

# A PRACTICAL EFFECT SYSTEM FOR SCALA

THIS IS A TEMPORARY TITLE PAGE  
It will be replaced for the final print by a version  
provided by the service academique.

Thèse N° 5935 (2013)  
présenté le 4 Septembre 2013  
à la Faculté Informatique et Communications  
Laboratoire de Méthodes de Programmation 1  
Programme Doctoral en Informatique, Communications et Infor-  
mation  
École Polytechnique Fédérale de Lausanne  
pour l'obtention du grade de Docteur ès Sciences  
par

Lukas Rytz

acceptée sur proposition du jury:

Prof Friedrich Eisenbrand, président du jury  
Prof Martin Odersky, directeur de thèse  
Prof Peter Müller, rapporteur  
Prof Ondřej Lhoták, rapporteur  
Prof Viktor Kunčák, rapporteur

Lausanne, EPFL, 2013



# Zusammenfassung

Computerprogramme interagieren mit ihrer Umgebung durch Seiteneffekte zur Ein- und Ausgabe. Andere Seiteneffekte wie Speicheränderungen oder Ausnahmen sind mächtige Werkzeuge und geben Programmierern viele Möglichkeiten. Quelltexte mit Seiteneffekten sind allerdings oft schwierig zu verstehen, Fehler verursacht durch Seiteneffekte sind schwer zu diagnostizieren und das mögliche Auftreten von Seiteneffekten hindert Übersetzer daran, den Code zu optimieren.

Das Ziel dieser Dissertation ist die Entwicklung eines praktischen Werkzeuges welches es Programmierern erlaubt, Seiteneffekte in Funktionen zu beschreiben und zu überprüfen. Wir stellen ein leichtes und einfach zu verstehendes *Typen-und-Effektsystem* vor welches Seiteneffekte, oder deren Ausbleiben, in häufigen Programmiermustern beschreiben kann. Die fundamentale Idee von Typen-und-Effektsystemen ist es, die Seiteneffekte einer Funktion in ihrer Typensignatur zu beschreiben. Um die möglichen Seiteneffekte eines Funktionsaufrufes zu berechnen benötigt das System einzig die Signatur der aufgerufenen Funktion und nicht deren Quelltext.

Obschon Typen-und-Effektsysteme seit mehr als 25 Jahren existieren ist ihr Gebrauch in etablierten Programmiersprachen minimal: die Überprüfung zur Übersetzungszeit von Ausnahmen in Java ist das einzige existierende Typen-und-Effektsystem in einer weit verbreiteten Sprache. In dieser Dissertation untersuchen wir die Probleme welche den praktischen Einsatz von Effektsystemen erschweren und schlagen neue Lösungen vor, welche diese Probleme gezielt angehen.

Methoden deren Seiteneffekte von den Effekten der Parameter abhängen sind allgegenwärtig, sowohl in objekt-orientierten als auch in funktionalen Programmiersprachen. Ein zentrales Element eines erfolgreichen Typen-und-Effektsystems ist daher eine leichte Syntax zur Beschreibung von effekt-polymorphen Funktionen. Wir entwerfen ein intuitives Notierungssystem für Effekt-Polymorphismus, sogenannte *relative Effektannotationen*, welches auf ausdrucksabhängigen Typen beruht und gleichzeitig mehrere Effekt-Domänen umfasst. Wir stellen ein generisches Effektsystem vor welches mehrere Arten von Effekten gleichzeitig überprüfen kann und das Implementieren von neuen Effekt-Domänen erleichtert. Wir untersuchen das System formell und validieren es mit einer Implementation für die Programmiersprache Scala.

---

Einer der meistbenutzten Seiteneffekte ist die Veränderung von Speicher. Diese Effekte sind aber aufgrund von Aliasing schwer zu kontrollieren. Wir stellen ein neues Effektsystem vor welches die Purity von Funktionen, die Absenz von Speicheränderungen, in funktionalen und objekt-orientierten Sprachen wie Scala überprüft. Das System kann das Ausbleiben von sichtbaren Seiteneffekten in gängigen Situationen zeigen, zum Beispiel die Konstruktion eines Containers mit einem veränderbaren Puffer. Die Effekt-Annotationen für Purity sind kompakt und einfach zu verstehen.

Wir haben das generische Effektsystem und Effekt-Domänen für EA, Ausnahmen und Purity als Übersetzer-Erweiterung für Scala implementiert. Effekt-Annotationen sind normale Typenannotationen und benötigen keine Veränderung der Syntax von Scala. Wir haben das System erfolgreich zur Überprüfung von Seiteneffekten in der Container-Bibliothek von Scala eingesetzt, welche funktionellen Code mit Seiteneffekten auf verschiedene Arten vermischt.

