \mathit{put})\ (\mathit{insert}\ n\ y\ \mathit{table}') \\
\quad\quad\quad\quad\quad\quad \mathit{return}\ y
\end{array}
$$

Note that $\mathit{checkArg}_{L_1}$ is applied on calls to the *external* fibonacci function f, while memo_{L_2} is applied to the *internal* calls of the Fibonacci module, exposed by pcFib.

While an improvement over the binary base/aspect approach of EffectiveAdvice, illustrated in Section 8.1, this approach has two major drawbacks. First, it is not scalable because we need a different $\mathbb{NIA}_{\mathrm{T}_n}$ monad to support a setting with n mutually exclusive effects for aspects. Second, it is necessary to use explicit lifting in the implementation of advice. The reason is that we are explicitly using an effect from a layer at an arbitrary position in monad stack. Because we need to preserve parametricity to enforce non-interference, an advice cannot make *any* assumptions on the monad transformers that compose the stack. In particular, it cannot assume that the transformers support implicit liftings from the inner layers of the stack. In fact, in the presence of implicit lifting the layer from which an effect comes depends on the concrete monad stack used. These issues hamper modularity and reusability of aspects. In general, there is a tension between implicit lifting—designed to make a layer provide several effects at once—and splitting the monad stack with one aspect/effect per layer. In Section 8.4 we address these issues by using monad views (Schrijvers and Oliveira, 2011).

8.3 Background: Monad Views

Monad views, recently developed by Schrijvers and Oliveira (2011), are a technique for handling the monad stack, which extends and complements the standard mechanisms of explicit and implicit liftings (Section 2.3). Monad views provide robust support for accessing the effects of the monad

stack without being coupled to a particular stack layout. Views are denoted using $\rightsquigarrow$, and are an instance of the $View$ type class that defines the $from$ operation. Additionally, we use *bidirectional views*, denoted with the $\bowtie$ type operator. In addition to $from$, a bidirectional view supports the to operation. The types of these operations are:

$$\begin{array}{lll} from :: (Monad\ m, Monad\ n, View\ (\rightsquigarrow)) & \Rightarrow n \rightsquigarrow m \rightarrow n\ a \rightarrow m\ a \\ to\ \ :: (Monad\ m, Monad\ n) & \Rightarrow n \bowtie m \rightarrow m\ a \rightarrow n\ a \end{array}$$

In short, given two monads n and m, a view $n \rightsquigarrow m$ transforms computations from n to m, and a bidirectional view $n \bowtie m$ can also transform computations from m to n.

View-specific operations. Views are first-class values, hence they can be used as arguments. For instance, consider the functions $getv$ and $putv$ defined in (Schrijvers and Oliveira, 2011):

$$\begin{array}{l} getv :: (Monad\ m, \mathbb{S}_{\mathrm{M}}\ s\ n, View\ (\rightsquigarrow)) \Rightarrow (n \rightsquigarrow m) \rightarrow m\ s \\ getv\ v = from\ v\ \$\ get \\ putv :: (Monad\ m, \mathbb{S}_{\mathrm{M}}\ s\ n, View\ (\rightsquigarrow)) \Rightarrow (n \rightsquigarrow m) \rightarrow s \rightarrow m\ () \\ putv\ v = from\ v \circ put \end{array}$$

Given an initial monad m and a view $n \rightsquigarrow m$, $getv$ returns a computation $m\ s$ from an arbitrary state layer n. Conversely, $putv$ puts a new value into state layer n.

Creating views Schrijvers and Oliveira propose the construction of views using *structural* and *nominal masks*, which are applied onto the layers of a monad stack (Schrijvers and Oliveira, 2011).

- A structural mask is a bit-like mask applied to the monadic stack in order to hide the layers that conflict with implicit lifting. Such a mask is created by concatenating unary masks for each layer using the ::: type operator:[4] $\square$ indicates a visible layer and $\blacksquare$ a hidden layer.
- A nominal mask refers to layers of the stack using *names* instead of relative positions. This is done with the *tag monad transformer* $\mathbb{T}$. Given an arbitrary type Tag, the layer $\mathbb{T}^{Tag}$ labels a particular position of the monad stack using type Tag. An example of a tagged monad stack (for some types Tag_1 and Tag_2) is:

  $$\mathbf{type}\ M = \mathbb{T}^{Tag_1}\ (\mathbb{S}_{\mathrm{T}}\ Int\ (\mathbb{T}^{Tag_2}\ \mathbb{E}_{\mathrm{T}}\ String\ \mathbb{I}))$$

  where the $\mathbb{S}_{\mathrm{T}}$ layer is labeled with Tag_1 and the $\mathbb{E}_{\mathrm{T}}$ layer is labeled with Tag_2.
  For inspecting tagged monad stacks, the type class $n \sqsubseteq_{Tag} m$ exposes a monad n representing the layer of the stack m tagged with type Tag. It also provides the $structure$ operation to obtain the view between n and m associated to Tag:

  $$\begin{array}{l} \mathbf{class}\ (Monad\ m, Monad\ n) \Rightarrow n \sqsubseteq_{Tag} m\ \mathbf{where} \\ \quad structure :: View\ (\rightsquigarrow) \Rightarrow Tag \rightarrow (n \rightsquigarrow m) \end{array}$$

[4] We follow the graphical notation used in (Schrijvers and Oliveira, 2011)

8.4 Beyond the Aspect/Base Distinction

Monad views enable a different approach to enforce non-interference. The idea is that aspects will be generic with respect to the effects they require using type class constraints, assuming exclusive access to a monad stack with those effects. To avoid non-interference, client code uses a concrete monad stack and transforms each advice into a view-specific advice where the aspect only sees the sections of the monad stack that it is allowed to access.

For instance, the *memo* advice described in Figure 7.2 requires access to a dictionary to store the precomputed results. This is explicit in the (inferred) type of the advice:

$$memo :: (Monad\ m, Ord\ a, \mathbb{S}_{\mathrm{M}}\ (Map\ a\ b)\ m) \Rightarrow Advice\ m\ a\ b$$

In a similar way we define *checkArg*, which requires access to an error effect:

$$\begin{array}{l}
checkArg :: (Monad\ m, Num\ a, \mathbb{E}_{\mathrm{M}}\ String\ m) \Rightarrow Advice\ m\ a\ b \\
checkArg\ proceed\ arg = \\
\quad \mathbf{if}\ arg < 0 \\
\quad\quad \mathbf{then}\ throwError\ \text{``}Error : negative\ argument'' \\
\quad\quad \mathbf{else}\ \ proceed\ arg
\end{array}$$

Arbitrarily Splitting the Monad Stack with Views Observe now that the advice does not depend on the specific position of an effect in the monad stack. The novelty with respect to using implicit liftings is that we can assign to each aspect a *virtual view* of the monad stack that only contains the effect available to them. To assign a part of the monad stack to an advice we define the *withView* function:

$$\begin{array}{l}
withView :: (Monad\ n, Monad\ m) \Rightarrow n \bowtie m \rightarrow Advice\ n\ a\ b \rightarrow Advice\ m\ a\ b \\
withView\ v\ adv\ proceed\ arg = from\ v\ \$\ adv\ (\lambda a \rightarrow to\ v\ (proceed\ a))\ arg
\end{array}$$

This function transforms an advice from a restricted monad n to an advice in the "complete" stack m, using a bidirectional view provided as argument. We require a bidirectional view because we need to lift the *proceed* function, with type $a \rightarrow n\ b$ into an equivalent function with type $a \rightarrow m\ b$—which by construction performs effects only on n. Then, because evaluation of the restricted advice yields a computation $n\ b$, we use the *from* operation to lift it into a computation $m\ b$.

Observe that by partially applying *withView* with a given view we obtain a function of type $Advice\ n\ a\ b \rightarrow Advice\ m\ a\ b$, which fits with the notion of advice combinators (Section 7.3). Therefore it is possible to export protected pointcuts that expose a particular section of the monad stack to external advice. Additionally we can define functions to transform join points and pointcuts, in a similar way to *withView*.

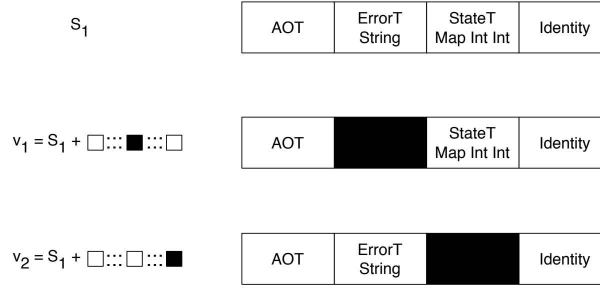


Figure 8.2: Applying structural masks to the monad stack S_1.

Using Structural Masks Consider a concrete monad stack S_1 which holds the required state and error effects.

$$\textbf{type}\ S_1 = \mathbb{A}_{\mathrm{T}}\ (\mathbb{E}_{\mathrm{T}}\ \mathit{String}\ (\mathbb{S}_{\mathrm{T}}\ (\mathit{Map\ Int\ Int})\ \mathbb{I}))$$

Then, we define the fibonacci function as follows:

$$\begin{aligned}
&\mathit{fibMemoErr}'\ n = \textbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ (\mathit{withView}\ v_1\ \mathit{memo}))\\
&\qquad\qquad\qquad\quad f \leftarrow \mathit{fib}\\
&\qquad\qquad\qquad\quad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ f)\ (\mathit{withView}\ v_2\ \mathit{checkArg}))\\
&\qquad\qquad\qquad\quad f \mathbin{\#} n\\
&\quad \textbf{where}\ v_1 = \square ::: \blacksquare ::: \square\\
&\qquad\qquad\ v_2 = \square ::: \square ::: \blacksquare
\end{aligned}$$

We define views v_1 and v_2 using structural masks. Both allow access to $\mathbb{A}_{\mathrm{T}}$, allowing AOP-specific operations into advice (*e.g.* deploying aspects). Besides that, v_1 exposes only the $\mathbb{S}_{\mathrm{T}}$ transformer, whereas v_2 only allows accessing to the $\mathbb{E}_{\mathrm{T}}$ transformer. Figure 8.2 depicts how views v_1 and v_2 define new *virtual monad stacks*, by applying structural masks to S_1. Note that structural masks can be applied only to monad transformers, but not to the monad at the bottom of the stack.

It is clear that now aspects do not need to perform explicit liftings and are not coupled to a particular monad stack. However, these issues are present when constructing views using nominal masks. Changes to the monad stack that is used to run client code need to be reflected in (potentially many) client functions that use structural masks.

Using Nominal Masks A more flexible approach that is not coupled to any particular monad stack is to use nominal masks to tag each effect required by aspects. Then client code can use the tags to directly access the effects and properly transform the advices. Consider a monad stack S_2, where the state and error layers are tagged:

$$\begin{aligned}
&\textbf{data}\ \mathit{StateTag}\\
&\textbf{data}\ \mathit{ErrorTag}\\
&\textbf{type}\ S_2 = \mathbb{A}_{\mathrm{T}}\ (\mathbb{T}^{\mathit{ErrorTag}}\ (\mathbb{E}_{\mathrm{T}}\ \mathit{String}\ (\mathbb{T}^{\mathit{StateTag}}\ (\mathbb{S}_{\mathrm{T}}\ (\mathit{Map\ Int\ Int})\ \mathbb{I}))
\end{aligned}$$

The stack is tagged at the type level, therefore we define two singleton types (with no data constructors), namely $\mathit{StateTag}$ and $\mathit{ErrorTag}$, to use as arguments for the $\mathbb{T}$ monad transformer.

The fibonacci function implemented using nominal masks is:

$$
\begin{array}{l}
\mathit{fibMemoErr''} :: \forall m\ n_1\ n_2.(\mathit{Monad}\ m, \\
\quad n_1 \sqsubseteq_{\mathit{StateTag}} (\mathbb{A}_{\mathrm{T}}\ m), \mathbb{S}_{\mathrm{M}}\ (\mathit{Map}\ \mathit{Int}\ \mathit{Int})\ n_1, \\
\quad n_2 \sqsubseteq_{\mathit{ErrorTag}} (\mathbb{A}_{\mathrm{T}}\ m), \mathbb{E}_{\mathrm{M}}\ \mathit{String}\ n_2) \\
\quad \Rightarrow \mathit{Int} \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ \mathit{Int} \\
\mathit{fibMemoErr''}\ n = \mathbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ (\mathit{withView}\ v_1\ \mathit{memo})) \\
\qquad\qquad\qquad\qquad f \leftarrow \mathit{fib} \\
\qquad\qquad\qquad\qquad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ f)\ (\mathit{withView}\ v_2\ \mathit{checkArg})) \\
\qquad\qquad\qquad\qquad f \mathbin{\#} n \\
\quad \mathbf{where}\ v_1 = \mathit{structure}\ \mathit{StateTag} :: n_1 \bowtie m \\
\qquad\qquad\ v_2 = \mathit{structure}\ \mathit{ErrorTag} :: n_2 \bowtie m
\end{array}
$$

In contrast to the previous definition, we need to use explicit type annotations because using nominal masks can lead to ambiguity in type inference[5]. Observe that we assume a monad m that is tagged with two singleton types $\mathit{StateTag}$ and $\mathit{ErrorTag}$. We use $\sqsubseteq$ to expose these layers as monads n_1 and n_2 respectively, and we constrain these monads to expose the corresponding effects. Therefore, by using nominal masks we can independently evolve the definition of S_2, as long as we keep the tagged layers expected by $\mathit{fibMemoErr''}$ (satisfying both the tag name and the required effect).

Perspectives on Using Views The content of the **do** expression is the same using structural or nominal masks. In fact it is possible to define a more generic function that takes views v_1 and v_2 as argument. Because views are first-class values, there is a wide design space on how to use them to control aspect interference. For example, aspects can be defined directly using $\sqsubseteq$ constraints as required. On the other hand, programmers must carefully define the views that are provided to each advice, because the typechecker cannot distinguish between intentional and accidental sharing of effects.

Controlling effect interference between aspects is a well-known and widely researched area in the AOP community. The two approaches presented in this chapter show that the concrete mechanism used to manage the monad stack determines the expressiveness of type-based reasoning techniques. We believe that the problem of assigning exclusive access to effects in the monadic stack originates from the fact that the monad stack is *public* and *transparent* to all components in a system. We conjecture that a mechanism that statically controls access to effects, while being flexible for developers ought to be devised, and indeed is a line of future work that transcends aspect-oriented programming. As a final remark, in a setting with an unrestricted deploy operation the restrictions on advice must be applied at each particular aspect deployment. This makes it difficult to establish global properties about advice in a system (which may require external static analysis). This can be solved with a custom $\mathbb{A}_{\mathrm{T}}$-like monad transformer that provides a more restricted deployment mechanism.

In the next chapter we focus on how to exploit the extensibility provided by our monadic weaver in order to modularly implement new aspect semantics, thus illustrating how our framework can be used as a research tool.

[5]The $\forall m\ n_1\ n_2$ annotation is required to use the type variables in the scope of a **do** expression.

Chapter 9

Modular Language Extensions

The typed monadic embedding of aspects supports modular extensions of the aspect language. The simplest extension is to introduce new user-defined pointcuts. More interestingly, because the language features a monadic weaver (Tabareau, 2012), we can modularly implement new semantics for aspect scoping and weaving. In addition, all language extensions benefit from the type-based reasoning techniques described before—to the best of our knowledge, this is a novel contribution of this work. In this chapter we describe the following developments:

- A user-defined control flow pointcut designator.
- Secure weaving, in which a set of join points can be hidden from advising.
- Privileged aspects that can see hidden join points from a secure computation.
- Aspect weaving with *execution levels* (Tanter, 2010).
- An example of type-based reasoning in the semantics of execution levels.

9.1 Control Flow Pointcut

An interesting illustration of extending the language with user-defined pointcuts is the case of control flow checks. Essentially, implementing the $pcCflow$ pointcut requires a way to track join points emitted during program execution. This tracking mechanism can be implemented modularly using a state monad transformer that holds a stack of join points, and an aspect that matches every join point, stores it in the stack, and then proceeds to obtain the result, which is returned after popping the stack. This corresponds to the stack-based implementation of $cflow$ described by Masuhara et al. (2003).

Join Point Stack To do this, we first define a join point stack as a list of existentially-quantified join points, EJP, just like we did to define the aspect environment as a list of homogeneous $EAspect$ values (Section 3.1).

$$\textbf{data}\ EJP = \forall a\ b\ m. Monad\ m \Rightarrow EJP\ (JP\ m\ a\ b)$$
$$\textbf{type}\ JPStack = [EJP]$$

Then, to collect the join points into a *JPStack* we define the $\mathbb{JP}_T$ monad transformer, reusing the implementation of the standard $\mathbb{S}_T$ transformer:

$$\mathbf{newtype}\ \mathbb{JP}_T\ m\ a = \mathbb{JP}_T\ (\mathbb{S}_T\ \mathit{JPStack}\ m\ a)$$

In addition, to support a polymorphic monad stack we define the $\mathbb{JP}_M$ type class as follows, and declare $\mathbb{JP}_T$ as an instance.

$$\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathbb{JP}_M\ m\ \mathbf{where} \\
\quad \mathit{getJPStack}\ \ \ :: m\ \mathit{JPStack} \\
\quad \mathit{pushJPStack} :: \mathit{EJP} \to m\ () \\
\quad \mathit{popJPStack}\ \ :: m\ () \\
\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathbb{JP}_M\ (\mathbb{JP}_T\ m)\ \mathbf{where} \ldots
\end{array}$$

Defining *pcCflow* Given the definitions above, the implementation of *pcCflow* is very similar to that of *pcCall* (Section 3.1).[1]

$$\begin{array}{l}
\mathit{pcCflow} :: \mathbb{JP}_M\ m \Rightarrow (a \to m\ b) \to \mathit{PC}\ m\ c\ (m'\ d) \\
\mathit{pcCflow}\ f = \mathit{return}\ \$\ \lambda_ \to \mathbf{do} \\
\quad \mathit{jpStack} \leftarrow \mathit{getJPStack} \\
\quad \mathit{return}\ \$\ \mathit{any}\ (\lambda \mathit{ejp} \to \mathit{compareFunEJP}\ f\ \mathit{ejp} \wedge \mathit{compareTypeEJP}\ f\ \mathit{ejp})\ \mathit{jpStack}
\end{array}$$

Here *compareFunEJP* checks the equality of the function bound to the join point and function f; and *compareTypeEJP* checks that the type of f is more general than the type of the join point. Function *any* returns whether any element of *jpStack* satisfies a given predicate. We can define the *pcCflowbelow* pointcut in a similar way.

Maintaining the Join Point Stack Now it remains to define the aspect that maintains the join point stack. We first define the *pcAny* pointcut, which matches all functions applications and pushes the corresponding join point into the stack.

$$\begin{array}{l}
\mathit{pcAny} :: \mathbb{JP}_M\ m \Rightarrow \mathit{PC}\ m\ a\ b \\
\mathit{pcAny} = \mathit{PC}\ \$\ \mathit{return}\ \$\ \lambda \mathit{jp} \to \mathbf{do}\ \mathit{pushJPStack}\ (\mathit{EJP}\ \mathit{jp}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathit{return}\ \mathit{True}
\end{array}$$

Note that the definition of *pcAny* preserves type soundness (Section 4.1.1) because its matched type is given by two fresh type variables a and b, and hence is the most general type possible. Next, we define *collectAdv* as an advice that performs *proceed*, pops the stack and returns the result.

$$\begin{array}{l}
\mathit{collectAdv}\ \mathit{proceed}\ \mathit{arg} = \mathbf{do}\ \mathit{result} \leftarrow \mathit{proceed}\ \mathit{arg} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mathit{popJPStack} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mathit{return}\ \mathit{result}
\end{array}$$

[1]Note that, as discussed in Section 4.1.1, we specifically declare that the matched type of the pointcut is in a different monad m'.

Finally, we define the $maintainJpStack$ aspect as follows.

$$maintainJpStack :: \mathbb{JP}_{\mathrm{M}}\ m \Rightarrow Aspect\ m\ a\ (m\ b)\ a\ b$$
$$maintainJpStack = aspect\ pcAny\ collectAdv$$

Although simple, this approach is inefficient because we are matching and storing all join points, instead of only those that can be queried in existing uses of $pcCflow$. Alternative optimizations can be defined, for example putting in the stack only relevant join points, or a per-flow deployment that allows using a boolean instead of a stack (Masuhara et al., 2003).

A consequence of not defining $pcCflow$ as a *primitive* pointcut is that we need to ensure that evaluation of $maintainJpStack$ occurs before than any other advice. Otherwise, control flow pointcuts from other aspects will have incorrect information to determine whether to execute the advice. This can be implemented directly in a custom $\mathbb{A}_{\mathrm{T}}$ transformer that takes a list of priority aspects and ensures they are always evaluated first during weaving.