**Schlagerwörter:** Typen-und-Effektsysteme, Effekt-Polymorphismus, Effekt-Annotationen, Purity, Scala, Übersetzer-Erweiterungen, Aufsteckbare Typensysteme, Typensysteme, Programmanalyse

# Abstract

Computer programs interact with their environment through IO effects. Other side effects such as state modifications or exceptions are powerful tools that give flexibility to programmers. However, source code with side effects is often hard to understand, bugs involving side effects are difficult to diagnose and the possibility of side effects prevents compilers from applying optimizations.

The goal of this dissertation is to provide programmers with a practical tool that allows them to specify and to verify side effects in their programs. We propose a lightweight and easy to understand *type-and-effect system* that can express side effects, or their absence, in common programming patterns. The fundamental idea of type-and-effect systems is to include the side effects of a function in its type signature. To compute the side effects of a function invocation, the system only needs to consider the function's signature but not its source code.

Even though type-and-effect systems exist for more than 25 years, their adoption in mainstream programming languages has been minimal: the compile-time verification of checked exceptions in Java is the only type-and-effect system that exists in a widely used language. In this dissertation, we identify a number of issues that hinder the adoption of effect systems and propose new ideas that address them effectively.

Methods whose effect depends on their parameters are ubiquitous in both object-oriented and functional programming languages. Hence, a necessary ingredient of a successful effect system is a lightweight syntax for expressing effect-polymorphic functions. We design an intuitive annotation system for effect-polymorphism called *relative effect annotations*, which uses dependent types and is independent of specific effect domains. We propose a generic effect system that can check multiple kinds of effects at the same time and that can be easily extended with new effect domains. We formally study the system and validate it with an implementation for the Scala programming language.

One of the most widely used side effects is state modification, but it is also one of the most challenging to control because of aliasing. We introduce a new effect system for purity with respect to state modifications that is designed for functional and object-oriented languages like Scala. The system can express purity of common programming patterns that involve higher-order code and local state, e.g., the construction of a collection using a mutable buffer.

---

The effect annotations for purity are concise and easy to understand.

We implemented the generic framework for effect checking and effect domains for purity, IO and exceptions as a compiler plugin for Scala. Effect annotations are expressed as standard type annotations and no changes to the Scala language are required. We successfully applied the effect system to the core of the Scala collections library which mixes higher-order code and side effects in various ways.

**Keywords:** type-and-effect systems, effect-polymorphism, effect annotations, purity, Scala, compiler plugins, pluggable type systems, type systems, program analysis

# Acknowledgements

I would like to thank my advisor Martin Odersky for the guidance and inspiration that form the basis of this thesis. I am deeply grateful for the opportunity to be part of an amazing team and for his trust to let me contribute to an exciting and unique project. Martin was much more than just an advisor: it was a lot of fun to spend time with him and his family in coffee breaks, on barbecue lawns, at dinner tables and on skiing slopes.

I thank the members of my thesis jury, Friedrich Eisenbrand, Ondřej Lhoták, Peter Müller and Viktor Kunčák for their time and for the helpful comments that helped me to improve my thesis.

My past and present colleagues at LAMP were not only wonderful collaborators but equally great friends and inspiring people to look up to. Thank you all for making this PhD such a great time! A special thanks goes to Nada and Philipp for the fruitful collaboration.

I am endlessly thankful towards my parents Gertrud and Peter. They have always encouraged me to accept new challenges and supported me in pursuing my goals. I am not only moved by their patience and endurance in loving and supporting their children, but also by their humble and diligent attitude in life. I would also like to thank my brothers Christian and Hanspeter who make me realize that a family is so much more than just friendships.

Finally, I would like to thank my wife Denisa for her unconditional love and encouragement. Undoubtedly, the most exciting event during the time of my PhD was not related to research, but our beautiful marriage. I am looking forward to our common life with heartwarming joy.

*September 2013*

Lukas Rytz





# Contents

<b>Zusammenfassung (Deutsch)</b>	<b>iii</b>
<b>Abstract (English)</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>Table of Contents</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Overview . . . . .	5
1.2 Contributions . . . . .	6
1.3 Related Work . . . . .	7
1.3.1 Type-and-Effect Systems . . . . .	7
1.3.2 Monads . . . . .	8
1.3.3 Alternative Systems for Controlling Effects . . . . .	9
1.3.4 Program Verification . . . . .	10
<b>2 A Generic Framework for Polymorphic Effect-Checking</b>	<b>13</b>
2.1 Introducing Type-and-Effect Systems . . . . .	14
2.1.1 Effects Have “May” Semantics . . . . .	15
2.1.2 A Generic Representation for Effects . . . . .	16
2.2 Effect-Polymorphism . . . . .	17
2.2.1 The Need for Lightweight Syntax . . . . .	19
2.2.2 Effect-Polymorphic Function Types . . . . .	19
2.3 Abstracting Over Effect Domains . . . . .	21
2.4 Combining Multiple Effect Domains . . . . .	22
2.4.1 Annotating Multiple Effect Domains . . . . .	22
2.5 Static Semantics . . . . .	24
2.5.1 Subtyping . . . . .	24
2.5.2 Typing Rules . . . . .	26
2.6 Examples of Concrete Effect Domains . . . . .	28
2.6.1 Exceptions . . . . .	28
2.6.2 Asynchronous Operations . . . . .	30
2.7 Dynamic Semantics . . . . .	32
	ix