9.2 Secure Weaving

For security reasons it can be interesting to protect certain join points from being advised. To support such a secure weaving, we define a new monad transformer $\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$, which embeds an (existentially quantified) pointcut that specifies the hidden join points, and we modify the weaving process accordingly (not shown here).

$$\textbf{data}\ EPC\ m = \forall a\ b. EPC\ (PC\ m\ a\ b)$$
$$\begin{aligned}\textbf{data}\ \mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m\ a = \mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ &(AspectEnv\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m) \rightarrow EPC\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m)\\ &\rightarrow m\ (a, (AspectEnv\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m), EPC\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m))))\end{aligned}$$

This can be particularly useful when used with the $pcCflow$ pointcut to protect the computation that occurs in the control flow of critical function applications. For instance, we can ensure that the whole control flow of function f is protected from advising during the execution of program p , assuming a function $run\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$, similar to $run\mathbb{A}_{\mathrm{T}}$ (Section 3.2):

$$run\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ (EPC\ (pcCflow\ f))\ p$$

9.3 Privileged Aspects

Hiding some join points to *all* aspects may be too restrictive. For instance, certain "system" aspects like access control should be treated as privileged and view all join points. Another example is the aspect in charge of maintaining the join point stack for the sake of control flow reasoning (used by $pcCflow$). In such cases, it is important to be able to define a set of privileged aspects, which can advise all join points, even those that are normally hidden in a secure computation. The implementation of a privileged aspects list is a straightforward extension to the secure weaving mechanism described above.

```
1  type Level = Int
2  newtype 𝔼𝕃T m a = 𝔼𝕃T (𝕊T Level m a)
3    -- primitive operations
4  inc  = 𝔼𝕃T $ λl → return ((), l + 1)
5  dec  = 𝔼𝕃T $ λl → return ((), l − 1)
6  at l = 𝔼𝕃T $ λ_ → return ((), l)
7    -- user-visible operations
8  current = 𝔼𝕃T $ λl → return (l, l)
9  up c     = do {inc; result ← c; dec; return result}
10 down c = do {dec; result ← c; inc; return result}
11 lambda_at f l = λarg → do n ← current
12                           at l
13                           result ← f arg
14                           at n
15                           return result
```

Figure 9.1: Execution levels monad transformer and level-shifting operations

9.4 Execution Levels

Execution levels avoid unwanted *computational interference* between aspects, *i.e.* when an aspect execution produces join points that are visible to others, including itself (Tanter, 2010). Execution levels give structure to execution by establishing a tower in which the flow of control navigates. Aspects are deployed at a given level and can only affect the execution of the underlying level. The execution of an aspect (both pointcuts and advices) is therefore not visible to itself and to other aspects deployed at the same level, only to aspects standing one level above. The original computation triggered by the last *proceed* in the advice chain is always executed at the level at which the join point was emitted. If needed, the programmer can use level-shifting operators to move execution up and down in the tower.

The monadic semantics of execution levels, first illustrated by Tabareau (2012), are implemented in the $\mathbb{EL}_\mathrm{T}$ monad transformer (Figure 9.1). The *Level* type synonym represents the level of execution as an integer. $\mathbb{EL}_\mathrm{T}$ wraps a *run* function that takes an initial level and returns a computation in the underlying monad m, with a value of type a and a potentially-modified level. As in the $\mathbb{A}_\mathrm{T}$ transformer, the monadic *bind* and *return* functions are the same as in the state monad transformer. The private operations *inc*, *dec*, and *at* are used to define the user-visible operations *current*, *up*, *down*, and *lambda_at*. In addition to level shifting with *up* and *down*, *current* reifies the current level, and *lambda_at* creates a *level-capturing function* bound at level l. When such a function is applied, execution jumps to level l and then goes back to the level prior to the application (Tanter, 2010).

The semantics of execution levels can be embedded in the definition of aspects themselves, by transforming the pointcut and advice of an aspect at deployment time, as shown in Figure 9.2.[2]

[2] For simplicity, in Section 3.2 we only described the default semantics of aspect deployment; aspect (un)deployment

$$
\begin{array}{l}
\mathit{deployInEnv}\ (\mathit{Aspect}\ (\mathit{pc} :: \mathit{PC}\ (\mathbb{A}_{\mathrm{T}}\ (\mathbb{EL}_{\mathrm{T}}\ m))\ \mathit{tpc})\ \mathit{adv})\ \mathit{aenv} = \\
\quad \mathbf{let} \\
\quad\quad \mathit{pcEL}\ \mathit{ldep} = (\mathit{PC}\ \$\ \mathit{return}\ \$\ \lambda \mathit{jp} \rightarrow \mathbf{do} \\
\quad\quad\quad \mathit{lapp} \leftarrow \mathit{current} \\
\quad\quad\quad \mathbf{if}\ \mathit{lapp} \equiv \mathit{ldep}\ \mathbf{then}\ \mathit{up}\ \$\ \mathit{runPC}\ \mathit{pc}\ \mathit{jp} \\
\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ \ \mathit{return}\ \mathit{False}) :: \mathit{PC}\ (\mathbb{A}_{\mathrm{T}}\ (\mathbb{EL}_{\mathrm{T}}\ m))\ \mathit{tpc} \\
\quad\quad \mathit{advEL}\ \mathit{ldep}\ \mathit{proceed}\ \mathit{arg} = \mathit{up}\ \$\ \mathit{adv}\ (\mathit{lambda_at}\ \mathit{proceed}\ \mathit{ldep})\ \mathit{arg} \\
\quad \mathbf{in}\ \mathbf{do}\ l \leftarrow \mathit{current} \\
\quad\quad\quad \mathit{return}\ \mathit{EAspect}\ (\mathit{Aspect}\ (\mathit{pcEL}\ l)\ (\mathit{advEL}\ l)) : \mathit{aenv}
\end{array}
$$

Figure 9.2: Redefining aspect deployment for execution levels semantics. An aspect is made level-aware by transforming its pointcut and advice.

$$
\begin{array}{l}
\mathit{showM}\ \ a = \mathit{return}\ (\mathit{show}\ \ a) \\
\mathit{logAdv}\ \mathit{proceed}\ \ a = \mathbf{do}\ \mathit{argStr} \leftarrow \mathit{showM}\ \#\ a \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{tell}\ (\texttt{"Arg: "} +\!\!\!+\ \mathit{argStr}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{result} \leftarrow \mathit{proceed}\ \ a \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{return}\ \mathit{result} \\
\mathit{program}\ \ n = \mathit{runM}\ \$\ \mathbf{do} \\
\quad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ (\mathit{showM} :: \rightarrow \mathit{Int} \rightarrow M\ \ \mathit{String}))\ \mathit{logAdv}) \\
\quad \mathit{showM}\ \#\ n
\end{array}
$$

Figure 9.3: A program that loops unless execution levels are used.

This is done by functions pcEL and advEL. pcEL first ensures that the current execution level lapp matches ldep, the level at which the aspect is deployed. If so it then runs the pointcut one level above. Similarly, advEL ensures that the advice is run one level above, with a $\mathit{proceed}$ function that captures the deployment level.

Example Figure 9.3 defines a generic logging advice, logAdv, which appends the argument and result of advised functions to the log[3]. In $\mathit{program}$, we deploy an aspect that intercepts all calls to showM (the monadic version of show) where the argument is of type Int (we require a type annotation for the pointcut because showM is a bounded polymorphic function—see Section 5.2 for details).

The evaluation of the program depends on the instantiation of the monad stack M. In a setting without execution levels, advising showM with logAdv triggers an infinite loop because logAdv internally performs open applications of showM, which are matched by the same aspect. Using the execution level semantics, evaluation terminates because the join point emitted by the advice is not

is actually defined using overloaded $(\mathit{un})\mathit{deployInEnv}$ functions.

[3]Using the tell function of the $\mathit{MonadWriter}$ class (denoted $\mathbb{W}_{\mathrm{M}}$), which is not described in Section 2.3, but which essentially is a state monad with append-only access.

visible to the aspect itself.

Interestingly, explicit open applications limit the possibilities of unwanted advising. More obliviousness, *e.g.* through partial application of $\#$, makes it harder to track down these issues. Nevertheless, identifying the source of the regression is not sufficient *per se*: in our example, if it is necessary for $logAdv$ to use open applications (so that other aspects can intervene), there is not much that can be done to avoid regression.

Beyond execution levels Execution levels adds a topological dimension to the composition of aspects into a system. However, their tower-like structure may be too restricted for certain scenarios, for instance for dynamic analyses aspects (Tanter et al., 2010). Recently, Tanter et al. (2012) proposed *programmable membranes* as a generalization of execution levels. We have developed a prototype implementation of membrane semantics in Effective Aspects (Figueroa et al., 2013), using the same approach of converting pointcuts and advices at deployment time. However, instead of passing the current level of execution (an integer), we maintain the bindings between membranes (a graph) using a state monad.

9.5 Reasoning about Language Extensions

The above extensions can be implemented in a dynamically typed language such as LAScheme (Tanter, 2010). However, it is challenging to provide any kind of reasoning about effects due to the dynamic nature of the language.

Enforcing Non-Interference in Language Extensions We can combine the monadic interpretation of execution levels with the management of effect interference (Chapter 8) in order to reason about level-shifting operations performed by base and aspect computations. For instance, it becomes possible to prevent aspect and/or base computation to use effects provided by the $\mathbb{EL}_{\mathrm{T}}$ monad transformer, thus ensuring that the default semantics of execution levels is preserved (and therefore that the program is free of aspect loops (Tanter et al., 2014)). For this we must consider a concrete monad stack that has the $\mathbb{A}_{\mathrm{T}}$ and $\mathbb{EL}_{\mathrm{T}}$ transformers on top:

$$\mathbf{type}\ \mathbb{AEL}_{\mathrm{T}}\ m = \mathbb{A}_{\mathrm{T}}\ \mathbb{EL}_{\mathrm{T}}\ m$$

Observe that this monad stack is general with respect to other effects it may contain. Then, we simply define an advice combinator that forbids access to the $\mathbb{EL}_{\mathrm{T}}$ layer, which provides the level-shifting operations, for instance:

$$levelAgnosticAdv = withView\ (\square ::: \blacksquare ::: \square)$$

This mask hides the layer with the execution-level-related effects, but allows access to $\mathbb{A}_{\mathrm{T}}$ at the top, and to the rest of the stack. Then to ensure level agnostic advice we just redefine $program$ to use this combinator, in a suitable monad stack M:[4]

[4]We use the $WriterT$ transformer ($\mathbb{W}_{\mathrm{T}}$), which is the canonical instance of $\mathbb{W}_{\mathrm{M}}$.

$$
\begin{array}{l}
\mathbf{type}\ M = \mathbb{AEL}_{\mathrm{T}}\ (\mathbb{W}_{\mathrm{T}}\ \mathit{String}\ \mathbb{I}) \\
\mathit{runM}\ c\ = \mathit{run}\mathbb{I}\ \$\ \mathit{run}\mathbb{W}_{\mathrm{T}}\ \$\ \mathit{run}\mathbb{EL}_{\mathrm{T}}\ (\mathit{run}\mathbb{A}_{\mathrm{T}}\ c)\ 0 \\
\mathit{program}\ n = \mathit{runM}\ \$\ \mathbf{do} \\
\quad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ (\mathit{showM} :: \rightarrow \mathit{Int} \rightarrow M\ \mathit{String}))\ (\mathit{levelAgnosticAdv}\ \mathit{logAdv})) \\
\quad \mathit{showM}\ \#\ n
\end{array}
$$

If more advanced use of execution levels is required, this contraint can be explicitly relaxed in the $\mathbb{A}_{\mathrm{T}}$ or $\mathbb{EL}_{\mathrm{T}}$ monad transformer, thus stressing in the type that it is the responsibility of the programmer to avoid infinite regression.

Using Types to Enforce Weaving Semantics The type system makes it possible to specify functions that can be woven, but only within a specific aspect monad. For instance, suppose that we want to define a *critical* computation, which must only be run with secure weaving for access control. The computation must therefore be run within the $\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$ monad transformer with a given pointcut *pc_ac* (*ac* stands for access control).

To enforce the use of $\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$ with a specific pointcut value would require the use of a dependent type, which is not possible in Haskell. This said, we can use the **newtype** data constructor together with its ability to derive automatically type class instances, to define a new type $\mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}$ that encapsulates the $\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$ monad transformer and forces it to be run with the *pc_ac* pointcut:

$$
\begin{array}{l}
\mathbf{newtype}\ \mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}\ m\ a = \mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m\ a)\ \mathbf{deriving}\ (\mathit{Monad}, \mathit{OpenApp}, \ldots) \\
\mathit{runSafe}\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}\ c) = \mathit{run}\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ (\mathit{EPC}\ \mathit{pc_ac})\ c
\end{array}
$$

Therefore, we can export the *critical* computation by typing it appropriately:

$$
\mathit{critical} :: \mathit{Monad}\ m \Rightarrow \mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}\ m\ a
$$

Because the $\mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}$ constructor is hidden in a module, the only way to run such a computation typed as $\mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}$ is to use *runSafe*. The *critical* computation is then only advisable with secure weaving for access control.

9.6 Other Approaches to Modular AOP Language Extensions

Although the extensibility of programming languages is an issue that goes beyond AOP, in this section we briefly discuss related work that directly addresses the issue of extending the semantics of aspect-oriented languages.

Extensible compilers The AspectBench Compiler (abc) (Avgustinov et al., 2006) is an extensible compiler to ease the development of AspectJ extensions. Its frontend is implemented using JastAdd (Ekman and Hedin, 2007), a general purpose extensible compiler for Java. Among the several AspectJ extensions based on we distinguish the recent work on Join Point Interfaces (**?**), and

the work on StrongAspectJ (De Fraine et al., 2008). We are not aware of other extensible compilers for other aspect languages, besides the ALIA4J framework discussed below.

ALIA4J framework The ALIA4J framework (Bockisch et al., 2011) goes beyond extensible compilers, aiming to be a general framework for the development of languages with advanced dispatching mechanisms. This is realized by a meta-model that serves as an intermediate representation language that can be later compiled into specific execution models. In particular, ALIA4J has been used to abstract the dispatching mechanism of the AspectJ and CaesarJ (Aracic et al., 2006) aspect languages.

Monadic interpreters Monadic interpreters (Liang et al., 1995) are the classic mechanism for modular language extensions in functional programming. Wand et al. (2004) defined a monadic interpreter to structure the denotational semantics of AspectJ. Although their work is untyped, it could benefit from the extensibility of monad transformers in order to feature modular language extensions. As we discussed before in Chapter 6, the extensibility features of our model come from the fact that our weaver is monadic.

This chapter ends the second part of this thesis, focused on controlling effects. We first described in Chapters 7 and 8 how to combine the principles of Open Modules with the techniques of EffectiveAdvice. A standing issue regarding our use of EffectiveAdvice is the lack of formal results regarding interference of aspects in a system. We have only shown that we can restrict certain pointcut/advice combinations to be deployed, which may be sufficient for local guarantees about aspect behavior. This issue is addressed in the third part of the thesis, where we establish how to reason compositionally about aspects in a system. Here the main challenge is that quantification forces us to reason about the context on which aspects are applied—in addition to the local properties of aspects.

Part III

Compositional Reasoning About Aspect Interference

Chapter 10

The Challenge of Compositional Reasoning

As we have illustrated in Part II, aspect-oriented programming promotes separation of concerns at the textual level, but semantic interactions between components of an aspect-oriented program are challenging to predict and control.

Consequently, the issue of interference has received a lot of attention in the AOP literature and related areas (which we discuss later in Chapter 14). In particular, Oliveira et al. (2012) developed MRI, which stands for *Modular Reasoning about Interference*, a purely functional model of incremental programming with effects. Effects are made explicit through the use of monads. MRI enables both modular reasoning and reasoning about non-interference of effects using a range of reasoning techniques like equational reasoning and parametricity. MRI has been used to express two theorems about harmless mixins. The central notion is that a mixin is harmless if the advised program is equivalent to the unadvised program, provided we ignore the effects introduced by the mixin. In MRI, harmlessness can be defined with respect to any computational effect, as long as an associated projection function exists to ignore the introduced effects. MRI therefore subsumes Dantas and Walker's notion of harmless advice, which is specific to I/O effects (Dantas and Walker, 2006).

While originally formulated as "EffectiveAdvice" (Oliveira et al., 2010) with a suggested connection to aspect-oriented programming, MRI does not address quantification; advices are mixins which are applied explicitly. The lack of quantification greatly simplifies modular reasoning, because it is enough to study a single module/function and a mixin in isolation. In addition, MRI only focuses on step-wise applications of mixins, in which the composition of a base component with a mixin can then be treated as a new base component for a subsequent mixin application. In contrast, in the pointcut/advice model of AOP, several aspects live in an aspect environment and are all woven at each join point.

The third part of this thesis addresses the challenge of reasoning about aspect interference in the presence of quantification. It has been argued that unrestricted quantification hampers modular reasoning, thereby requiring a form of global reasoning (Kiczales and Mezini, 2005). Recovering modular reasoning can be achieved by restricting quantification, for instance following the Open Modules approach (Aldrich, 2005). Yet, as we demonstrate in this work, while unrestricted quantification hampers *modular* reasoning, it is amenable to *compositional* reasoning: global harmlessness

results can be obtained through the composition of smaller proofs. This compositionality makes it possible to evolve an aspect-oriented system and *reuse* previously-established results.

In particular, we develop a framework for establishing harmlessness results about aspect-oriented systems in a compositional manner, using Haskell as a conveniente source language for System F_ω as it is done also in MRI. First, we establish an abstract monadic model of AOP, which is more amenable to analysis, with a level of complexity similar to that of MRI; and then we simplify the model presented in Chapter 3 in order to adapt it to this abstract specification. Finally, we formulate a general behavioral equivalence theorem between a given aspect-oriented system run with respect to two different aspect environments, modulo projection of additional side-effects. This general theorem is proven assuming four sufficient conditions that have to be established separately. When an aspect-oriented system evolves, only some of these conditions may need to be re-established in order to preserve the general theorem.

This chapters informally illustrates the challenges of compositional reasoning about aspect interference (Section 10.1) and complements the background on monadic programming (Section 2.3) by introducing the concept of monadic equational reasoning in Haskell, in particular the algebraic laws required of monads and monad transformers (Section 10.2).

10.1 Compositional Reasoning, Informally

To illustrate the challenges of reasoning about aspect interference, we introduce a simple base program (written in an imaginary ML-like language) defined in terms of some known functions f and g.

$$\begin{aligned} \mathit{prog}\ x\ y = & \ \mathbf{let}\ r_1 = f\ x\ \mathbf{in} \\ & \ \mathbf{let}\ r_2 = g\ y\ \mathbf{in} \\ & \ r_1 + r_2 \end{aligned}$$

In the following, we present different changes to a system composed of this program and some aspects, and consider questions related to semantic equivalence. We define aspects as a pointcut/advice pair, and use run to execute programs with certain aspects.

Adding aspects We first add an aspect to the existing system. For instance, to log all calls to f we define a new system:

$$s_1 = \mathit{run}\ [(\mathit{call}\ f, \mathit{log})]\ \mathit{prog}$$

with a typical implementation of the logging advice:

$$\begin{aligned} \mathit{log}\ \mathit{proceed}\ x = & \ \mathit{print}\ \texttt{"Entering function ..."} \\ & \ \mathit{proceed}\ x \end{aligned}$$

Is the behavior of s_1 equivalent to the original program? Strictly speaking, they are not equivalent if we consider the output generated by $print$. However, we observe that the return value of the

system is left unchanged, and that if we *ignore* the printed output, both systems are equivalent. This corresponds to the notion of *harmlessness* established in MRI (Oliveira et al., 2012). In the general case, establishing that applying the logging aspect is harmless requires to reason globally about the aspect and the composed system.

Some questions arise when we see, intuitively, that the logging advice is harmless for every function on which it may be applied. This property of logging when seen as a mixin is formalized and proven in MRI, but can we use this knowledge when the advice is applied to a system via quantification?

Widening quantification We now widen the quantification of the logging aspect, modifying the pointcut to match additional join points. For instance, if we now want to log calls to g, it suffices to define a combined pointcut:

$$s_2 = \mathit{run}\ [(\mathit{call}\ f \vee \mathit{call}\ g, \mathit{log})]\ \mathit{prog}$$

Intuitively, this change is also harmless. But how to prove it formally? Do we need to reason globally about the system from scratch? or can we reuse some facts from the proof that logging f in the system is harmless?

Evolving the base program We now evolve the base program by replacing the use of f with that of another function h:

$$\begin{aligned} \mathit{prog}'\ x\ y = {} & \mathbf{let}\ r_1 = h\ x\ \mathbf{in} \\ & \mathbf{let}\ r_2 = g\ y\ \mathbf{in} \\ & r_1 + r_2 \end{aligned}$$
$$s_3 = \mathit{run}\ [(\mathit{call}\ f \vee \mathit{call}\ g, \mathit{log})]\ \mathit{prog}'$$

A first observation is that $\mathit{call}\ f$ will never match. We must change references to f also in the aspect environment:

$$s_4 = \mathit{run}\ [(\mathit{call}\ h \vee \mathit{call}\ g, \mathit{log})]\ \mathit{prog}'$$

Changing f for h will most assuredly modify the semantics of the base program, and consequently of the system. This is expected when the base program is evolving. However, we may want to know if the logging aspect is still harmless in this new system. The question is: what amount of reasoning do we need to perform? Do we need to prove again that logging is harmless with respect to the whole system, or can we reason compositionally and only verify that the advice is harmless with respect to h?

Widening quantification, revisited Let us now consider a memoization aspect, with the following advice definition:

$$
\begin{array}{l}
\mathit{memo}\ \mathit{proceed}\ x = \mathbf{if}\ (\mathit{member}\ x\ \mathit{table}) \\
\qquad\qquad\qquad\qquad \mathbf{then}\ \mathit{table}\ [x] \\
\qquad\qquad\qquad\qquad \mathbf{else\ let}\ r = \mathit{proceed}\ x\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathit{insert}\ (\mathit{table}, x, r) \\
\qquad\qquad\qquad\qquad\qquad r
\end{array}
$$

The advice maintains a reference to a lookup $table$ of precomputed values, indexed by argument x. If the result bound to x is already in the table, it is immediately returned. Otherwise the value is computed, stored in the table for future references, and returned.

It is intuitively clear that adding memoization on calls to f is harmless. In fact, if we manually apply $memo$ as a mixin on top of f, then we even know formally that it is harmless (Oliveira et al., 2012).

Now, if we follow the quantification widening scenario from above—which was harmless with the logging advice—is the harmlessness of memoization preserved?

$$s_5 = \mathit{run}\ [(\mathit{call}\ f \vee \mathit{call}\ g, \mathit{memo})]\ \mathit{prog}$$

The answer to the question actually depends on the context in which the advice is applied. In a context where f and g actually are the same function, or one of both is never applied, then harmlessness is preserved. But if f and g are different functions that are both applied, the behavior of the composed system is drastically affected because the same lookup table is used to store results from both functions!

Compositional reasoning The examples presented above illustrate that, in presence of quantification, it is generally not enough to establish local properties for aspects, but it is also required to reason about the context in which those aspects are applied. Therefore, the modular reasoning techniques developed in the case of MRI are not directly applicable in a setting with quantification, because some form of global reasoning is generally required.

But global reasoning need not be monolithic, which is why we provide a formal framework to establish global equivalence properties in a compositional manner. Compositional reasoning facilitates the task of formally establishing properties about aspect-oriented programs. In practice, while it is possible to apply monolithic global reasoning to tiny systems like the ones considered in this section, this approach hardly scales to larger systems. Furthermore, compositional reasoning accommodates software evolution: it makes it possible to reuse previously-established results that are stable under the considered change scenarios.

10.2 Background: Monadic Reasoning in a Nutshell

The compositional reasoning framework proposed in this work is formulated in a monadic setting. We now complement the overview of monadic programming presented in Section 2.3, by introducing the core concepts of equational reasoning, observational equivalence, and the algebraic laws for monads and monad transformers.

$$
\begin{array}{ll}
\text{-- State} & \\
\mathbb{S}_{\mathrm{T}} \quad :: \mathbb{S}_{\mathrm{T}}\ s\ m\ a & \\
run\mathbb{S}_{\mathrm{T}} :: \mathbb{S}_{\mathrm{T}}\ s\ m\ a \rightarrow s \rightarrow m\ (a, s) & \\
\pi_S \quad :: s \rightarrow \mathbb{S}_{\mathrm{T}}\ s\ m\ a \rightarrow m\ a & \\
\mathbf{class}\ Monad\ m \Rightarrow \mathbb{S}_{\mathrm{M}}\ m \mid m \rightarrow s\ \mathbf{where} & \\
\quad get :: m\ s & \\
\quad put :: s \rightarrow m\ () &
\end{array}
$$

$$
\begin{array}{l}
\text{-- Writer} \\
\mathbb{W}_{\mathrm{T}} \quad :: \mathbb{W}_{\mathrm{T}}\ w\ m\ a \\
run\mathbb{W}_{\mathrm{T}} :: \mathbb{W}_{\mathrm{T}}\ w\ m\ a \rightarrow m\ (a, w) \\
\pi_W \quad :: \mathbb{W}_{\mathrm{T}}\ w\ m\ a \rightarrow m\ a \\
\mathbf{class}\ (Monoid\ w, Monad\ m) \Rightarrow \\
\qquad \mathbb{W}_{\mathrm{M}}\ w\ m \mid m \rightarrow w\ \mathbf{where} \\
\quad tell :: w \rightarrow m\ ()
\end{array}
$$

Figure 10.1: State and Writer monads transformers: constructors, evaluation and projection functions.

In addition, we summarize in Figure 10.1 the definitions of the state ($\mathbb{S}_{\mathrm{T}}$) and writer ($\mathbb{W}_{\mathrm{T}}$) monad transformers, used in the next chapters. The figure shows the types of their constructors ($\mathbb{S}_{\mathrm{T}}$, $\mathbb{W}_{\mathrm{T}}$), evaluation functions ($run\mathbb{S}_{\mathrm{T}}$, $run\mathbb{W}_{\mathrm{T}}$), and projection functions (π_S, π_W). The projection functions remove the corresponding effect from the monad stack (here, by discarding the threaded state or writer).

10.2.1 Equational Reasoning and Observational Equivalence

Equational reasoning is the process of transforming a program by replacing expressions in a manner similar to high-school algebra. Expression e_1 can be replaced by e_2 only if the two are *equivalent*. Observational equivalence, denoted as $\equiv$ in the paper, is an equivalence relation between expressions that holds whenever two expressions have the same observable behavior. That is, $e_1 \equiv e_2$ iff for every program context C, both $C[\mathrm{e}_1]$ and $C[\mathrm{e}_2]$ yield the same value, or both diverge. For example, consider the η-reduction rule from the λ-calculus, which states that $\lambda x \rightarrow f\ x \equiv f$ (when x is not free in f).

10.2.2 Monad Laws

Monad laws are crucial for equational reasoning in a monadic setting (Wadler, 1992). A proper monad is one that obeys the following three laws:

$$
\begin{array}{ll}
return\ x \gg\!= f\ \equiv f\ x & \text{-- left identity} \\
p \gg\!= return \quad \equiv p & \text{-- right identity} \\
(p \gg\!= f) \gg\!= h \equiv p \gg\!= \lambda x \rightarrow (f\ x \gg\!= h) & \text{-- associativity}
\end{array}
$$

The first two laws, left and right identity, state that $return$ neither changes the value nor performs any computational effect. The associativity law states that only the order of computations is relevant in a $\gg\!=$ expression. In the same way, monad transformers need to satisfy the following laws:

$$\begin{aligned} &lift \circ return \equiv return && \text{-- identity preservation} \\ &lift\ (m \gg\!\!= f) \equiv lift\ m \gg\!\!= (lift \circ f) && \text{-- comp. preservation} \end{aligned}$$

Note that Haskell does not enforce that declared instances of the $Monad$ or $MonadTrans$ classes actually respect these laws. This has to be proven separately for each considered instance.

We have illustrated informally why compositional reasoning is desirable. Because we can reuse certain proofs about the system or about aspects, the key benefit of compositional reasoning is that it scales to large systems—in contrast to monolithic global reasoning. Nevertheless, we need to be precise about what compositional reasoning is and under what scenarios we can reuse harmlessness proofs. We will make this claims precise in the following chapter, using equational reasoning and the monadic laws. In particular, we will establish a general theorem about compositional harmlessness, whose preconditions specify the situations where proofs must be re-established.

Chapter 11

Compositional Reasoning, Formally

11.1 Abstracting Monadic AOP

Our approach to compositional reasoning relies on a monadic formulation of AOP, but is independent from the concrete implementation of an aspectual computation monad transformer. In this chapter, we define an aspectual computation monad transformer denoted $\mathbb{A}_{\mathrm{T}}$ in an abstract manner, by prescribing its interface and properties. The theorem of compositional reasoning in Section 11.2 is established based on this abstract specification only.

11.1.1 Join Point Model

As before, we consider a join point model in which join points are function applications. However, here we abstract over any concrete design choice by introducing an abstract join point type, on which pointcuts predicate:

$$\begin{array}{l}\mathbf{data}\ \mathit{Jp}\ m\ a\ b \\ \mathbf{type}\ \mathit{Pc}\ m\ a\ b = \mathit{Jp}\ m\ a\ b \rightarrow \mathit{Bool}\end{array}$$

The type variables respectively denote the underlying monad stack, and the argument and return types of the applied function. The concrete representation of Jp can hold more information (*e.g.* contextual information, tags) or less, if some information is not meant to be used in pointcuts.

Because a denotational model cannot assume implicit generation of join points, we require the presence of an *open application* operator $\#$ that takes a function of type $a \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ b$ and returns a function of the same type whose application produces a join point (this effect is encapsulated in the $\mathbb{A}_{\mathrm{T}}$ monad transformer):

$$(\#) :: (a \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ b) \rightarrow (a \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ b)$$

Note that, in general, there is no reason to assume a single manner to generate join points, so there can indeed be a *family* of operators $\#^i$, which are interpreted by the aspect weaver as needed.

Finally, one can view a partial open application $f\ \#^i$ as an open function, whose application produces join points.

An advice is a function that executes in place of a join point matched by a pointcut. The first argument of the advice, typically called $proceed$, is a function which represents the original computation at the matched join point. An aspect simply pairs a pointcut with an advice.

$$\textbf{type}\ Advice\ m\ a\ b = (a \to m\ b) \to (a \to m\ b)$$
$$\textbf{type}\ Aspect\ m\ a\ b = (Pc\ m\ a\ b, Advice\ m\ a\ b)$$

Aspect environment The aspects to be deployed in a given aspectual computation are specified in a list of aspects called an *aspect environment*:

$\textbf{type}\ AEnv\ m = \ldots$ -- an ADT to be specified

As described in Chapter 4, supporting polymorphic aspects implies that the aspect environment should be an heterogeneous list. In order to avoid accidental complexity, we do not consider this issue in this abstract specification.

Aspectual computation Given a concrete $\mathbb{A}_{\mathrm{T}}$ transformer, we require a function that evaluates an $\mathbb{A}_{\mathrm{T}}$ computation given an aspect environment:

$$run\mathbb{A}_{\mathrm{T}} :: Monad\ m \Rightarrow AEnv\ (\mathbb{A}_{\mathrm{T}}\ m) \to \mathbb{A}_{\mathrm{T}}\ m\ a \to m\ a$$

Abstracting open applications Similarly to the $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{W}_{\mathrm{M}}$ type classes, we introduce a type class to define an abstract interface for performing open applications:

$$\textbf{class}\ Monad\ m \Rightarrow \mathbb{A}_{\mathrm{M}}\ m\ \textbf{where}$$
$$\quad \#^i :: (Int \to m\ Int) \to (Int \to m\ Int)$$
$$\textbf{instance}\ Monad\ m \Rightarrow \mathbb{A}_{\mathrm{M}}\ (\mathbb{A}_{\mathrm{T}}\ m)\ \textbf{where} \ldots$$

The only operation of this class is $\#^i$, and we require that any monad $\mathbb{A}_{\mathrm{T}}\ m$ be an instance of this class. Note that $\mathbb{A}_{\mathrm{M}}$ allows a form of type-based reasoning about open applications: any function of type $\forall m.D\ m \Rightarrow a \to m\ b$, where D is a class constraint that does not entail $\mathbb{A}_{\mathrm{M}}$, cannot perform any open applications (and hence cannot emit join points).

11.1.2 Necessary Properties of $\mathbb{A}_{\mathrm{T}}$

To be a correct model, the $\mathbb{A}_{\mathrm{T}}$ transformer needs to satisfy a number of properties. First, it has to satisfy the monad transformer laws, and when applied to any monad m, the monad laws must be satisfied as well. Moreover, for all aspect environments $aenv$, the function $run\mathbb{A}_{\mathrm{T}}\ aenv$ must be a *monad morphism*.

Definition 8 A monad morphism h is a function of type

$$h :: \forall a. M_1\ a \rightarrow M_2\ a$$

that transforms computations in one monad M_1 into computations in another monad M_2. The function satisfies two laws:

$$\begin{array}{rcll} h \circ \mathit{return} & \equiv & \mathit{return} & \\ h\ (m \gg\!= f) & \equiv & h\ m \gg\!= h \circ f & (\forall m, f) \end{array}$$

For $\mathit{run}\mathbb{A}_{\mathrm{T}}$, the first monad is $\mathbb{A}_{\mathrm{T}}\ m$ and the second monad is just m. Moreover, the two monad morphism laws have an intuitive meaning in this setting: the first law expresses that weaving has no impact on pure computations, and the second law expresses that weaving is compositional.[1]

In the same spirit, we also require that a third law holds for $\mathit{run}\mathbb{A}_{\mathrm{T}}\ \mathit{aenv}$:[2]

$$\mathit{run}\mathbb{A}_{\mathrm{T}}\ \mathit{aenv} \circ \mathit{lift} \ \equiv\ \mathit{id}$$

This law expresses that $\mathit{run}\mathbb{A}_{\mathrm{T}}\ \mathit{aenv}$ is a left inverse of lift. In words, weaving an effectful computation that does not involve open applications has no impact.

These laws have to be established whenever a concrete $\mathbb{A}_{\mathrm{T}}$ transformer is implemented. We will come back to this when presenting a simple $\mathbb{A}_{\mathrm{T}}$ transformer in Chapter 12.

11.1.3 Running Example in Monadic Style

Section 10.1 used pseudo-code to describe a base program and aspects. In Haskell, the base program is defined in monadic style using the **do** notation as follows:

$$\begin{array}{l} \mathit{prog}\ x\ y = \mathbf{do}\ r_1 \leftarrow f\ \#^i\ x \\ \qquad\qquad\qquad\ \ r_2 \leftarrow g\ \#^j\ y \\ \qquad\qquad\qquad\ \ \mathit{return}\ (r_1 + r_2) \end{array}$$

The program can be run as an aspectual computation in the $\mathbb{A}_{\mathrm{T}}$ transformer with a logging aspect on open applications of f as follows:

$$\mathit{run}\mathbb{A}_{\mathrm{T}}\ [(\mathit{fPc}, \mathit{log})]\ (\mathit{prog}\ 5\ 12)$$

The pointcut fPc is left undefined at this stage, since in this abstract model we do not prescribe a specific way to denote functions. The definitions of the log and memo advices in monadic style are given in Figure 11.1.

---

[1] If $\mathbb{A}_{\mathrm{T}}$ supports dynamic deployment of aspects, as in the model described in Chapter 3, weaving cannot be compositional. We can nevertheless prove the monad morphism laws for the static fragment, and deal with dynamic deployment on a case-by-case basis.

[2] This law actually subsumes the first monad morphism law, as $\mathit{return} \equiv \mathit{lift} \circ \mathit{return}$.

$$
\begin{array}{l}
\mathit{log} :: \mathbb{W}_{\mathrm{M}}\ \mathit{String}\ m \Rightarrow \mathit{Advice}\ m\ a\ b \\
\mathit{log}\ \mathit{proceed}\ x = \textbf{do}\ \mathit{tell}\ \texttt{"Entering function ..."} \\
\qquad\qquad\qquad\qquad\ \mathit{proceed}\ x \\
\mathit{memo} :: (\mathit{Ord}\ a, \mathbb{S}_{\mathrm{M}}\ (\mathit{Map}\ a\ b)\ m) \Rightarrow \mathit{Advice}\ m\ a\ b \\
\mathit{memo}\ \mathit{proceed}\ x = \textbf{do} \\
\quad \mathit{table} \leftarrow \mathit{get} \\
\quad \textbf{if}\ \mathit{member}\ x\ \mathit{table} \\
\qquad \textbf{then}\ \mathit{return}\ (\mathit{table}\ !\ x) \\
\qquad \textbf{else do}\ y \leftarrow \mathit{proceed}\ x \\
\qquad\qquad\qquad \mathit{table}' \leftarrow \mathit{get} \\
\qquad\qquad\qquad \mathit{put}\ (\mathit{insert}\ x\ y\ \mathit{table}') \\
\qquad\qquad\qquad \mathit{return}\ y
\end{array}
$$

Figure 11.1: Logging and memoization advice in monadic style

11.2 Compositional Harmlessness Theorem

This section formalizes our approach to compositional reasoning about aspect interference. This approach revolves around the following general theorem, which provides a framework for the reasoning. The theorem considers an AOP system that is run with respect to a particular aspect environment aenv. The theorem states that, under four sufficient conditions, the system preserves its observable behavior under an alternative aspect environment aenv' that may introduce additional effects. With the four conditions it provides a step-by-step guide to proving non-interference.

A key property of the theorem is that it supports *compositional* reasoning. Compositionality is achieved because the theorem splits the system into two parts, an open function $f\ \#^i$ and a *context* c, whose conditions are independent, can be proven separately, and can be reused in different compositions. Moreover, the system can easily be decomposed into all the individual open functions (rather than just two parts) by repeated application of the theorem. In fact, the third condition below, which relates to the context, is an instance of the theorem and thus explicitly invites this systematic decomposition.

Theorem 2 (Compositional Harmlessness Theorem) Given an expression:

$$\mathit{system} :: \forall m.C\ m \Rightarrow A \to \mathbb{A}_{\mathrm{T}}\ m\ B$$

Here A and B are some types, and m is a type variable constrained by some type class constraints C that at least require m to be an instance of Monad.

We assume that system is given in terms of the following decomposition:

$$\mathit{system} \equiv c\ (f\ \#^i)$$

where c, f and i are arbitrary values of the following types (with C_f entailed by C; again A' and

B' are some types):

$$c :: \forall m. C\ m\ \Rightarrow (A' \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ B') \rightarrow A \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ B$$
$$f :: \forall m. C_f\ m \Rightarrow A' \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ B'$$

Also, we are given two aspect environments $aenv$ and $aenv'$ of types:

$$aenv\ :: \forall m. D\ m \Rightarrow AEnv\ (\mathbb{A}_{\mathrm{T}}\ m)$$
$$aenv' :: \forall m. D\ m \Rightarrow AEnv\ (\mathbb{A}_{\mathrm{T}}\ (T\ m))$$

where T is some instance of $MonadTrans$ and D is a type class constraint that at least requires m to be an instance of $Monad$.

The given projection function:

$$\pi :: \forall m\ a. Monad\ m \Rightarrow T\ m\ a \rightarrow m\ a$$

is a left-inverse of $lift$ that removes the additional T effect from the monad stack $T\ m$.

If the four conditions on c and f given below hold, then we have that:

$$run\mathbb{A}_{\mathrm{T}}\ aenv\ system \equiv \pi\ (run\mathbb{A}_{\mathrm{T}}\ aenv'\ system)$$

The four conditions on c and f are:

1. Compositional weaving

$$\forall env. run\mathbb{A}_{\mathrm{T}}\ env\ (c\ (f\ \#^i)) \equiv run\mathbb{A}_{\mathrm{T}}\ env\ c\ (lift \circ run\mathbb{A}_{\mathrm{T}}\ env \circ (f\ \#^i))$$

2. Compositional projection

$$\pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ c\ (lift \circ lift \circ \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ (f\ \#^i))$$
$$\equiv$$
$$\pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ c\ (lift \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ (f\ \#^i))$$

3. Contextual harmlessness

$$run\mathbb{A}_{\mathrm{T}}\ aenv \circ c \circ (\lambda g \rightarrow lift \circ g) \equiv \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ c \circ (\lambda g \rightarrow lift \circ lift \circ g)$$

4. Local harmlessness

$$run\mathbb{A}_{\mathrm{T}}\ aenv \circ (f\ \#^i) \equiv \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ (f\ \#^i)$$

PROOF. The proof proceeds by straightforward equational reasoning:

$$
\begin{array}{l}
\quad run\mathbb{A}_{\mathrm{T}}\ aenv\ system \\
\equiv\ \ \{\text{-system decomposition -}\} \\
\quad run\mathbb{A}_{\mathrm{T}}\ aenv\ (c\ (f\ \#^{i})) \\
\equiv\ \ \{\text{-compositional weaving -}\} \\
\quad run\mathbb{A}_{\mathrm{T}}\ aenv \circ c\ (lift \circ run\mathbb{A}_{\mathrm{T}}\ aenv \circ f\ \#^{i}) \\
\equiv\ \ \{\text{-local harmlessness -}\} \\
\quad run\mathbb{A}_{\mathrm{T}}\ aenv \circ c\ (lift \circ \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ f\ \#^{i}) \\
\equiv\ \ \{\text{-contextual harmlessness -}\} \\
\quad \pi\ (run\mathbb{A}_{\mathrm{T}}\ aenv' \circ c\ (lift \circ lift \circ \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ f\ \#^{i})) \\
\equiv\ \ \{\text{-compositional projection -}\} \\
\quad \pi\ (run\mathbb{A}_{\mathrm{T}}\ aenv' \circ c\ (lift \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ f\ \#^{i}) \\
\equiv\ \ \{\text{-compositional weaving -}\} \\
\quad \pi\ (run\mathbb{A}_{\mathrm{T}}\ aenv'\ (c\ (f\ \#^{i}))) \\
\equiv\ \ \{\text{-system decomposition -}\} \\
\quad \pi\ (run\mathbb{A}_{\mathrm{T}}\ aenv'\ system)
\end{array}
$$

□

We now explain and illustrate how the theorem can be used.