## Contents

---

2.7.1	Extensible Effect Domains . . . . .	32
2.7.2	Evaluation Rules . . . . .	33
2.8	Effect Soundness . . . . .	34
2.8.1	Consistency Requirement . . . . .	35
2.8.2	Soundness Proofs . . . . .	36
2.9	Conclusion . . . . .	37
<b>3</b>	<b>Dependent Types for Relative Effects Declarations</b>	<b>39</b>
3.1	Overview . . . . .	40
3.1.1	Relative Effect Declarations . . . . .	41
3.2	Formalization . . . . .	41
3.2.1	Subtyping . . . . .	42
3.2.2	Typing Rules . . . . .	48
3.3	Relative Effect Declarations in Scala . . . . .	51
3.3.1	Syntax for Relative Effect Annotations . . . . .	51
3.3.2	Refined Types for Effect-Polymorphism . . . . .	54
3.3.3	Relative Effects for Nested Definitions . . . . .	55
3.4	Expressiveness of Relative Effects . . . . .	59
3.5	Related Work . . . . .	63
3.6	Conclusion . . . . .	64
<b>4</b>	<b>A Type-and-Effect System for Purity</b>	<b>65</b>
4.1	Introduction . . . . .	65
4.2	Overview . . . . .	66
4.2.1	Purity and Modification Effects . . . . .	66
4.2.2	Ownership and Locality . . . . .	67
4.2.3	Freshness and Result Localities . . . . .	69
4.2.4	Effects of Field Updates . . . . .	70
4.2.5	Freshness Depends on Purity . . . . .	71
4.3	Formalization . . . . .	72
4.3.1	Subtyping . . . . .	74
4.3.2	Typing Rules . . . . .	75
4.3.3	Typing PUR Requires ANF . . . . .	79
4.4	Implementation of the Purity System for Scala . . . . .	80
4.4.1	Assignment Effects . . . . .	80
4.4.2	Flow-Insensitivity to Support Higher-Order Code . . . . .	83
4.4.3	Polymorphic Purity Effects . . . . .	84
4.4.4	Examples and Limitations . . . . .	86
4.5	Related Work . . . . .	93
4.5.1	Regions . . . . .	94
4.5.2	Ownership Types . . . . .	94
4.5.3	Pointer Analysis . . . . .	95
4.5.4	Other Related Work . . . . .	96

4.6	Conclusion . . . . .	97
<b>5</b>	<b>Effect Checking in Scala</b>	<b>99</b>
5.1	Programming With Effects . . . . .	99
5.1.1	Annotating Effects in Multiple Domains . . . . .	101
5.1.2	Ascriptions and Effect Casts . . . . .	102
5.1.3	Annotating Constructors and Default Arguments . . . . .	103
5.1.4	Singleton Objects, Lazy Values and By-Name Parameters . . . . .	104
5.1.5	Effects Affect Typing and Subtyping . . . . .	105
5.2	Effect Checking in the Scala Collections Library . . . . .	107
5.2.1	Option . . . . .	107
5.2.2	Breaks . . . . .	108
5.2.3	Core Collection Classes . . . . .	109
5.3	Implementing Effect Domains . . . . .	111
5.3.1	Effect Lattice . . . . .	111
5.3.2	Domain Definition . . . . .	112
5.4	Internals of the Compiler Plugin . . . . .	114
5.4.1	Compiler Plugins for Scala . . . . .	114
5.4.2	Naming and Typing in the Scala Compiler . . . . .	115
5.4.3	Implementation of the Effects Plugin . . . . .	116
5.4.4	Propagation of Type Annotations in the Scala Compiler . . . . .	119
5.4.5	Implementing Effect Checking as a Separate Compilation Phase . . . . .	123
5.5	Future Work . . . . .	124
5.5.1	Effect Annotations for Existing Libraries . . . . .	124
5.5.2	Effect Inference for Existing Libraries . . . . .	125
5.5.3	External Effect Domain Definitions . . . . .	125
5.6	Conclusion . . . . .	126
<b>6</b>	<b>Conclusion</b>	<b>127</b>
<b>A</b>	<b>Soundness Proof for LPE</b>	<b>129</b>
A.1	Lemmas . . . . .	129
A.1.1	Canonical Forms . . . . .	129
A.1.2	Value Typing Environment . . . . .	130
A.1.3	Substitution Lemmas . . . . .	130
A.2	Soundness Theorems . . . . .	134
A.2.1	Preservation . . . . .	134
A.2.2	Effect Soundness . . . . .	136
	<b>Bibliography</b>	<b>147</b>
	<b>Curriculum Vitae</b>	<b>149</b>