11.2.1 System Decomposition

The starting point is to *view* the system as the composition of a particular function f and a context c. For instance, we can write our running example as $c_1\ (f_1\ \#^{i})$ where

$$
\begin{array}{l}
f_1 = f \\
c_1 = \lambda f\ x\ y \rightarrow \mathbf{do}\ r_1 \leftarrow f\ x \\
\qquad\qquad\qquad\qquad\ \ r_2 \leftarrow g\ \#^{j}\ y \\
\qquad\qquad\qquad\qquad\ \ return\ (r_1 + r_2)
\end{array}
$$

Here the context c_1 is just $system$ abstracted over $f\ \#^{i}$. Note that the same $system$ can be decomposed in many different ways, in order to focus on different open functions.

11.2.2 Compositional Weaving

The first condition states that weaving the composite system is equivalent to weaving the context c and the function f separately and then composing them.

While the compositional weaving condition is formulated in terms of the specific c and f, it comes *almost* for free from the three laws that $run\mathbb{A}_{\mathrm{T}}\ env$ satisfies (recall Section 11.1.2). To see why, let us consider the essential ways in which c can use f. There are two permitted ways:

1. c does nothing with f, and thus whether f is woven or not is inconsequential.
2. c invokes f (once or more), which means embedding it in its larger computation (once or more) with $\gg=$, which is where the second law comes in. Note that the second law can be used repeatedly to tackle a larger computation sequence $m \gg= f_1 \gg= \ldots \gg= f_n$.

However, there is also one way in which the condition can be violated:

3. The context c is itself weaving the open function with a custom aspect environment. One such example is:

$$c = \lambda f \rightarrow \mathit{lift} \circ \mathit{run}\mathbb{A}_{\mathrm{T}}\ [\,] \circ f$$

where c weaves the function with an empty aspect environment, irrespective of the aspect environment used to weave c itself.

This illegal use of f can be avoided by introducing a measure of parametricity. Instead of using the fixed monad transformer $\mathbb{A}_{\mathrm{T}}$ and its fixed function $\#^i$ in c and f, we make c and f parametric in the particular type and function definition. This parameterization is conveniently achieved by imposing the $\mathbb{A}_{\mathrm{M}}$ constraint on the monad stack instead of applying the $\mathbb{A}_{\mathrm{T}}$ transformer. It prevents c from invoking the weaving function $\mathit{run}\mathbb{A}_{\mathrm{T}}$ locally on f because $\mathit{run}\mathbb{A}_{\mathrm{T}}$ only works for $\mathbb{A}_{\mathrm{T}}\ m'$ and not for all possible m that instantiate $\mathbb{A}_{\mathrm{M}}$. We summarize our technique for establishing compositional weaving in the following conjecture.

Conjecture 1 Provided that f and c have the following polymorphic types:

$$\begin{array}{l} c :: \forall m.(C\ \ m, \mathbb{A}_{\mathrm{M}}\ m) \Rightarrow (A' \rightarrow m\ B') \rightarrow A \rightarrow m\ B \\ f :: \forall m.(C_f\ m, \mathbb{A}_{\mathrm{M}}\ m) \Rightarrow\ \ A' \rightarrow m\ B' \end{array}$$

the condition of compositional weaving holds.

We believe that this conjecture can be proven with logical relations, which is a rather technically challenging task that is currently out of the scope of this work.

11.2.3 Compositional Projection

The second condition expresses that composing the projected context c and projected function f is equivalent to projecting the composition.

This condition has a similar shape as that for compositional weaving. Hence, in the case that the projection function π is a monad morphism, then the same solution as for compositional weaving applies. For instance, the projection π_W of the writer effect (used in the logging advice) is well-known (and easily verified) to be a monad morphism. This means that, if the system abstracts over the implementation of the writer effect with the type class constraint $\mathbb{W}_{\mathrm{M}}$, then its projection is indeed compositional.

However, it is a very strong requirement for the projection function to be a monad morphism.

For instance, the projection π_S of the state effect is *not* a monad morphism:

$$\begin{aligned} \pi_S\ 0\ (get \ggg\!= \lambda x \to put\ (x+1) \gg get) &\equiv return\ 1 \\ \pi_S\ 0\ (get \ggg\!= \lambda x \to put\ (x+1)) \gg \pi_S\ 0\ get &\equiv return\ 0 \end{aligned}$$

This explains why we must be careful when adding the $memo$ advice of Figure 11.1, which has a memo table as its state, to our running example. If the pointcut of this advice matches both the function f on the one hand and the function g in the context c on the other hand, then the two uses of the advice may interfere through the shared state. For instance, the result for $f\ 3$ may be stored in the table and later wrongly used as if it were the result for $g\ 3$. This problem is not discovered when we consider the impact of $memo$ on c and f separately. On the contrary, $memo$ is contextually and locally harmless, but globally harmful. We only discover this problem because compositional projection does not hold. This illustrates why compositional projection is a crucial condition.

In some cases, the use of $memo$ in a larger system is nevertheless harmless. As we cannot take the monad morphism route to establishing this, we need to resort to alternative techniques.

- If the woven function $run\mathbb{A}_{\mathrm{T}}\ aenv' \circ (f\ \#^i)$ does not use the projected effect, then projection is indeed compositional. This is for instance the case when $memo$ does not advise f. We can formally capture this as:

  $$\exists h, lift \circ h \equiv run\mathbb{A}_{\mathrm{T}}\ aenv' \circ (f\ \#^i)$$

  Let us now reason about the relevant part of the left-hand side of the condition:

  $$\begin{aligned} & lift \circ lift \circ \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ (f\ \#^i) \\ \equiv\ & \{\text{-assumption -}\} \\ & lift \circ lift \circ \pi \circ lift \circ h \\ \equiv\ & \{\text{-}\pi \text{ is left inverse of } lift \text{ -}\} \\ & lift \circ lift \circ h \\ \equiv\ & \{\text{-assumption -}\} \\ & lift \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ (f\ \#^i) \end{aligned}$$

  If we plug this conclusion into the left-hand side of the compositional projection condition, we obtain its right-hand side. In other words, the condition follows from the assumption.

- The dual assumption from the above is that the context c does not use the projected effect. This is for instance the case when $memo$ advises f but not c. Unfortunately, this case is not as straightforward. While c does not directly interfere with the effect, it may indirectly create interference by invoking f repeatedly and those invocations may interfere with one another through their shared effect. This requires reasoning about the compatibility of an advised f with itself. For instance, in the case of $memo$ it is perfectly fine for multiple invocations of f to share the memo table; in fact, that is exactly the point of memoization. A counterexample is an advice that monitors whether a function is invoked at most n number of times, where n is the first input its called with, and raises an error when that limit is exceeded. This advice is perfectly fine for a function in isolation that takes n (recursive) calls, but when there are multiple separate invocations, then the error may be triggered inadvertently.

Note that we can safely memoize both f and g in our example, if separate tables are used. This amounts to using two instances of $memo$ that each act on a different $\mathbb{S}_{\mathrm{T}}$ layer in the monad stack. In this setup, the state of the components is isolated from each other. Hence, this scenario involves the two classes of compositional projection discussed above.

11.2.4 Contextual Harmlessness

The third condition expresses that as far as the context c is concerned, the aspect environments $aenv$ and $aenv'$ are indistinguishable. There are various ways in which $aenv$ and $aenv'$ can be related for this to be true, for example:

- Unused aspects (pc, a), where the pointcut pc does not match any join point in c, can be freely added or removed.
- Two aspects (pc_1, a_1) and (pc_2, a_2) can be reordered if they either do not match on the same applications in c or their advices commute ($a_1 \circ a_2 \equiv a_2 \circ a_1$).
- The pointcut of an aspect can be replaced by one that matches the same join points in c.
- The advice of an aspect can be replaced by one that behaves in the same way with respect to c.
- Multiple aspects can be replaced simultaneously by another set of aspects that together behave in the same way on c, redistributing the work among themselves, *e.g.* splitting a predicate into two disjoint ones.

Note that the contextual harmlessness condition is a variant of the general theorem itself, but on a smaller system that only consists of the context c. Hence, it can be proven by recursively decomposing the context and invoking the general theorem on the two parts. This insight is essential to scale up our approach from a two-function system to arbitrarily complex systems.

For instance, in the running example we can build a simpler system from c_1, namely $c_1\ (\mathit{lift} \circ h)$, where $h :: C\ m \Rightarrow A' \rightarrow m\ B'$ is universally quantified. This form is *smaller* than the original system because it features fewer open applications; h's type is constrained to not feature any. The resulting system has the form:

$$
\begin{aligned}
system' = \lambda x\ y \rightarrow \mathbf{do}\ & r_1 \leftarrow \mathit{lift}\ (h\ x) \\
& r_2 \leftarrow g\ \#^j\ y \\
& \mathit{return}\ (r_1 + r_2)
\end{aligned}
$$

which can be decomposed as $system' = c_2\ (f_2\ \#^j)$:

$$
\begin{aligned}
f_2 &= g \\
c_2 &= \lambda g\ x\ y \rightarrow \mathbf{do}\ r_1 \leftarrow \mathit{lift}\ (h\ x) \\
&\qquad\qquad\qquad\quad r_2 \leftarrow g\ y \\
&\qquad\qquad\qquad\quad \mathit{return}\ (r_1 + r_2)
\end{aligned}
$$

Here we can consider the harmlessness of the extended environment $aenv'$ separately for g and c_2. Note that since c_2 does not contain any more open applications, contextual harmlessness is trivially established for it.

11.2.5 Local Harmlessness

The fourth condition requires the harmlessness of the extended aspect environment $aenv'$ with respect to a single function seen in isolation. In our recursively decomposed example, this means we can study the impact of $aenv'$ on f and g individually.

We do not go into detail here, but devote Chapter 13 to adapting the techniques of MRI for proving this condition in our setting. These techniques involve both regular proofs based on equational reasoning over the actual implementations of function and advice, as well as the more lightweight parametricity-based techniques that only need to consider the types.

In this chapter we have developed the main contribution of the third part of the thesis. The compositional harmlessness theorem precisely describes four sufficient conditions to establish that swapping an aspect environment for another one preserves the semantics of a system. The theorem is established in terms of an abstract monadic AOP system, which must fulfill certain characteristics. In the next chapter we instantiate a simple monadic AOP model, which is a simplification of the full-fledged model presented in the first two parts of the thesis, in order to illustrate the use of the compositional harmlessness theorem. In a broader sense, a main challenge that remains is how to translate—at least intuitively—the preconditions and the conclusion of the theorem, which are rather technical and too specific to the monadic setting, to other aspect languages.

Chapter 12

A Simple Monadic AOP Model

In order to illustrate concrete applications of compositional reasoning about aspect interference, we now describe a simple monomorphic monadic model of pointcut/advice AOP in Haskell. The model is a simplification of the monadic embedding of aspects described in Chapter 3. The main differences are that this model:

1. does not support polymorphic aspects; only functions of type $Int \to m\ Int$, for some monad m, are open to advice.
2. only has pure pointcuts, *i.e.* pointcuts that cannot use monadic effects.
3. uses an abstract syntax tree of computations that expose function applications as join points and turn it into a monad transformer.
4. does not support dynamic aspect deployment; $\mathbb{A}_{\mathrm{T}}$ computations are evaluated under a fixed aspect environment.
5. uses a more general model of *tagged* open weaving to specify quantification.

12.1 An Embedding of Open Applications

We implement $\mathbb{A}_{\mathrm{T}}$ as a monad transformer that captures open function applications in a syntactic form.[1] The interpreter function $run\mathbb{A}_{\mathrm{T}}$ interpretes the open applications by weaving them with the aspect environment.

Join point model Join points represent open function application. In order not to deal with function equality or type comparisons as discussed in Section 5.1.1, we rely on *tagged* applications: pointcuts match join points based on tag equality ($pcTag$). Here, tags are just integers:

$$\begin{array}{l}\textbf{type}\ Tag = Int \\ \textbf{data}\ Jp\ m\ a\ b = Jp\ Tag\end{array}$$

[1] Our $\mathbb{A}_{\mathrm{T}}$ implementation is a close cousin of a free monad.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathbb{A}_{\mathrm{T}}\ m)\ \textbf{where} \\
\quad \mathit{return} \ \ = \mathbb{A}_{\mathrm{T}} \circ \mathit{return} \circ \mathit{Return} \\
\quad m \gg\!= f = \mathbb{A}_{\mathrm{T}}\ (\mathit{un}\mathbb{A}_{\mathrm{T}}\ m \gg\!= \lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \\
\qquad \mathit{Return}\ x \rightarrow \mathit{un}\mathbb{A}_{\mathrm{T}}\ (f\ x) \\
\qquad \mathit{OpenApp}\ t\ g\ x\ k \rightarrow \\
\qquad\quad \mathit{return}\ (\mathit{OpenApp}\ t\ g\ x\ (\lambda y \rightarrow k\ y \gg\!= f))) \\
\textbf{instance}\ \mathit{MonadTrans}\ \mathbb{A}_{\mathrm{T}}\ \textbf{where} \\
\quad \mathit{lift}\ ma = \mathbb{A}_{\mathrm{T}}\ (ma \gg\!= \lambda a \rightarrow (\mathit{return} \circ \mathit{Return})\ a)
\end{array}
$$

Figure 12.1: $\mathbb{A}_{\mathrm{T}}$ instances for the Monad and $\mathit{MonadTrans}$ type classes.

$$\mathit{pcTag}\ t\ (\mathit{Jp}\ t') = t \equiv t'$$

Note that in this simple instantiation of monadic AOP, join points only embed the tag of an open application, and neither the applied function nor the argument.

Defining the monad transformer The $\mathbb{A}_{\mathrm{T}}$ transformer extends a given monad m with the ability to expose some open function applications. A computation $\mathbb{A}_{\mathrm{T}}\ m\ a$ is denoted by an alternating sequence of computations in the monad m and exposed open function applications starting with the former.

$$
\begin{array}{l}
\textbf{data}\ \mathbb{A}_{\mathrm{T}}\ m\ a = \mathbb{A}_{\mathrm{T}}\ \{\mathit{un}\mathbb{A}_{\mathrm{T}} :: m\ (\mathit{Result}\mathbb{A}_{\mathrm{T}}\ m\ a)\} \\
\textbf{data}\ \mathit{Result}\mathbb{A}_{\mathrm{T}}\ m\ a \\
\qquad = \mathit{Return}\ a \\
\qquad \mid\ \mathit{OpenApp}\ \mathit{Tag} \qquad\qquad\qquad \text{-- tag} \\
\qquad\qquad\qquad (\mathit{Int} \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ \mathit{Int}) \quad \text{-- function} \\
\qquad\qquad\qquad \mathit{Int} \qquad\qquad\qquad \text{-- argument} \\
\qquad\qquad\qquad (\mathit{Int} \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ a) \quad \text{-- continuation}
\end{array}
$$

The $\mathit{Result}\mathbb{A}_{\mathrm{T}}$ value indicates what comes next after an m computation. Either the computation is done, which is denoted by the Return constructor, or an open function application comes next, denoted by the $\mathit{OpenApp}$ constructor. In particular, $\mathit{OpenApp}\ t\ g\ x\ k$ denotes the open application of g to x with tag t, followed by the continuation k that proceeds the computation with the result of the open application. Figure 12.1 shows the instances for the Monad and $\mathit{MonadTrans}$ type classes. Observe that for open applications, $\gg\!=$ extends the corresponding continuation k with operation f.

Open Applications Function $\mathit{openApp}$ creates the denotation of tagged open applications:

$$\mathit{openApp}\ t\ f\ x = \mathbb{A}_{\mathrm{T}}\ (\mathit{return}\ (\mathit{OpenApp}\ t\ f\ x\ \mathit{return}))$$

Because return is the left and right identity of $\gg\!=$, we use it as the continuation that proceeds with the result of the open application. Hence, in isolation, open applications provide a semantics-

preserving connection point for composition through $\gg=$. Using $openApp$, $\mathbb{A}_\mathrm{T}$ can be declared as an instance of the $\mathbb{A}_\mathrm{M}$ type class:

$$
\begin{array}{l}
\textbf{instance}\ Monad\ m \Rightarrow \mathbb{A}_\mathrm{M}\ (\mathbb{A}_\mathrm{T}\ m)\ \textbf{where} \\
\quad f\ \#^t\ x = openApp\ t\ f\ x
\end{array}
$$

12.2 Running $\mathbb{A}_\mathrm{T}$ Computations

We define the $run\mathbb{A}_\mathrm{T}$ interpreter function which evaluates an $\mathbb{A}_\mathrm{T}$ computation:

$$
\begin{array}{l}
run\mathbb{A}_\mathrm{T}\ aenv\ m = un\mathbb{A}_\mathrm{T}\ m \gg= go\ \textbf{where} \\
\quad go\ (Return\ r) = return\ r \\
\quad go\ (OpenApp\ t\ f\ x\ k) = \\
\qquad un\mathbb{A}_\mathrm{T}\ (weave\ f\ aenv\ (Jp\ t) \gg= \\
\qquad\quad \lambda woven_f \rightarrow woven_f\ x \gg= k) \gg= go
\end{array}
$$

This function is defined in terms of the locally-defined go function. In case of $Return\ r$ values, it simply unwraps and $return$s value r. When it encounters an open application, it creates a join point $Jp\ t$ and uses the $weave$r to apply the matching aspects deployed in $aenv$. This yields the $woven_f$ function which is applied to argument x. The result of the application is given to continuation k, whose resulting computation is evaluated recursively using go.

12.3 Aspect Weaving

The weaver is defined recursively on the aspect environment as follows:

$$
\begin{array}{l}
weave :: Monad\ m \Rightarrow (Int \rightarrow m\ Int) \rightarrow AEnv\ m \rightarrow Jp\ m\ Int\ Int \rightarrow m\ (Int \rightarrow m\ Int) \\
weave\ f\ [\,]\ _ \qquad\qquad\qquad = return\ f \\
weave\ f\ ((pc, adv) : asps)\ jp = weave\ (\textbf{if}\ pc\ jp\ \textbf{then}\ adv\ f\ \textbf{else}\ f)\ asps\ jp
\end{array}
$$

For each aspect it applies the pointcut to the join point. Then it continues weaving on the rest of the aspect environment using either $adv\ f$ if the pointcut matches, or f otherwise.

12.4 Properties of $\mathbb{A}_\mathrm{T}$

To exploit the general result of the previous section, we need to establish that $\mathbb{A}_\mathrm{T}$ is a proper aspectual monad transformer that satisfies the necessary properties described in Section 11.1.2.

Lemma 1 (Monad laws for $\mathbb{A}_\mathrm{T}$) $\mathbb{A}_\mathrm{T}$ fulfills the monad transformer laws. In addition, for any monad m, $\mathbb{A}_\mathrm{T}\ m$ fulfills the monad laws.

```
runA_T aenv (m >>=_{A_T} f)
≡ {-unfold >>=_{A_T} -}
runA_T aenv (A_T (unA_T m >>=_m λr → case r of
  Return x          → unA_T (f x)
  OpenApp t x g k → return_m (
    OpenApp t x g (λy → k y >>=_{A_T} f))))
≡ {-unfold runA_T and unA_T ∘ A_T ≡ id -}
(unA_T m >>=_m λr → case r of
  Return x          → unA_T (f x)
  OpenApp t x g k → return_m
    (OpenApp t x g (λy → k y >>=_{A_T} f))) >>=_m go
≡ {-assoc. of >>=_m + distributing go over case -}
unA_T m >>=_m λr → case r of
  Return x          → unA_T (f x) >>=_m go
  OpenApp t x g k → return_m
    (OpenApp t x g (λy → k y >>=_{A_T} f)) >>=_m go
≡ {-folding runA_T + unA_T ∘ A_T ≡ id + left id + go -}
unA_T m >>=_m λr → case r of
  Return x          → runA_T aenv (f x)
  OpenApp t x g k → unA_T (A_T (return_m
    (OpenApp t x g (λy → k y >>=_{A_T} f)))) >>=_m go
≡ {-left id of m + folding runA_T -}
unA_T m >>=_m λr → case r of
  Return x          → return x >>=_m runA_T aenv ∘ f
  OpenApp t x g k → runA_T aenv
    (A_T (return_m (OpenApp t x g (λy → k y >>=_{A_T} f))))
≡ {-folding >>=_{A_T} -}
unA_T m >>=_m λr → case r of
  Return x          → return x >>=_m runA_T aenv ∘ f
  OpenApp t x g k → runA_T aenv
    ((A_T (return_m (OpenApp t x g k) >>=_{A_T} f)))
≡ {-co-induction hypothesis -}
unA_T m >>=_m λr → case r of
  Return x          →
    return x >>=_m runA_T aenv ∘ f
  OpenApp t x g k → runA_T aenv
    (A_T (return_m (OpenApp t x g k)))
    >>=_m runA_T aenv ∘ f
≡ {-factoring runA_T aenv ∘ f from case -}
unA_T m >>=_m λr → (case r of
  Return x          → return x
  OpenApp t x g k → runA_T aenv
    (A_T (return_m (OpenApp t x g k))))
    >>=_m runA_T aenv ∘ f
≡ {-assoc. of m -}
(unA_T m >>=_m λr → case r of
  Return x          → return x
  OpenApp t x g k → runA_T aenv
    (A_T (return_m (OpenApp t x g k))))
    >>=_m runA_T aenv ∘ f
≡ {-unfold runA_T + unA_T ∘ A_T ≡ id -}
(unA_T m >>=_m λr → case r of
  Return x          → return x
  OpenApp t x g k →
    return_m (OpenApp t x g k >>=_m go))
>>=_m runA_T aenv ∘ f
≡ {-folding go -}
(unA_T m >>=_m go) >>=_m runA_T aenv ∘ f
≡ {-folding runA_T -}
runA_T aenv m >>=_m runA_T aenv ∘ f
```

Figure 12.2: Proof of the second monad morphism law for $run\mathbb{A}_T$.

Lemma 2 ($run\mathbb{A}_T$ monad morphism) For any aspect environment $aenv$, $run\mathbb{A}_T\ aenv$ is a monad morphism. Furthermore, it is also a left inverse of $lift$.

The proofs proceed by straightforward equational reasoning and co-induction on the shape of the monadic composition, and are available in Appendix A. Crucially, the proofs rely on the monad and monad transformer laws for $\mathbb{A}_T$.

Given the importance of the compositionality of weaving (which corresponds to the second law of monad morphisms), we show its proof in Figure 12.2. This law is fundamental for the formalization of Chapter 11 and for the theorem of the following chapter. The proof consists of folding and unfolding the definitions of $run\mathbb{A}_T$, its internally-defined function go, and the $\gg=$ operation of $\mathbb{A}_T$; it also uses the monad laws on m, and the identity $un\mathbb{A}_T \circ \mathbb{A}_T \equiv \mathbb{A}_T \circ un\mathbb{A}_T \equiv id$. A crucial step is the use of the co-induction hypothesis to start folding the definitions.

This chapter developed a simplified monadic model of AOP in order to connect the formal results of Chapter 11 and a simplified version of the framework presented in Part I and Part II. The main simplifications are the lack of effectful pointcuts and the static deployment of aspects. In contrast,

a significant improvement that ought to be adopted in the full-fledged framework is to expose open applications as an abstract syntax tree (similar to a free monad). Because this approach reifies open applications, we can perform arbitrary program transformations on them. How to exploit this facility is an immediate line of future work.

Chapter 13

Local Harmlessness

In Chapter 11, we have shown how the first three conditions of Theorem 2 can be met. This chapter develops local harmlessness results using the monadic AOP model of Chapter 12. We now discuss how local harmlessness of the updated aspect environment $aenv'$ with respect to the initial environment $aenv$ can be established in this setting. Concretely, we must prove that:

$$run\mathbb{A}_{\mathrm{T}}\ aenv \circ (f\ \#^{i}) \equiv \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ (f\ \#^{i})$$

We observe that the problem of reasoning about aspect interference for an isolated function woven by aspects is directly analogous to the work of MRI in the model of mixins. Therefore, we can benefit from the established results of MRI in at least two ways:

- Translate AOP programs into the setting of MRI; establish the required program equivalence in this setting, and interpret this result back into the AOP model. This approach allows us to reuse directly all the theorems proven in the MRI model.
- Lift the reasoning techniques developed in MRI to the AOP setting, to establish similar harmlessness results. This path is potentially more general and avoids a translation to MRI, but it does entail the need to re-establish all theorems proven in the MRI model in the AOP model.[1]

Here, we adopt the first approach, leaving the second one as a possible line of future work.

13.1 AOP-MRI Translation

We present a commutative correspondence diagram that gives a high-level overview of the chosen technique. In this diagram, the local harmlessness condition of Theorem 2 is represented by path (d). Instead of proving this directly, the goal is to obtain (d) by the composition of paths (a), (b) and (c).

[1] Although in Part II we adapt the use of parametricity to enforce non-interference of pointcuts, advice and base programs to the AOP model, rigorous formal results have not been established yet in the model of Chapter 3.

$$
\begin{array}{ccc}
f_{MRI} + mix & \overset{(b)}{\underset{\equiv}{\longrightarrow}} & \pi(f_{MRI} + mix') \\
(a) \; \cong & & \cong \; (c) \\
f_{AOP} + aenv & \overset{\equiv}{\underset{(d)}{\longrightarrow}} & \pi(f_{AOP} + aenv')
\end{array}
$$

Starting from an AOP system composed by function f_{AOP} and aspect environment $aenv$, step (a) involves finding a function f_{MRI} and a mixin mix, such that their composition is equivalent to this initial system. In the same way, step (c) requires to find a mixin mix' equivalent to aspect environment $aenv'$. Given this, we can reuse the reasoning techniques and established results of MRI to determine the equivalence of step (b) between f_{MRI} composed with mix and the projection of f_{MRI} composed with mix'.

A drawback of this approach is that it is not known how to perform steps (a) and (c) in a general manner, because there are AOP programs that cannot be expressed using mixins, as illustrated later. Still, we can prove that a connection exists for a wide family of functions and aspect environments (Theorem 3 below).

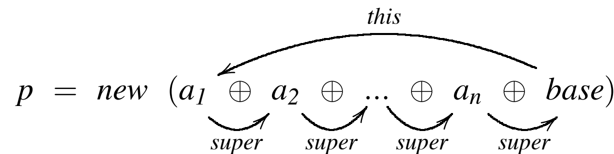
We now briefly summarize the MRI model and prove a theorem connecting MRI to AOP. Then, using the Fibonacci function as a concrete example, we show how to prove that the logging and memoization aspects from Figure 11.1 are locally harmless.

13.2 Background: the MRI Framework

MRI models inheritance by the composition of mixins through open recursion. This inheritance model is defined as (Oliveira et al., 2012):

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Open}\ s = s \to s \\
\mathit{new} :: \mathit{Open}\ s \to s \\
\mathit{new}\ a = \mathit{fix}\ (\lambda f \to a\ f) \\
(\oplus) :: \mathit{Open}\ s \to \mathit{Open}\ s \to \mathit{Open}\ s \\
a_1 \oplus a_2 = \lambda \mathit{super} \to a_1\ (a_2\ \mathit{super})
\end{array}
$$

The type $\mathit{Open}\ s$ represents an open component of type s. new is a fixpoint combinator that closes, or instantiates, an open component that is potentially extended. Finally, the $\oplus$ operator defines component composition. The following diagram (taken from (Oliveira et al., 2012)) illustrates the inheritance model:



To create a component $\oplus$ instantiates $super$ references for every extended component, and new instantiates the self-reference $this$.

MRI formally captures the notion of *harmlessness* that has been used in Part III of this work. Given a mixin mix and base component bse, then mix is harmless if:

$$\pi\ (new\ (mix \oplus bse)) \equiv run\mathbb{I} \circ new\ bse$$

for some projection π. Here $run\mathbb{I}$ is the projection of the identity monad, which has no computational effect.

MRI provides two harmless mixin theorems (Oliveira et al., 2012). Using these theorems it is formally proven that logging is harmless for any arbitrary function. It is also proven that memoization is harmless when applied to the Fibonacci function. In the examples of next section we detail the specific techniques used in these proofs.

13.3 Connecting MRI to AOP

There is a direct connection between an advice and a mixin, as witnessed by the types of these entities: both type synonyms $Advice\ m\ a\ b$ and $Open\ (a \to m\ b)$ denote the same type $(a \to m\ b) \to (a \to m\ b)$. This reveals that any MRI mixin can be used as an advice. However the converse is not generally true if an advice performs open applications. For instance an aspect could trigger infinite regression by matching join points emitted on its own advice. However, if an advice uses a type class constraint that does not entail $\mathbb{A}_{\mathrm{M}}$ (which means that it cannot perform any open application), this cannot happen.

To connect a base function f_{AOP} with an open recursive equivalent function, we need a stronger constraint. Namely, we ask that f_{AOP} is equivalent to the fixpoint of an open recursion f_{MRI} (that does not make use of open application) in the following way:

$$f_{AOP} \equiv fix\ (\lambda f \to f_{MRI}\ (f\ \#^t))$$

Putting these together, we can state a general theorem that relates MRI to AOP and eases considerably the establishment of steps (a) and (c) of the correspondence diagram.

Definition 9 (AOP agnostic function) A function

$$f :: \forall m.C\ m \Rightarrow (A \to m\ B) \to (A \to m\ B)$$

is *AOP-agnostic* iff C is a type class constraint that entails $Monad$ but not $\mathbb{A}_{\mathrm{M}}$. This means that the function does not emit join points.

Theorem 3 Given AOP-agnostic functions

$$f_{MRI} :: \forall m.C\ \ m \Rightarrow (A \to m\ B) \to (A \to m\ B)$$
$$adv_i\ :: \forall m.D_i\ m \Rightarrow (A \to m\ B) \to (A \to m\ B)$$

and given an aspect environment

$$aenv = [(pcTag\ t_1, adv_1), \ldots, (pcTag\ t_n, adv_n)]$$

we have that

$$run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{AOP}\ \#^t \equiv new\ (adv'_k \oplus \ldots \oplus adv'_1 \oplus f_{MRI})$$

where f_{AOP} is such that

$$f_{AOP} \equiv fix\ (\lambda f \rightarrow f_{MRI}\ (f\ \#^t))$$

and $[adv'_1, \ldots, adv'_k] = [adv_i \mid pcTag\ t_i\ (Jp\ t) \equiv True]$ is the list of all advices in $aenv$ whose $pcTag\ t_i$ pointcut matches $\#^t$ applications.

PROOF. The proof proceeds by equational reasoning and co-induction:

$$\begin{aligned}
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{AOP}\ \#^t \\
\equiv\ & \{\text{-definition of } f_{AOP}\ \text{-}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ fix\ (\lambda f \rightarrow f_{MRI}\ (f\ \#^t))\ \#^t \\
\equiv\ & \{\text{-unfolding of the fixpoint -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{MRI}\ (f_{AOP}\ \#^t)\ \#^t \\
\equiv\ & \{\text{-compositionality of weaving -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{MRI}\ (run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{AOP}\ \#^t)\ \#^t \\
\equiv\ & \{\text{-co-induction hypothesis -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{MRI}\ (new\ (adv'_k \oplus \ldots \oplus adv'_1 \oplus f_{MRI}))\ \#^t \\
\equiv\ & \{\text{-weaving -}\} \\
& adv'_k \circ \ldots \circ adv'_1 \circ f_{MRI}\ (new\ (adv'_k \oplus \ldots \oplus adv'_1 \oplus f_{MRI})) \\
\equiv\ & \{\text{-definition of } \oplus\ \text{-}\} \\
& adv'_k \oplus \ldots \oplus adv'_1 \oplus f_{MRI}\ (new\ (adv'_k \oplus \ldots \oplus adv'_1 \oplus f_{MRI})) \\
\equiv\ & \{\text{-folding the } new \text{ fixpoint -}\} \\
& new\ (adv'_k \oplus \ldots \oplus adv'_1 \oplus f_{MRI})
\end{aligned}$$

□

The same proof can be made for any model of AOP as described in Section 11.1; one just has to accommodate the proof according to the concrete way (in particular the ordering) in which aspects are woven.

Example that cannot use Theorem 3 We now present an aspect-oriented implementation of the Fibonacci function that cannot be translated into MRI by Theorem 3. In this example, taken from (Aldrich, 2005), the function is split into a base case that simply returns 1, and an advice that handles the recursive calls. The composed function $plainFib$ combines the base program and advice to provide the regular unoptimized version of Fibonacci.

$$\begin{aligned}
& plainFib\ n = run\mathbb{A}_{\mathrm{T}}\ [(pcTag\ t, fibAdv)]\ (fibBase\ \#^t\ n) \\
& fibBase\ _ = return\ 1
\end{aligned}$$

$$
\begin{array}{ll}
\mathit{fib}_{AOP} :: \mathbb{A}_{\mathrm{M}}\ m \Rightarrow \mathit{Int} \to m\ \mathit{Int} & \mathit{fib}_{MRI} :: \mathit{Monad}\ m \Rightarrow \mathit{Open}\ (\mathit{Int} \to m\ \mathit{Int}) \\
\mathit{fib}_{AOP}\ n = \mathbf{case}\ n\ \mathbf{of} & \mathit{fib}_{MRI}\ \mathit{this}\ n = \mathbf{case}\ n\ \mathbf{of} \\
\quad 0 \to \mathit{return}\ 1 & \quad 0 \to \mathit{return}\ 1 \\
\quad 1 \to \mathit{return}\ 1 & \quad 1 \to \mathit{return}\ 1 \\
\quad _ \to \mathbf{do}\ y \leftarrow \mathit{fib}_{AOP}\ \#^t\ (n-1) & \quad _ \to \mathbf{do}\ y \leftarrow \mathit{this}\ (n-1) \\
\qquad\qquad x \leftarrow \mathit{fib}_{AOP}\ \#^t\ (n-2) & \qquad\qquad x \leftarrow \mathit{this}\ (n-2) \\
\qquad\qquad \mathit{return}\ (x+y) & \qquad\qquad \mathit{return}\ (x+y) \\
\mathit{plainFib}_{AOP}\ :: \mathit{Monad}\ m \Rightarrow \mathit{Int} \to m\ \mathit{Int} & \mathit{plainFib}_{MRI}\ :: \mathit{Monad}\ m \Rightarrow \mathit{Int} \to m\ \mathit{Int} \\
\mathit{plainFib}_{AOP}\ = \mathit{run}\mathbb{A}_{\mathrm{T}}\ [\,] \circ \mathit{fib}_{AOP}\ \#^t & \mathit{plainFib}_{MRI}\ = \mathit{new}\ \mathit{fib}_{MRI} \\
\mathit{logFib}_{AOP}\ :: \mathit{Monad}\ m \Rightarrow & \mathit{logFib}_{MRI}\ :: \mathit{Monad}\ m \Rightarrow \\
\quad \mathit{Int} \to \mathbb{W}_{\mathrm{T}}\ \mathit{String}\ m\ \mathit{Int} & \quad \mathit{Int} \to \mathbb{W}_{\mathrm{T}}\ \mathit{String}\ m\ \mathit{Int} \\
\mathit{logFib}_{AOP}\ = \mathit{run}\mathbb{A}_{\mathrm{T}}\ [(\mathit{pcTag}\ t, \mathit{log}')] \circ \mathit{fib}_{AOP}\ \#^t & \mathit{logFib}_{MRI}\ = \mathit{new} \circ (\mathit{log} \oplus \mathit{fib}_{MRI}) \\
\mathit{memoFib}_{AOP} :: \mathit{Monad}\ m \Rightarrow & \mathit{memoFib}_{MRI} :: \mathit{Monad}\ m \Rightarrow \\
\quad \mathit{Int} \to \mathbb{S}_{\mathrm{T}}\ (\mathit{Map}\ \mathit{Int}\ \mathit{Int})\ m\ \mathit{Int} & \quad \mathit{Int} \to \mathbb{S}_{\mathrm{T}}\ (\mathit{Map}\ \mathit{Int}\ \mathit{Int})\ m\ \mathit{Int} \\
\mathit{memoFib}_{AOP} = \mathit{run}\mathbb{A}_{\mathrm{T}}\ [(\mathit{pcTag}\ t, \mathit{memo})] \circ \mathit{fib}_{AOP}\ \#^t & \mathit{memoFib}_{MRI} = \mathit{new}\ (\mathit{memo} \oplus \mathit{fib}_{MRI})
\end{array}
$$

Figure 13.1: Fibonacci function. Left: in the simple pointcut/advice model of Chapter 12. Right: in the MRI setting (taken from (Oliveira et al., 2012))

$$
\begin{array}{l}
\mathit{fibAdv}\ \mathit{proceed}\ n = \\
\quad \mathbf{if}\ (n \leqslant 2)\ \mathbf{then}\ \mathit{proceed}\ n \\
\qquad\qquad\quad \mathbf{else\ do}\ f_1 \leftarrow \mathit{fibBase}\ \#^t\ (n-1) \\
\qquad\qquad\qquad\qquad f_2 \leftarrow \mathit{fibBase}\ \#^t\ (n-2) \\
\qquad\qquad\qquad\qquad \mathit{return}\ (f_1 + f_2)
\end{array}
$$

We cannot apply Theorem 3 because of the type of fibAdv. Since fibAdv performs open applications of $\mathit{fibBase}$, its type necessarily contains a type class constraint that entails $\mathbb{A}_{\mathrm{M}}$; thus violating the initial condition of Theorem 3. In fact, it does not seem possible to define fibAdv using mixins, because the full aspect environment is woven upon each open application, whereas mixins can only execute the next component using super.

Applying the theorem We now present an example, using the Fibonacci function as a concrete value for f, on how to follow the steps of the correspondence diagram to prove the harmlessness of the logging and memoization advices of Figure 11.1. We consider the starting environment aenv to be empty; and illustrate the case of adding each aspect individually. Figure 13.1 presents the Fibonacci function in the AOP and MRI models, along with their plain, logged and memoized versions.

13.4 Harmlessness of Logging

The local harmlessness of log applied to fib_{AOP} corresponds to the following lemma:

Lemma 3 $\mathit{plainFib}_{AOP} \equiv \pi_W \circ \mathit{logFib}_{AOP}$

PROOF. Following the commutative correspondence diagram, by composition of Lemma 4, Lemma 5 and Lemma 6. □

Step (a) We must translate $plainFib_{AOP}$ into MRI. We choose $plainFib_{MRI}$ as its translation, hence we must prove the following:

Lemma 4 $plainFib_{AOP} \equiv plainFib_{MRI}$

The proof is direct consequence of Theorem 3, using the equality

$$fib_{AOP} \equiv fix\ (\lambda f \rightarrow fib_{MRI}\ (f\ \#^t))$$

that can be proven by equational reasoning and induction on the integer argument.

Step (b) For the second step we need to prove:

Lemma 5 $plainFib_{MRI} \equiv \pi_W \circ logFib_{MRI}$

Here we benefit from the results of MRI. In MRI the local harmlessness of logging is proven for any arbitrary component, like fib_{MRI}; hence it holds for this particular case (Oliveira et al., 2012).

In fact the general harmlessness of logging is an application of the *harmless mixin theorem* of MRI (Oliveira et al., 2012). This theorem is proven using: (i) parametricity to ensure that the base component cannot access the effects used by the mixin; (ii) a mixin combinator to guarantee that $super$ is called exactly once, and that the arguments and return values are not modified; and (iii) the algebraic laws for monadic effects. Consequently, any mixin that satisfies this theorem is also harmless for functions that are AOP-agnostic (Definition 9).

Step (c) Finally, we prove the equivalence between $logFib_{AOP}$ and $logFib_{MRI}$:

Lemma 6 $\pi_W \circ logFib_{AOP} \equiv \pi_W \circ logFib_{MRI}$

Again, this is a direct consequence of Theorem 3.

13.5 Harmlessness of Memoization

Proving the harmlessness of memoization involves the same steps as that of logging. In this case we greatly benefit from the established results of MRI, because proving step (b) is rather complex. The issue is that, conversely to logging, memoization is not harmless in general; hence this property must be proven for each particular function. The main difficulty of such a proof is to show that the function maintains an invariant on the memoization table: namely, that the stored values actually correspond to the results of the original function. In the work of Oliveira et al. (2012) this is

proven for fib_{MRI}, developing a long equational reasoning proof—the Coq proof assistant is used to manage the complexity of the proof.

It is in complex situations like this that the interest of following the steps of the AOP-MRI correspondence diagram is justified. In addition, we can benefit from new results about harmlessness of specific mixins.

This chapter was devoted to show how to fulfill the local harmlessness requirement of Theorem 2 for the particular case of our simple monadic model. In particular we sketched a general approach for a subset of advices that can be expressed in the mixin-based model of MRI. Nevertheless, recall that the actual fact that we needed to prove is that:

$$run\mathbb{A}_{\mathrm{T}}\ aenv \circ (f\ \#^{i}) \equiv \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ (f\ \#^{i})$$

Then, depending on the specific environments $aenv$ and $aenv'$ involved, and of course depending on the specific function f, the difficulty of proving this requirement may vary greatly. Indeed, we envision the need to use a proof assistant like Coq in order to ease the task. Finally, we believe that establishing local harmlessness in our full-fledged model is a very interesting line of future work. Indeed, a first challenge to address is how to address the potential interference of effectful pointcuts with regard to the MRI notion of harmlessness.

Chapter 14

Related Work, Part III

We have extensively related our work to EffectiveAdvice (Oliveira et al., 2010) (Chapter 7 and Chapter 8) and its successor, MRI (Oliveira et al., 2012) (Chapter 10 to Chapter 13). Indeed, the work developed in this thesis is built upon these pieces of work: first, the monadic embedding of aspects in Haskell developed in Chapter 3 is a practical programming system that can be seen as extending EffectiveAdvice with quantification, but it does not describe how to do formal reasoning. The model can also be regarded as an extension of Open Modules (Aldrich, 2005) that incorporates computational effects modeled using monads. In addition, the work presented in Part III was motivated by the desire to bring the reasoning power of MRI to aspect-oriented programming with quantification.

In the AOP literature the issues of modular reasoning and aspect interference are closely related and in some cases intertwined; however we choose to present the relevant work in two separate sections, according to our particular (and subjective) point of view. We start reviewing the approaches to modular reasoning and AOP (Section 14.1), as well as the proposals to reason about aspect interference (Section 14.2).

14.1 Approaches to Modular Reasoning in AOP

In the AOP literature we identify two lines of work that mainly address the conflict between full-blown aspect quantification and modular reasoning. On the one hand, approaches like Open Modules (Aldrich, 2005) try to *protect* software entities from advising. On the other hand, approaches like execution levels (Tanter, 2010) try to limit the *scope* of aspects, that is, the set of join points exposed to pointcuts of applicable aspects.

14.1.1 Protecting Modules from Aspects

We now describe Open Modules (Aldrich, 2005), aspect-aware interfaces (Kiczales and Mezini, 2005), crosscutting interfaces (Griswold et al., 2006) and join point interfaces (**?**).

Open Modules

Open Modules (OM) (Aldrich, 2005) is a module system that is both *open* to extension with advice and also amenable to modular reasoning. As we already described this proposal in Section 7.1, we now describe in more detail the relation between OM and our work.

Our model significantly extends OM in two ways. First, we feature first-class polymorphic pointcuts, aspects and advices; whereas only pointcuts are considered first-class in OM. And second, we introduce explicit reasoning about effects by using monads. However, although our pointcut language is more expressive, we can currently retain the modular reasoning of OM only when restricting the pointcut language to one single pointcut, namely $pcCall$, just like in Aldrich's proposal.

The module system of OM is based on that of ML. Consequently, there are two fundamental constructs in the OM module system: *signatures* and *structures*. A signature specifies a particular interface while a structure can be seen as a particular implementation of a signature. In contrast, we reuse Haskell's module system to directly export first-class pointcuts. In OM pointcuts and advices refer to specific *labels*, which correspond to top-level declarations inside a module structure. To hide internal calls from external advice, OM features a *sealing* operation that works both at the type level and at the level of the operational semantics of the language. At the type level, sealing a module means that only the members in the ascribed signature are visible. At the operational level, sealing a module hides the internal calls within a module by creating a set of internal labels, which are used in the implementation of the module; and a set of external labels, which are used in the external signature of the module. Because pointcuts and advice are predicated over labels instead of being directly defined over functions, sealing a module ensures that external advices will always refer to external labels and similarly that internal advices will refer to internal labels.

In our full-fledged model we hide internal applications following a similar approach, but exploiting our notion of function identity rather than relying on labels. Given an internal function f, which is not exported by its module, it is not possible for an external aspect to create a $pcCall\ f$ pointcut simply because f is not available (however it is possible to use a generic $pcType$ pointcut, which would match open applications of f, hence our restriction to $pcCall$). To distinguish between the internal and external identity of a function we can simply define and export a new function $g = f$, that is computationally equivalent but has a different identity. If no pointcut is exported then we have encoded the semantics of OM: internal advice can affect f, external advice can only affect external calls to g, and the module can export a pointcut to provide access to internal calls as desired. Although both approaches are similar, Open Modules presents a special-purpose language and type system, whereas we do not extend the module nor the type system of Haskell. Interestingly, we can straightforwardly define a sealing transformation in the AOP model of Chapter 12 because open applications are reified in the monadic structure. This would allow us to preserve the guarantees of OM with quantification beyond only $pcCal$.

Finally, we observe that proving the equivalence of two modules in the formal setting of OM relies on "global" reasoning with unrestricted quantification. Our formal framework (Chapter 11) could be used to enhance that part of the reasoning.

Aspect-aware and crosscutting interfaces

Kiczales and Mezini (2005) also recognize that the crosscutting nature of aspects hampers modular reasoning. Moreover, they argue that strictly modular reasoning about programs written in the presence of quantification is not feasible, and introduce a notion of aspect-aware interfaces that rely on a global reasoning step to infer precise dependencies. Once this step is performed, extended modular reasoning is available for a particular deployment of aspects in a system.

Similar to aspect-aware interfaces, Griswold et al. (2006) propose crosscutting interfaces (XPIs) as a design pattern to improve modularity in the design of aspect-oriented programs in languages like AspectJ. Unlike aspect-aware interfaces, XPIs are not inferred from a particular system deployment but are rather explicitly designed and, as along as the pattern is followed, should not require global analysis.

The idea of XPIs raised from an experiment that used AOP to improve the design on a large Java system. Among their findings, the authors argue that it is difficult to enforce invariants on aspects due to the obliviousness property of AspectJ because "[...] apparently innocuous changes or extensions to the code base could then change the matched join points, violating assumptions the aspects made" (Griswold et al., 2006).

The essential idea of XPIs is to serve as an intermediate layer—and more specifically as a contract— between the base code that is subject to aspects, and the concrete implementation of advices. In particular, an XPI is a regular aspect that only defines pointcuts. These pointcuts have a public interface and represent abstract sets of join points. The specific join points matched by the pointcuts must be adjusted during the evolution of the software. Additionally, the XPI describes a semantic specification by indicating pre- and post-conditions that advices must satisfy. Then, in order to apply advice to a system aspects must not directly reference the base code—instead, aspects must refer to the pointcuts defined in a concrete XPI.

Both Kiczales and Mezini (2005) as well as Griswold et al. (2006) evaluate their work by comparing two implementations of a simple figure editor. One implementation uses the traditional AspectJ approach while the other is designed using aspect-aware interfaces or XPIs. In both cases the authors considered certain evolution scenarios (different in each case) and concluded that their approach helped to develop aspect-oriented software that is more robust an extensible.

In their work, Kiczales and Mezini advocate for compositional, or rather incremental, reasoning as we do in our work. They argue that this is reflected in the implementation of AspectJ (at least as of version 1.2 (Kiczales and Mezini, 2005)), where the weaver process constructs a *weaving plan* with information similar to aspect-aware interfaces to avoid recompiling unmodified code. On the other hand, the main drawback of XPIs is that it lacks any language-enforced mechanism. As a design pattern, it relies on the discipline of developers and therefore cannot provide any strong guarantees about modularity. Finally, neither of these proposals has been used to perform formal (modular) reasoning.

Join point interfaces

? propose Join Point Interfaces (JPIs) as a means to separate base code and aspect code while enabling separate development and modular reasoning. In their work, join point interfaces are an

additional layer of abstraction between base code and aspects, defined syntactically as a method signature with return type, a formal-parameter list and a list of thrown exceptions. Advice now is deployed on join point interfaces instead of pointcuts that directly relate to the base code. Classes need to explicitly *exhibit* a join point interface declaring its signature and the corresponding pointcut. Join points can be emitted implicitly, as in plain AspectJ, or explicitly using closure join points (Bodden, 2011).

Bodden and colleagues implemented JPIs as an extension to AspectJ. The following example adapted from (**?**) illustrates the concepts described above:

```
jpi void CheckingOut(double price, Customer cust);

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

  void checkOut(final Item item, double price, final int amount, final Customer cus) {
     ...
  }
}
```

A join point interface is declared using the jpi keyword followed by a method-like signature, including return type and the list of checked exceptions. The example defines the CheckingOut interface that semantically represents that an item is about to be bought, in the context of a e-commerce system.

The ShoppingSession class implements part of the business logic, in particular the checkOut method. The exhibits declaration binds a regular AspectJ pointcut to the join point interface. Now consider an aspect to give customers a 5% discount on their birthday. This is implemented as:

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

where the advice uses exactly the same signature of the join point interface.

Finally, the implementation of JPIs features static and modular type checking, which supports modular reasoning and guarantees the absence of weave-time errors. In addition, they support the notion of *generic advice* (using Java generics) as well as providing a mechanism to control global quantification.

Connection with this work The proposals described in this section advocate for the definition of an extended module (or class) interface to tame the power of unrestricted quantification through

pointcuts. In our work we have followed the approach of Open Modules, but we consider that our join point model could reflect the other approaches without much significant challenges. This discussion also highlights part of the debate about obliviousness in the AOP community. Both aspect-aware interfaces and XPIs keep obliviousness as a core concept in their implementation, while trying to work around some of the issues caused by it. In contrast, Open Modules opts for a restricted design where obliviousness is forbidden between module boundaries. Finally, JPIs are designed as a pragmatic complement to AspectJ. Both kinds of aspects, plain AspectJ aspects with full obliviousness and JPI-bound aspects, can co-exist in the same system. Our work does not feature obliviousness. Open function applications are explicitly declared using the # operator; although we argue that some form of obliviousness can be achieved by using some form of preprocessing (*e.g.* a macro system or a source-to-source translation from other language).

14.1.2 Limiting the Scope of Aspects

Regarding this line of work we describe the proposals of statically and dynamically scoped aspects (Dutchyn et al., 2006), scoping strategies (Tanter, 2008), and the *topological* approach of execution levels (Tanter, 2010).

Statically and dynamically scoped aspects

AspectScheme (Dutchyn et al., 2006) is a higher-order functional aspect language that features first-class aspects. In contrast to first-order aspect languages like AspectJ, aspects in AspectScheme are *dynamically deployed*. This poses the challenge of determining the scope of aspects, in other words, when will aspects be active or applicable to a particular join point.

AspectScheme addresses this question by allowing aspects to be either dynamically- or statically-scoped. The `(fluid-around pc adv body)` expression deploys a dynamically-scoped aspect, with pointcut `pc` and advice `adv`, that applies to any join point emitted during the dynamic extent of the evaluation of `body`. Conversely, the `(around pc adv body)` expression deploys a statically-scoped aspect. This kind of aspect is bound to applications that are lexically explicit in the `body`.[1] As statically-scoped aspects may be unfamiliar to some readers, we will explain it by-example, using the original examples presented by Dutchyn et al. (2006). For instance, consider:

```
(around (call open-file) trace-advice)
  (open-file "vancouver"))
```

In this case, the aspect will apply because the application of `open-file` is lexically bound in the body. The same will happen in this program:

```
(let ([static-traced-open (around (call open-file) trace-advice)
                             (lambda (f) (open-file f))])
    (static-traced-open "vancouver"))
```

for the same reason. However, the aspect will not apply in the following case:

[1] This is similar to AspectJ's `within` pointcut designator.

```
(let ([apply-to-vancouver (lambda (f) (f "vancouver"))])
  (around (call open-file) trace-advice
    (apply-to-vancouver open-file)))
```

because the application of `open-file` is in the dynamic extent of `apply-to-vancouver`, instead of being lexically bound. It is important to observe that aspects bound lexically in a procedure will apply in all future applications of that function—even after the evaluation of the `around` expression has finished—in a way similar to per-object deployment in AspectJ.

Scoping strategies

Tanter (2008) proposed the *scoping strategies* mechanism as a generalization to control the scope of dynamically-deployed aspects. Essentially, this consists in considering and controlling three orthogonal dimensions that determine the scope of an aspect:

- c: *call stack propagation*, to control whether the aspect sees join points produced beyond the activation of a new stack frame.
- d: *delayed evaluation propagation*, to control whether the aspect is captured in created procedural values, such as functions and objects, in order to see the join points of their future evaluations.
- f: *local join point filtering*, to refine in a deployment-local manner the join points seen by the aspect.

A deployment strategy $\delta(c, \mathrm{d}, f)$ is defined in terms of these three dimensions. Tanter uses AspectJ to illustrate the need for flexible scoping of aspects, and implements a prototype interpreter that extends AspectScheme. In this prototype c, d and f are regular first-class functions, therefore allowing total flexibility on the scoping conditions for each dimension.

In a nutshell, dynamic deployment using scoping strategies is performed using a single syntactical construct $deploy(a, \delta(c, \mathrm{d}, f), \mathrm{e})$. Evaluating this expression deploys the aspect (or set of aspects) a into expression e, using the scoping determined by strategy $\delta(c, \mathrm{d}, f)$. Scoping strategies go beyond dynamic and lexical scoping, as proposed in AspectScheme, and can express a full continuum between these two extremes.

Topological Scoping of Aspects

In subsequent work Tanter addresses the issue of infinite regression of aspects. Loosely speaking, this problem happens when an aspect advises one of its own join points, triggering an infinite loop. As a solution Tanter proposed execution levels, which we briefly described in Section 9.4. Later, Tanter et al. (2012) proposed the more general *membrane model* for AOP. These proposals add a *topological dimension* to control the scope of aspects, because the propagation of join points now flows between nodes in a graph. In execution levels the graph follows a strictly linear ordering, whereas in the membrane model the graph can be arbitrary. The fundamental idea behind topological scoping is that the place of execution (*i.e.* the level or the membrane) in which a join point is emitted is a dynamically-scoped property of execution, instead of being bound to a static code entity.

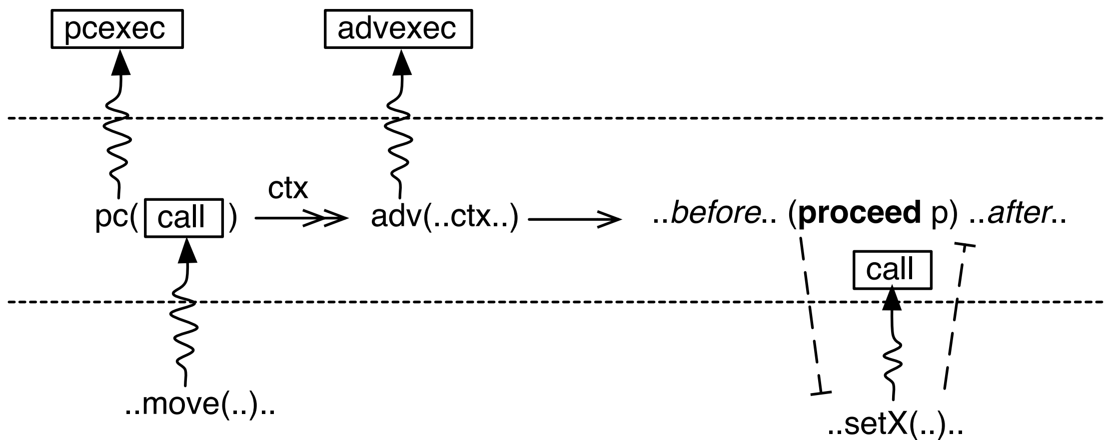


Figure 14.1: Execution levels in action: pointcut and advice are evaluated at level 1, `proceed` goes back to level 0 (from (Tanter, 2010))

Execution levels A program computation is structured into *levels*. Aspects deployed at level n only observe join points from level $n-1$. In turn, the computation of an aspect (*i.e.* the evaluation of its pointcuts and advice) is reified as join points visible at level $n+1$. This way aspects cannot match their own join points, thus avoiding this kind of infinite regression. By default, the execution level changes only as a consequence of the weaving process. Finally, the semantics of execution levels guarantee that evaluating the original computation, which corresponds to the last `proceed` call in an advice chain, is performed at its original level.

Figure 14.1 describes the default behavior of execution levels in the context of a figure editor in AspectJ. First, the base code evaluates the method `move`, which emits a call join point at level 1. An aspect deployed at level 1 evaluates its pointcut against the join point. The join points emitted by the pointcut are depicted as `pcexec`, and are emitted at level 2. After the pointcut accepts the join point, the advice of the aspect is evaluated. Potentially, the pointcut exposes some context `ctx` to the advice. In turn, advice evaluation also emits join points, denoted as `advexec`, at level 2. Finally, when the advice calls the original computation through `proceed`, the level is shifted-down to that of the original computation computation, namely, to level 0. Join points generated during the evaluation of the original computation, such as a call to `setX`, are emitted at level 1.

The default semantics of execution levels avoid aspect regression but are inflexible. In some cases it may be required to expose advice execution to aspects that observe base level computation. To provide flexibility to developers, Tanter proposed explicit level-shifting operators: **up** and **down**. Shifting an arbitrary expression using **up** or **down** moves its computation one level above or below, affecting the visibility of its join points. With these operators programmers can specify the level at which computation is performed, according to specific needs. A refined proposal including a formal proof of how execution levels avoid a certain class of infinite regression has been recently published (Tanter et al., 2014).

Membranes for AOP As a generalization of execution levels, Tanter et al. (2012) proposed the model of programmable membranes for AOP (briefly described in Section 9.4). The idea of the membrane model for AOP was born from the need to generalize execution levels to different

topologies. Instead of a linear or tower-like structure like in execution levels, the membrane model features a *membrane topology*, which is an arbitrary graph with membranes as vertices and an advising relation as edges.

Aspects are registered in a given membrane, and their computation (pointcuts, advice) happens inside that membrane, and is only visible to the advising membranes. We recently developed a prototype implementation of membranes (Figueroa et al., 2013) in order to informally assess the impact of the model with respect to aspect interference. A simple model of membranes has also been included in the PHANtom language (an AOP language for Pharo Smalltalk) (Fabry and Galdames, 2014). Unlike execution levels, the design and development of the membrane model has not stabilized yet, and is a potential direction for future work.

Connection with this work Regarding the scope of aspects in our full-fledged model, we have presented dynamic deployment at the top level only. Deployed aspects behave like dynamically-scoped aspects, except that they extend to the whole evaluation of the program rather than to the scope of a particular expression. Because our framework is extensible with respect to aspect (un)deployment, we can easily implement the scoping mechanisms described in this section. Indeed, we have already shown in Section 9.4 how to modularly implement the semantics of execution levels by defining a custom monad transformer. In a related publication (Figueroa et al., 2013) we have also implemented the semantics of the membrane model. Therefore, the modular implementation of these scoping mechanisms serve as evidence in favor of our claims regarding the extensibility of our framework as well as its use as a tool for experimenting with novel aspect semantics.

Regarding our model for compositional reasoning, we feature only static deployment of aspects because otherwise weaving is not compositional (Section 11.1.2). In order to support dynamic deployment we would need to reason statically about the static fragments between deployments. We believe this approach will allow us to reason compositionally regardless of the deployment mechanism in use.

14.2 Reasoning about Interference in AOP

In addition the work described above, there is a vast literature which specifically address interference analysis in the setting of AOP. Here, we only discuss the most directly related work; an extensive and recent review of the area, which also covers reasoning techniques in functional, object-oriented, and feature-oriented programming can be found in (Oliveira et al., 2012).

Interference of stateful aspects

A *stateful aspect* (Douence et al., 2002) is an aspect that is defined in terms of a sequence of join points during program execution, instead of a single join point. The aspect itself is defined as a finite-state machine, whose state changes upon matching the next join point of its corresponding sequence. Because stateful aspects can evolve according to the whole story of execution, they are specially well-suited for tasks such as a security or error detection.

Regarding the interaction between stateful aspects, Douence et al. (2004) present a formal approach to establish that two stateful aspects commute, and in that sense do not interfere. Their work, specific to the state effect, is also based on equational reasoning, but no theorem is stated. Instead an algorithm checks the cases where aspects are independent, leaving conflicts to be resolved by the programmer. Conflicts are resolved using specific composition or adapter operators.

Observers and Assistants

A well-known situation of non-interference has been captured by Clifton and Leavens (2002) as *observers*. Similar to augmentation advice, observers do not change the behavior of a module. On the other hand, *assistants* are aspects that are explicitly allowed to interfere with the specification of a module. Assistants must be explicitly accepted by a module through an `accepts` declaration. This declaration must refer to an aspect using its fully qualified name. In our work, assistants can be related to protected pointcuts. The difference is that protected pointcuts accept any advice that conforms to their restrictions, whereas with assistants we need to fully known in advance the identity of the aspect that is accepted.

Later, Clifton, Leavens, and Noble (2007) proposed an extension of AspectJ with annotations to control two forms of interference on control and heap effects. The correctness of annotations is checked using a type-and-effect system. To achieve a similar separation of the monad stack we have used the parametricity-based techniques of EffectiveAdvice, as well as monad views (Chapter 8).

Intraprocedural analyses

Rinard et al. (2004) present a classification for different kinds of advice, depending on their control-flow and data-flow properties. They also present automatic program analyses for AspectJ that report about the interactions between aspects and a system. However no proofs are given that the analyses are actually correct.

As we described before (Section 7.4), the kinds of advice based on their control flow behavior are: augmentation, replacement, narrowing and combination advice. Regarding computational effects, the analyses consider whether aspects and base code share access to a field. Besides writing and reading to a field, which corresponds to the state effect, the authors use the notion of *abstract fields* to denote other actions that are externally visible. Quoting from (Rinard et al., 2004), the kinds of effectful interactions between advice and method are:

- **Orthogonal**: The advice and method access disjoint fields.
- **Independent**: Neither the advice nor the method may write a field that the other may read or write.
- **Observation**: The advice may read one or more fields that the method may write but they are otherwise independent.
- **Actuation**: The advice may write one or more fields that the method may read but they are otherwise independent.
- **Interference**: The advice and method may write to the same field.

In our we have already related to the classification with respect to control flow behavior, following EffectiveAdvice (Oliveira et al., 2010). In our work, these characterizations can be determined by the granularity of the type classes related to monadic effects. For example, the standard $\mathbb{S}_{\mathrm{M}}$

class defines both the get and put method, which makes it difficult to reason about shared access. Indeed, Oliveira et al. (2012) address this issue by redefining $\mathbb{S}_\mathrm{M}$ in terms of two separate classes, one for each operation.

The main difference between our approach and that of Rinard and colleagues is that we aim to statically enforce a certain kind of interaction, while their purpose is to identify and inform developers about the behavior of an already composed system. Finally, an informal idea of compositional reasoning is considered in this work. By knowing the kind of interaction of a specific aspect added to a system developers can focus their reasoning only into the potentially problematic interactions caused by the new aspect.

Harmless advice

Dantas and Walker (2006) define an object calculus extended with *harmless advice*. Unlike regular aspect-oriented advice, harmless advice can only change the termination behavior of a program and perform I/O. Therefore, harmless advice presents a weak non-interference property: either it changes the termination behavior, or the final result is not affected by advice. They argue that under the restrictions of harmlessness, aspect-oriented advice can still perform many of its characteristic applications: profiling, invariant checking and program monitoring. As a case study, the authors ported a set of security policies originally implemented for Java and found that only one policy, which limited the sending rate of data in the network, was not harmless.

Theoretically, Dantas and Walker developed a typed lambda calculus with explicitly labeled control-flow points and with advice; following the calculus presented by Walker et al. (2003). Non-interference is enforced by a type and effect system based on a lattice of *protection domains*. Similar to systems for information flow, the idea is that if $p < q$ in the protection domain lattice, then advice in domain p cannot interfere with computation in domain q. In our we used the notion of non-interference defined in MRI (Oliveira et al., 2012), which subsumes the notion of Dantas and Walker.

Translucid contracts

Ptolemy (Rajan and Leavens, 2008) is an object-oriented and AO-like language that uses event types, features explicit announcement of events, and where handlers can react upon events in a manner similar to AOP advice. Event types abstract the occurrence of particular events in a system, can expose context through bound variables, and can constrain the behavior of handlers through blackbox contracts.

Bagherzadeh et al. (2011) proposed *translucid contracts* for Ptolemy as a mechanism to specify the control-flow behavior of handlers—which was not possible using just blackbox specifications. Based on structural refinements, a translucid contract is a template-like algorithm that abstractly describes the behavior expected from concrete handlers. The authors show how the categories of advice related to control flow defined by Rinard et al. (2004) can be specified using translucid contracts. Additionally, they show a small example to argue that translucid contracts are strictly more expressive that those categories.

A core feature of translucid contracts is that they support modular verification. This is done by combining a static verification step in the type checking phase of the compiler, which checks that

handlers are actual refinements of the translucid contract, with certain dynamic runtime checks that are required due to some dynamic deployment features of their language.

Our work is fairly similar to that of Ptolemy and translucid contracts. Both approaches feature explicit announcement of join points (resp. events). Also, both translucid contracts and protected pointcuts serve as an external specification to which external advices or handlers must conform. Moreover, these specifications are based on types: a translucid contract is bound to an event type, and the type of a protected pointcut directly represents the restrictions. We believe that the model of tagged function applications (Chapter 12) can be designed to closely represent the approach of event types as implemented in Ptolemy.

Regarding the specification of control flow behavior, both approaches can at least express the categories of Rinard et al. (2004). Also, in their work Bagherzadeh et al. (2011) recognize the similarity between translucid contracts and EffectiveAdvice: "Their work shares commonalities with ours in terms of explicit interfaces having more expressive contracts to state and enforce the behavior of interactions. However, it is difficult to adapt their ideas built upon their non-AO core language [...] as they do not support quantification" (Bagherzadeh et al., 2011). It is not clear however which approach is more expressive. An advantage of our approach is that protected pointcuts can also impose restrictions on the computational effects that advices can use.

Formal verification of aspects

Starting from his pioneering work on superposition for distributed systems (Katz, 1993), Katz (2006) has later refined his work to give a classification of aspects. He distinguishes three kinds of classes of temporal behavior: *spectative superposition* (that amounts to harmlessness), *regulative superposition* (that can modify which actions occur, but cannot change the computation performed by an individual action) and *invasive superposition* (that can change anything). Inspired by these categories, Djoko Djoko et al. (2006) have recently proposed to capture observer, aborter and confiner aspects directly in the language under consideration. Namely for each category, they define a specific aspect language with the property that any aspect written in that language belongs to the category.

Recently, Disenfeld and Katz (2013) defined a compositional model checking method for events and aspects specification using temporal logic on event detection. The technique is used to detect interference in systems where aspects may be activated during the execution of other aspects.

In a similar approach, Krishnamurthi et al. (2004) also present a technique for modular verification of aspects. Given a set of temporal logic properties that must be satisfied along with a fixed set of pointcuts, they generate sufficient conditions on the pointcuts themselves to enable modular verification.

Part IV

Conclusions

Chapter 15

Contributions

This chapter briefly reviews the main contributions of this thesis. To summarize, this thesis presented two main original contributions:

1. A lightweight, full-fledged, typed, and monadic embedding of the pointcut/advice model of aspect-oriented programming in Haskell.
2. A general theorem of compositional harmlessness, proved for an abstract monadic AOP framework.

We now discuss in more detail the specific contributions embodied in each of these two items.

A Full-fledged Monadic Embedding of Aspects

Our first main contribution is the development of a novel approach to embed aspects in a existing language. We exploit monads and the Haskell type system to define a typed monadic embedding that supports both modular language extensions and (informal) reasoning about effects with pointcut/advice aspects.

A core specific contribution is the proof that type soundness follows from our design, despite using a potentially unsafe type coercion. Although the use of anti-unification for typing aspects is not novel (*cf.* AspectML), it encompasses an important contribution of our work. First, to the best of our knowledge, this is the first anti-unification algorithm that works statically (at compile time) on types themselves. This is done exploiting Haskell type classes as a means to perform type-level computation. Second, the ability to perform type-level anti-unification is crucial in avoiding the development of a special purpose or ad-hoc type system (either from scratch or as an extension). Because we use the plain Haskell type system, our library is potentially easier to maintain through future iterations of the Haskell language.

Another specific contribution is that we reconcile the flexibility of dynamically-typed aspect languages, like AspectScheme and AspectScript, in particular first-class aspects, pointcuts and advices, and user-defined pointcuts; with the guarantees of statically typed languages like AspectML.

Moreover, compared to other approaches to statically-typed polymorphic aspect languages, the proposed embedding is more lightweight, expressive, extensible, and amenable to interference analysis.

An interesting point that reflects the tradeoff between expressiveness and safety is the definition of the pointcut language. On the one hand, allowing user-defined pointcuts brings the flexibility of dynamic aspect languages, but entails the responsibility of proving their soundness in a case-by-case basis. On the other hand, we can ensure pointcut safety by providing only a fixed set of predefined safe pointcuts and logical combinators, as is done in AspectML (Dantas et al., 2008). In our design we opted for the former because we believe that its benefits far outweighs its cost, in particular when considering our framework as a research tool to experiment with novel aspect semantics. In addition we consider that this approach is compatible with the design philosophy of Haskell to provide "escape hatches", like $unsafeCoerce$, in order to go beyond what the type system can currently prove as safe.

Additionally, our model can be regarded as the marriage of Open Modules with EffectiveAdvice. On the one hand, we can use the standard module system of Haskell to present a public interface, while still allowing internal aspects (although for now it is limited to $pcCall$ pointcuts, just like in Open Modules). On the other hand we can use the advice combinators from EffectiveAdvice in order to restrict the behavior of advice. Our proposal of protected pointcuts neatly encompasses both approaches by allowing developers to present a protected interface for external advising.

Finally, we have illustrated how to exploit the capability to implement modular language extensions. We believe that our model can serve as a vehicle for research into specific aspect semantics without the burden of developing a full-blown language from scratch. Moreover, all reasoning techniques or theorems developed in the monadic setting are immediately available to these language extensions, without any cost to researchers.

A Framework for Compositional Interference Reasoning

In the pointcut/advice model of aspect-oriented programming, unrestricted quantification through pointcuts forces global reasoning. We show that such global reasoning can be compositional. Compositionality is crucial for formal reasoning to scale up to large systems; equivalence proofs are hard to develop, so they should be partially reused as much as possible when a system evolves. We develop a framework for compositional reasoning about interference, using monads to express and reason about effects in a pure functional setting.

We introduce a general equivalence theorem that relies on four sufficient conditions—namely compositional weaving, compositional projection of effects, contextual and local harmlessness—that can be proven and reused independently. We demonstrate how the framework can be used to reason about a variety of scenarios related to the evolution of aspect-oriented programs.

Chapter 16

Perspectives

Based on the limitations of our work and some ideas originated through its development, we now outline several potential directions for future research.

Regarding AOP and MRI

The first and most direct line of work is to scale the expressiveness of the model defined in Part III to that of the full-fledged model of Parts I and II, while preserving the reasoning results established in Chapter 11. Because MRI does not features quantification, this probably requires the adaptation of the techniques used in MRI to address effectful pointcuts.

Another line of work involving MRI was described before in Chapter 13. Recall that to prove local harmlessness we opted to translate AOP programs into the MRI setting, rather than lift the MRI results directly to our AOP model. This explicitly limits the applicability of our framework, in particular Theorem 3, to advices which can expressed as a mixin. Hence it is not possible yet to directly reuse our results in aspect semantics that allow aspects to advice other aspects. Second, the statement of Theorem 3 requires a suitable f_{MRI} function that—we conjecture—might be systematically derived from the corresponding f_{AOP} function. In this situation it would be useful to rephrase the theorem in terms of the aspect model in order to avoid work that might be tedious or error prone. Finally, further investigation on how to apply the results from MRI directly into the AOP setting may yield new and interesting research questions related to the nature of quantification and its interaction with effects.

Mechanization of the Model

To be stricter in the formalization of our model and our proofs, we are interested in describing the model of Part III in the Coq proof assistant. A reason to choose Coq, besides personal preference, is to benefit from the recent formalization of monad transformers by Delaware et al. (2013).

Modular Reasoning

A promising line of future research is to study means to strengthen compositional reasoning to achieve modular reasoning under certain scenarios. For instance, our definition of protected pointcuts (Section 7.3) has yet to be formalized. Additionally, because, ultimately, unrestricted quantification is incompatible with modular reasoning, it is appealing to combine the coarse-grained modular reasoning provided by Open Modules (Aldrich, 2005) with our compositional reasoning techniques for reasoning about equivalence of modules.

Reasoning About Language Extensions

Chapter 9 shows how to modularly define new aspect semantics, in particular execution levels. It would be interesting to exploit our framework for compositional reasoning in order to establish formal properties of such extensions. For example, in the case of execution levels it is proven that certain kinds of loops are avoided by default (Tanter et al., 2014); this is proven in the setting of a core calculus based on small-step operational semantics. It would be interesting to compare the development of a similar proof in our setting, using equational reasoning and parametricity.

Another interesting point is related to the membrane semantics for AOP. In previous work we informally suggested that the membrane model in combination with the control-flow combinators (Section 7.4) were sufficient to avoid a large range of scenarios of aspect interference (Figueroa et al., 2013). This can be now formalized in the reasoning framework developed in this work.

Handling of the Monadic Stack

In Chapter 8 we illustrated that the handling of the monadic stack was crucial to establish non-interference properties between components in a system. We also proposed to use monad views as a potential solution to some of these issues. However there is a gap between the specification of monad views and the formal results established in MRI: namely, the parametricity-based theorems that underlie the proofs of MRI have not been established for monad views. We believe this could be a fruitful line of future work that goes beyond aspect-oriented programming.

Anti-unification with Subtyping

The main theoretical result of Part I is to show that by using a type-level anti-unification algorithm we can ensure the type safety of pointcut/advice pairs. A particular limitation of this algorithm is that it does not take into account type class constraints, therefore the least general type of two types whose constructors do not match is just an *unbounded* type variable.

However, in a general setting, the real shortcoming is that anti-unification does not take into

account the subtyping relation of the types in a system (the type class hierarchy in the case of Haskell). Ideally, by taking the type hierarchy into account, a novel anti-unification algorithm could derive a least general type that is more interesting and useful than just a type variable. This line of work appears to be closely related to the type system proposed in StrongAspectJ (De Fraine et al., 2008).

Connection with Type-and-effect Systems

The connection between monads and type-and-effect systems is well known (Wadler, 1998). It may be fruitful to establish the correspondence between our monadic model of aspects and a type-and-effect systems. Moreover, and related also to the handling of the monadic stack, we are interested in investigating the connection between monads and type-and-capabilities systems as proposed by Pottier (2013).

Compositional Reasoning for AspectJ

From the practitioners' point of view it can be argued that our compositional harmlessness theorem is too specific to Haskell and to the monadic setting. It remains to be seen how we can translate the results, or at least the intuitions, behind our development to a mainstream aspect language like AspectJ. In particular we believe that the four preconditions of the theorem may be incorporated as heuristics for the visualization of aspects in a system, for example as developed in AspectMaps (Fabry et al., 2014).

Empirical Evaluation of Crosscutting Concerns in Haskell

Because Haskell is used not only for research but also for development in the software industry, we are interested in performing an empirical study of whether aspects can improve off-the-shelves Haskell software. To this end we plan to analyze (part of) the *Hackage* software repository. Hackage is the *de-facto* repository for open source software written in Haskell, featuring over 5000 packages written by over 1000 people. As a first step we plan to use existing aspect-mining techniques (*e.g.* those surveyed by Kellens et al. (2007)) to identify the prevalence of crosscutting concerns in Haskell packages. Then, we would like to perform some case studies comparing the original and AO-refactored versions of some subset of software packages.

Bibliography

ALDRICH, J. 2005. Open modules: Modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, A. P. Black, Ed. Number 3586 in Lecture Notes in Computer Science. Springer-Verlag, Glasgow, UK, 144–168.

AOSD 2008 2008. *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*. ACM Press, Brussels, Belgium.

AOSD 2010 2010. *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*. ACM Press, Rennes and Saint Malo, France.

AOSD 2011 2011. *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. ACM Press, Porto de Galinhas, Brazil.

ARACIC, I., GASIUNAS, V., MEZINI, M., AND OSTERMANN, K. 2006. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*. Lecture Notes in Computer Science Series, vol. 3880. Springer-Verlag, 135–173.

ASSAF, A. AND NOYÉ, J. 2008. Dynamic AspectJ. In *Proceedings of the 4th ACM Dynamic Languages Symposium (DLS 2008)*. ACM Press, Paphos, Cyprus.

AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, J., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2006. abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*. Lecture Notes in Computer Science Series, vol. 3880. Springer-Verlag, 293–334.

BAGHERZADEH, M., RAJAN, H., LEAVENS, G. T., AND MOONEY, S. 2011. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. See AOSD 2011 (2011).

BOCKISCH, C., SEWE, A., MEZINI, M., AND AKŞIT, M. 2011. An overview of alia4j: An execution model for advanced-dispatching languages. J. Bishop and A. Vallecillo, Eds. Springer-Verlag, Berlin, Heidelberg, 131–146.

BODDEN, E. 2011. Closure joinpoints: Block joinpoints without surprises. See AOSD 2011 (2011), 117–128.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17,* 4, 471–523.

CLIFTON, C. AND LEAVENS, G. T. 2002. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proceedings of the 1st Workshop on Foundations of Aspect-Oriented Languages (FOAL 2002)*, G. T. Leavens and R. Cytron, Eds. Number 02-06 in Technical Report. Department of Computer Science, Iowa State University.

CLIFTON, C., LEAVENS, G. T., AND NOBLE, J. 2007. MAO: Ownership and effects for more effective reasoning about aspects. In *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, E. Ernst, Ed. Number 4609 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 451–475.

DANTAS, D. S. AND WALKER, D. 2006. Harmless advice. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*. ACM Press, Charleston, South Carolina, USA, 383–396.

DANTAS, D. S., WALKER, D., WASHBURN, G., AND WEIRICH, S. 2008. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems 30,* 3, Article No. 14.

DE FRAINE, B., SÜDHOLT, M., AND JONCKERS, V. 2008. StrongAspectJ: flexible and safe pointcut/advice bindings. See AOSD 2008 (2008), 60–71.

DE MEUTER, W. 1997. Monads as a theoretical foundation for aop. In *In International Workshop on Aspect-Oriented Programming at ECOOP*. Springer-Verlag, 25.

DELAWARE, B., EN TOM SCHRIJVERS, S. K., AND D. S. OLIVEIRA, B. C. 2013. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN Conference on Functional Programming (ICFP 2013)*. ACM Press, Boston, MA, USA.

DISENFELD, C. AND KATZ, S. 2013. Specification and verification of event detectors and responses. See Kinzle (2013).

DJOKO DJOKO, S., DOUENCE, R., FRADET, P., AND LE BOTLAN, D. 2006. CASB: Common aspect semantics base. Tech. Rep. AOSD-Europe Deliverable D41, AOSD-Europe-INRIA-7, INRIA, France.

DOUENCE, R., FRADET, P., AND SÜDHOLT, M. 2002. A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, D. Batory, C. Consel, and W. Taha, Eds. Lecture Notes in Computer Science Series, vol. 2487. Springer-Verlag, Pittsburgh, PA, USA, 173–188.

DOUENCE, R., FRADET, P., AND SÜDHOLT, M. 2004. Composition, reuse and interaction analysis of stateful aspects. See Lieberherr (2004), 141–150.

DOUENCE, R., FRADET, P., AND SÜDHOLT, M. 2005. Trace-based aspects. In *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, Eds. Addison-Wesley, Boston, 201–217.

DOUENCE, R., MOTELET, O., AND SÜDHOLT, M. 2001. A formal definition of crosscuts. In

Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001), A. Yonezawa and S. Matsuoka, Eds. Lecture Notes in Computer Science Series, vol. 2192. Springer-Verlag, Kyoto, Japan, 170–186.

DUTCHYN, C., TUCKER, D. B., AND KRISHNAMURTHI, S. 2006. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming 63,* 3, 207–239.

EKMAN, T. AND HEDIN, G. 2007. The JastAdd extensible Java compiler. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*. ACM Press, Montreal, Canada, 1–18. ACM SIGPLAN Notices, 42(10).

ERNST, E., Ed. 2014. *Proceedings of the 13th International Conference on Modularity*. ACM Press, Lugano, Switzerland. To appear.

FABRY, J. AND GALDAMES, D. 2014. PHANtom: a modern aspect language for Pharo Smalltalk. *Software—Practice and Experience 44,* 4, 393–412.

FABRY, J., KELLENS, A., DENIER, S., AND DUCASSE, S. 2014. AspectMaps: Extending Moose to visualize AOP software. *Science of Computer Programming 79,* 1, 6–22.

FIGUEROA, I., SCHRIJVERS, T., TABAREAU, N., AND TANTER, É. 2014. Compositional reasoning about aspect interference. See Ernst (2014). To appear.

FIGUEROA, I., TABAREAU, N., AND TANTER, É. 2013. Taming aspects with monads and membranes. In *Proceedings of the 12th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2013)*. ACM Press, Fukuoka, Japan, 1–6.

FIGUEROA, I., TABAREAU, N., AND TANTER, É. 2014. Effective Aspects: A typed monadic embedding of aspects. *8400*, 145–192.

FIGUEROA, I. AND TANTER, É. 2011. A semantics for execution levels with exceptions. In *Proceedings of the 10th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2011)*. ACM Press, Porto de Galinhas, Brazil, 7–11.

FIGUEROA, I., TANTER, É., AND TABAREAU, N. 2012. A practical monadic aspect weaver. See FOAL (2012), 21–26.

FILMAN, R. E. AND FRIEDMAN, D. P. 2000. Aspect-oriented programming is quantification and obliviousness. Tech. rep.

FOAL 2012. *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*. ACM Press, Potsdam, Germany.

GRISWOLD, W. G., SULLIVAN, K., SONG, Y., SHONLE, M., TEWARI, N., CAI, Y., AND RAJAN, H. 2006. Modular software design with crosscutting interfaces. *IEEE Software 23,* 1, 51–60.

HOFER, C. AND OSTERMANN, K. 2007. On the relation of aspects and monads. In *Proceedings of AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007)*. 27–33.

HOFFMAN, K. AND EUGSTER, P. 2007. Bridging Java and AspectJ through explicit join points. In *Proceedings of the 9th International Symposium on Principles and Practice of Programming in Java*. ACM Press, 63–72.

JONES, M. P. 2000. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*. Number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 230–244.

KATZ, S. 1993. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems 15,* 2, 337–356.

KATZ, S. 2006. Aspect categories and classes of temporal properties. In *Transactions on Aspect-Oriented Software Development*. Lecture Notes in Computer Science Series, vol. 3880. Springer-Verlag, 106–134.

KELLENS, A., MENS, K., AND TONELLA, P. 2007. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 143–162.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. 2001. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, J. L. Knudsen, Ed. Number 2072 in Lecture Notes in Computer Science. Springer-Verlag, Budapest, Hungary, 327–353.

KICZALES, G., IRWIN, J., LAMPING, J., LOINGTIER, J., LOPES, C., MAEDA, C., AND MENDHEKAR, A. 1996. Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, M. Akşit and S. Matsuoka, Eds. Lecture Notes in Computer Science Series, vol. 1241. Springer-Verlag, Jyväskylä, Finland, 220–242.

KICZALES, G. AND MEZINI, M. 2005. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering (ICSE 2005)*. ACM Press, St. Louis, MO, USA, 49–58.

KINZLE, J., Ed. 2013. *Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013)*. ACM Press, Fukuoka, Japan.

KRISHNAMURTHI, S., FISLER, K., AND GREENBERG, M. 2004. Verifying aspect advice modularly. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*. 137–146.

LEIJEN, D. AND MEIJER, E. 1999. Domain specific embedded compilers. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, T. Ball, Ed. 109–122.

LIANG, S., HUDAK, P., AND JONES, M. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL 95)*.

ACM Press, San Francisco, California, USA, 333–343.

Lieberherr, K., Ed. 2004. *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*. ACM Press, Lancaster, UK.

Masuhara, H. and Kawauchi, K. 2003. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03)*. Lecture Notes in Computer Science Series, vol. 2895. 105–121.

Masuhara, H. and Kiczales, G. 2003. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, L. Cardelli, Ed. Number 2743 in Lecture Notes in Computer Science. Springer-Verlag, Darmstadt, Germany, 2–28.

Masuhara, H., Kiczales, G., and Dutchyn, C. 2003. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003)*, G. Hedin, Ed. Lecture Notes in Computer Science Series, vol. 2622. Springer-Verlag, 46–60.

Masuhara, H., Tatsuzawa, H., and Yonezawa, A. 2005. Aspectual Caml: an aspect-oriented functional language. In *Proceedings of the 10th ACM SIGPLAN Conference on Functional Programming (ICFP 2005)*. ACM Press, Tallin, Estonia, 320–330.

Moggi, E. 1991. Notions of computation and monads. *Information and Computation 93,* 1, 55–92.

Oliveira, B. C. d. S., Schrijvers, T., and Cook, W. R. 2010. EffectiveAdvice: discplined advice with explicit effects. See AOSD 2010 (2010), 109–120.

Oliveira, B. C. D. S., Schrijvers, T., and Cook, W. R. 2012. MRI: Modular reasoning about interference in incremental programming. *Journal of Functional Programming 22*, 797–852.

Parnas, D. 1972. On the criteria for decomposing systems into modules. *Communications of the ACM 15,* 12, 1053–1058.

Peyton Jones, S., Vytiniotis, D., Weirich, S., and Shields, M. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming 17,* 1, 1–82.

Pierce, B. 2012. Types. http://www.seas.upenn.edu/ sweirich/plmw12/Slides/plmw12-Pierce.pdf.

Pierce, B. C. 2002. *Types and programming languages*. MIT Press, Cambridge, MA, USA.

Plotkin, G. D. 1970. A note on inductive generalization. *Machine Intelligence 5*, 153–163.

Pottier, F. 2013. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming 23,* 1, 38–144.

Rajan, H. and Leavens, G. T. 2008. Ptolemy: A language with quantified, typed events.

In *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, J. Vitek, Ed. Number 5142 in Lecture Notes in Computer Science. Springer-Verlag, Paphos, Cyprus, 155–179.

REYNOLDS, J. C. 1970. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence 5*, 135–151.

RINARD, M., SALCIANU, A., AND BUGRARA, S. 2004. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM Symposium on Foundations of Software Engineering (FSE 12)*. ACM Press, 147–158.

SCHRIJVERS, T. AND OLIVEIRA, B. C. 2011. Monads, zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN Conference on Functional Programming (ICFP 2011)*. ACM Press, Tokyo, Japan, 32–44.

SHEARD, T. AND JONES, S. P. 2002. Template meta-programming for haskell. *SIGPLAN Not. 37,* 12, 60–75.

STEIMANN, F., PAWLITZKI, T., APEL, S., AND KÄSTNER, C. 2010. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology 20,* 1, Article 1.

SULLIVAN, K., GRISWOLD, W. G., RAJAN, H., SONG, Y., CAI, Y., SHONLE, M., AND TEWARI, N. 2010. Modular aspect-oriented design with XPIs. *ACM Transactions on Software Engineering and Methodology 20,* 2. Article 5.

SULZMANN, M. AND WANG, M. 2007. Aspect-oriented programming with type classes. In *Proceedings of the Sixth Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007)*. ACM Press, Vancouver, British Columbia, Canada, 65–74.

TABAREAU, N. 2012. A monadic interpretation of execution levels and exceptions for AOP. In *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD 2012)*, É. Tanter and K. J. Sullivan, Eds. ACM Press, Potsdam, Germany.

TABAREAU, N., FIGUEROA, I., AND TANTER, É. 2013. A typed monadic embedding of aspects. See Kinzle (2013), 171–184.

TANTER, É. 2008. Expressive scoping of dynamically-deployed aspects. See AOSD 2008 (2008), 168–179.

TANTER, É. 2010. Execution levels for aspect-oriented programming. See AOSD 2010 (2010), 37–48.

TANTER, É., FIGUEROA, I., AND TABAREAU, N. 2014. Execution levels for aspect-oriented programming: Design, semantics, implementations and applications. *Science of Computer Programming 80,* 1, 311–342.

TANTER, É., MORET, P., BINDER, W., AND ANSALONI, D. 2010. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Gen-*

erative Programming and Component Engineering (GPCE 2010). ACM Press, Eindhoven, The Netherlands, 113–122.

TANTER, É., TABAREAU, N., AND DOUENCE, R. 2012. Taming aspects with membranes. See FOAL (2012), 3–8.

TARR, P. L., OSSHER, H. L., HARRISON, W. H., AND JR., S. M. S. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*. 107–119.

TOLEDO, R., LEGER, P., AND TANTER, É. 2010. AspectScript: Expressive aspects for the Web. See AOSD 2010 (2010), 13–24.

WADLER, P. 1992. The essence of functional programming. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL 92)*. ACM Press, Albuquerque, New Mexico, USA, 1–14.

WADLER, P. 1998. The marriage of effects and monads. *ACM SIGPLAN Notices 34,* 1, 63–74.

WADLER, P. AND BLOTT, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 89)*. ACM Press, Austin, TX, USA, 60–76.

WALKER, D., ZDANCEWIC, S., AND LIGATTI, J. 2003. A theory of aspects. In *Proceedings of the 8th ACM SIGPLAN Conference on Functional Programming (ICFP 2003)*. ACM Press, Uppsala, Sweden, 127–139.

WAND, M., KICZALES, G., AND DUTCHYN, C. 2004. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems 26,* 5, 890–910.

Appendix A

Proofs of the Properties of the Simple Monadic AOP Model

A.1 Monad Laws

A.1.1 Left Identity

$$
\begin{array}{l}
\mathit{return}_{\mathbb{A}_\mathrm{T}}\ x\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f \\
\equiv\ \ \{\text{-unfolding } \gg\!=_{\mathbb{A}_\mathrm{T}} \text{ and } \mathit{return}_{\mathbb{A}_\mathrm{T}} \text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ (\mathbb{A}_\mathrm{T}\ (\mathit{return}_m\ (\mathit{Return}\ x)))\ \gg\!=_m \lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\qquad \rightarrow \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow \mathit{return}_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)) \\
\equiv\ \ \{\text{-}\mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv \mathit{id}\ \text{-}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{return}\ (\mathit{Return}\ a)\ \gg\!=_m \lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\qquad \rightarrow \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow \mathit{return}_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)) \\
\equiv\ \ \{\text{-left identity of } \gg\!=_m \text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\textbf{case}\ \mathit{Return}\ x\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\qquad \rightarrow \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow \mathit{return}_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)) \\
\equiv\ \ \{\text{-applying } \textbf{case} + \mathbb{A}_\mathrm{T} \circ \mathit{un}\mathbb{A}_\mathrm{T} \equiv \mathit{id}\ \text{-}\} \\
f\ x
\end{array}
$$

A.1.2 Right Identity

$$
\begin{array}{l}
p\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ \mathit{return}_{\mathbb{A}_\mathrm{T}} \\
\equiv\ \ \{\text{-unfolding } \gg\!=_{\mathbb{A}_\mathrm{T}} \text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m \lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \\
\qquad \mathit{Return}\ x \qquad\qquad \rightarrow \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{return}\ x) \\
\qquad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow \mathit{return}_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ \mathit{return}_{\mathbb{A}_\mathrm{T}}))
\end{array}
$$

$$\equiv \quad \{\text{-}un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}} \equiv id \text{ + unfolding } return_{\mathbb{A}_{\mathrm{T}}} \text{ -}\}$$
$$\begin{array}{l}
\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m \lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\quad\ \rightarrow un\mathbb{A}_{\mathrm{T}}\ (\mathbb{A}_{\mathrm{T}}\ (return_m\ (\mathit{Return}\ x))) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow return_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ return_{\mathbb{A}_{\mathrm{T}}}))
\end{array}$$
$$\equiv \quad \{\text{-}\mathbb{A}_{\mathrm{T}} \circ un\mathbb{A}_{\mathrm{T}} \equiv id \text{ + co-induction hypothesis -}\}$$
$$\begin{array}{l}
\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m\ (\lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\quad\ \rightarrow return_m\ (\mathit{Return}\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow return_m\ (\mathit{OpenApp}\ t\ x\ g\ k))
\end{array}$$
$$\equiv \quad \{\text{-folding } \textbf{case} \text{ branches -}\}$$
$$\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m\ (\lambda r \rightarrow return_m\ r))$$
$$\equiv \quad \{\text{-}\eta\text{-reduction -}\}$$
$$\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m\ return_m)$$
$$\equiv \quad \{\text{-right identity of } \gg\!=_m \text{ + } un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}} \equiv id \text{ -}\}$$
$$p$$

A.1.3 Associativity of $\gg\!=_{\mathbb{A}_{\mathrm{T}}}$

$$(p\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ f)\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ h$$
$$\equiv \quad \{\text{-unfold } \gg\!=_{\mathbb{A}_{\mathrm{T}}} \text{ -}\}$$
$$\begin{array}{l}
[\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m \lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\quad\ \rightarrow un\mathbb{A}_{\mathrm{T}}\ (f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow return_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ f))]\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ h
\end{array}$$
$$\equiv \quad \{\text{-unfold } \gg\!=_{\mathbb{A}_{\mathrm{T}}} \text{ + simplifications -}\}$$
$$\mathbb{A}_{\mathrm{T}}\ ((un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m\ (\lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \ldots))\ \gg\!=_m\ (\lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \ldots))$$
$$\equiv \quad \{\text{-associativity of } \gg\!=_m \text{ -}\}$$
$$\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m \lambda x \rightarrow ((\lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \ldots)\ x\ \gg\!=_m\ (\lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \ldots)))$$
$$\equiv \quad \{\text{-}\beta\text{-reduction -}\}$$
$$\begin{array}{l}
\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m \lambda x \rightarrow (\textbf{case}\ x\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\quad\ \rightarrow un\mathbb{A}_{\mathrm{T}}\ (f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow return_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ f)) \\
\quad \gg\!=_m\ (\lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \ldots))
\end{array}$$
$$\equiv \quad \{\text{-distributing } \gg\!=_m \text{ over the } \textbf{case} \text{ branches -}\}$$
$$\begin{array}{l}
\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m \lambda x \rightarrow (\textbf{case}\ x\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\quad\ \rightarrow un\mathbb{A}_{\mathrm{T}}\ (f\ x)\ \gg\!=_m\ (\lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \ldots) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow return_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ f) \\
\quad \gg\!=_m\ (\lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \ldots)))
\end{array}$$
$$\equiv \quad \{\text{-}id \equiv un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}} \text{ + left unit of } m \text{ and } \textbf{case} \text{ -}\}$$
$$\begin{array}{l}
\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m \lambda x \rightarrow (\textbf{case}\ x\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\quad\ \rightarrow un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ (f\ x)\ \gg\!=_m\ (\lambda r \rightarrow \textbf{case}\ r\ \textbf{of} \ldots)) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow return_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ f\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ h)))
\end{array}$$
$$\equiv \quad \{\text{-folding definition of } \gg\!=_{\mathbb{A}_{\mathrm{T}}} \text{ -}\}$$
$$\begin{array}{l}
\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m \lambda x \rightarrow (\textbf{case}\ x\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\quad\ \rightarrow un\mathbb{A}_{\mathrm{T}}\ (f\ x\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ h) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow return_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ f\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ h)))
\end{array}$$
$$\equiv \quad \{\text{-}\eta\text{-abstraction + }\alpha\text{-renaming -}\}$$
$$\begin{array}{l}
\mathbb{A}_{\mathrm{T}}\ (un\mathbb{A}_{\mathrm{T}}\ p \ \gg\!=_m \lambda r \rightarrow (\textbf{case}\ r\ \textbf{of} \\
\quad \mathit{Return}\ x \qquad\quad\ \rightarrow un\mathbb{A}_{\mathrm{T}}\ ((\lambda x \rightarrow f\ x\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ h)\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \rightarrow return_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_{\mathrm{T}}} \lambda x \rightarrow f\ x\ \gg\!=_{\mathbb{A}_{\mathrm{T}}}\ h)))
\end{array}$$

$\equiv$ {-folding definition of $\gg\!=_{\mathbb{A}_\mathrm{T}}$ -}
$p \gg\!=_{\mathbb{A}_\mathrm{T}} \lambda x \to (f\ x \gg\!=_{\mathbb{A}_\mathrm{T}} h)$

A.2 Monad Transformer Laws

A.2.1 Identity Preservation

$\mathit{lift}\ (\mathit{return}_m\ x)$
$\equiv$ {-unfold *lift* -}
$\mathbb{A}_\mathrm{T}\ (\mathit{return}_m\ x \gg\!=_m (\lambda a \to \mathit{return}_m \circ \mathit{Return}\ a))$
$\equiv$ {-left identity -}
$\mathbb{A}_\mathrm{T}\ (\mathit{return}_m \circ \mathit{Return}\ x)$
$\equiv$
$\mathit{return}_{\mathbb{A}_\mathrm{T}}\ x$

A.2.2 Composition Preservation

$\mathit{lift}\ m \gg\!=_{\mathbb{A}_\mathrm{T}} (\mathit{lift} \circ f)$
$\equiv$ {-unfold $\gg\!=_{\mathbb{A}_\mathrm{T}}$ -}
$\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift}\ m) \gg\!=_m \lambda r \to$ **case** r **of**
  $\mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift} \circ f\ x)$
  $\mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y \gg\!=_{\mathbb{A}_\mathrm{T}} f))$
$\equiv$ {-unfold *lift* -}
$\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ (\mathbb{A}_\mathrm{T}\ (m \gg\!=_m \lambda a \to \mathit{return}_m\ (\mathit{Return}\ a))) \gg\!=_m \lambda r \to$ **case** r **of**
  $\mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift} \circ f\ x)$
  $\mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y \gg\!=_{\mathbb{A}_\mathrm{T}} f)))$
$\equiv$ {-$\mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv \mathit{id}$ + associativity of $\gg\!=_m$ -}
$\mathbb{A}_\mathrm{T}\ (m \gg\!=_m \lambda a \to (\mathit{return}_m\ (\mathit{Return}\ a) \gg\!=_m \lambda r \to$ **case** r **of**
  $\mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift} \circ f\ x)$
  $\mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$\ \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y \gg\!=_{\mathbb{A}_\mathrm{T}} f)))$
$\equiv$ {-left identity + **case** -}
$\mathbb{A}_\mathrm{T}\ (m \gg\!=_m \lambda a \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift} \circ f\ a))$
$\equiv$ {-unfold *lift* -}
$\mathbb{A}_\mathrm{T}\ (m \gg\!=_m \lambda a \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathbb{A}_\mathrm{T}\ (f\ a \gg\!=_m \lambda a \to \mathit{return}_m\ (\mathit{Return}\ a))))$
$\equiv$ {-$\mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv \mathit{id}$ + η-reduction + assoc. of $\gg\!=_m$ -}
$\mathbb{A}_\mathrm{T}\ ((m \gg\!=_m f) \gg\!=_m \lambda a \to \mathit{return}_m\ (\mathit{Return}\ a))$
$\equiv$ {-fold *lift* -}
$\mathit{lift}\ (m \gg\!=_m f)$

A.3 $run\mathbb{A}_{\mathrm{T}}$ is a Monad Morphism

A.3.1 Identity preservation

$run\mathbb{A}_{\mathrm{T}}\ aenv \circ return_{\mathbb{A}_{\mathrm{T}}}$
$\equiv$ {-unfolding $return_{\mathbb{A}_{\mathrm{T}}}$ -}
$run\mathbb{A}_{\mathrm{T}}\ aenv \circ \mathbb{A}_{\mathrm{T}} \circ return_m \circ Return$
$\equiv$ {-unfolding $run\mathbb{A}_{\mathrm{T}}$ -}
$un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}} \circ return_m \circ Return \ \gg\!=_m \ go$
$\equiv$ {-$un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}} \equiv id$ + left identity -}
$go \circ Return$
$\equiv$ {-evaluation -}
$return_m$

A.3.2 Compositionality

See Figure 12.2.

A.3.3 $run\mathbb{A}_{\mathrm{T}}$ is left inverse of $lift$

$run\mathbb{A}_{\mathrm{T}}\ aenv\ (lift\ m)$
$\equiv$ {-unfold $lift$ -}
$run\mathbb{A}_{\mathrm{T}}\ aenv\ (\mathbb{A}_{\mathrm{T}}\ (m \ \gg\!=_m \lambda a \rightarrow return_m \circ Return\ a))$
$\equiv$ {-unfold $run\mathbb{A}_{\mathrm{T}}$ -}
$un\mathbb{A}_{\mathrm{T}}\ (\mathbb{A}_{\mathrm{T}}\ (m \ \gg\!=_m \lambda a \rightarrow return_m \circ Return\ a)) \ \gg\!=_m \ go$
$\equiv$ {-$un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}} \equiv id$ + associativity of $\gg\!=_m$ -}
$m \ \gg\!=_m \lambda a \rightarrow (return_m \circ Return\ a \ \gg\!=_m \ go)$
$\equiv$ {-left identity + evaluating go -}
$m \ \gg\!=_m \ return_m$
$\equiv$ {-right identity -}
m
$\equiv$ {-fold id -}
$id\ m$