# Chapter 1

## Introduction

Computer programs communicate with their environment and perform changes to it. For example they can display the result of a computation, log information about their progress, write data to files or send packets in the network. Such operations are performed by invoking specific built-in functions provided by programming languages. These functions are said to be “side effecting” because they not only compute a resulting value, but also have observable effects on the environment. The side effects listed above deal with input / output (IO). In general, each observable behavior of a function other than computing a result is a computational effect. In most programming languages functions can perform effects such as modifying existing program state, returning a random value, raising an exception, entering an infinite loop or blocking a thread.

On the one hand, side effects are necessary and powerful because they enable interaction with the environment and give a lot of flexibility to programmers. For example, mutable arrays have constant time read and update operations, and hardware supported implementations make operations on them very fast on most platforms. Efficient implementations of algorithms and data structures are often based on arrays and state mutation effects. On the other hand, source code that has side effects is often hard to understand and bugs can be difficult to diagnose.

### **Purely Functional Programming Languages**

A programming language is called *purely functional* if functions written in it cannot have side effects. For the definition of functional purity which we elaborate in this section, we consider the article by Sabry [1998] entitled “What is a purely functional language?”. The fundamental characteristic that distinguishes languages with side effects from pure languages is whether the order of evaluation matters for the semantics of a program.

To illustrate the impact of the evaluation strategy on the semantics, Sabry uses a language called  $\Lambda_1$ , a lambda calculus with a global counter accessible through the primitives `inc` and

## Chapter 1. Introduction

---

read. The counter is initialized to 0, function read returns its current value and function inc increments the counter and returns its previous value. We consider the following program:

$$(\lambda x. \lambda y. y + x) \text{ inc read}$$

If we evaluate this term using call-by-value, it first reduces to “ $(\lambda x. \lambda y. y + x) 0 1$ ” and then gives the final result 1. The evaluation using call-by-name first reduces the term to “read + inc” and then yields the final result 0.

Consequently, Sabry defines a language to be purely functional if it can be implemented using either call-by-name, call-by-need or call-by-value with no observable differences, but he explicitly excludes termination effects from the observable behavior of a program. For example, in a purely functional language, the same term that produces a result in call-by-name might diverge when evaluated using call-by-value. By this definition, the above example proves that  $\Lambda_l$  is not a purely functional programming language.

One advantage of the absence of side effects is that it enables equational reasoning: if a value is defined as “ $x = \text{fun arg}$ ”, then all occurrences of the invocation “fun arg” are equal to the value  $x$ . Pure programs are easier to understand for both programmers and code analysis tools. Furthermore, since the evaluation order does not influence the semantics of a program, compilers and runtime environments have more possibilities to optimize programs. For example, independent expressions can be executed in parallel or operations on data structures can be *fused*, as shown by Coutts et al. [2007].

Yet, programs are ultimately executed for the sake of their side effects and therefore purely functional languages are faced with the problem of integrating effects. In order to remain purely functional, a language needs to ensure two properties with respect to side effects across all evaluation strategies: first, the order in which effects are executed has to be consistent, and second, effectful operations have to be executed exactly once, i.e., not discarded, nor duplicated.

The most popular solution for integrating effects into a pure language are Monads [Moggi, 1991], [Peyton Jones and Wadler, 1993]. In particular, they are widely used in the Haskell programming language [Wadler, 1997]. Other solutions include linear types [Wadler, 1990] and the idea of using witness values to create dependencies between effectful operations [Terauchi and Aiken, 2005]. The reader is referred to Chapter 1 of the PhD thesis of Lippmeier [2010] which presents an approachable and extensive introduction to these systems.

### Impure Programming Languages

Languages that are not purely functional are called *impure* programming languages. They choose to trade the ability to re-order the evaluation of expressions for the added expressiveness of allowing arbitrary side effects. Impure languages typically use a call-by-value

---

evaluation semantics because this makes programs with side effects easier to understand: a block of statements is evaluated from top to bottom and left to right, so the lexical structure reveals the order in which side effects are executed<sup>1</sup>.

Today's most popular programming languages, including C, Java, Scala, Python and JavaScript, are impure languages with call-by-value evaluation semantics. The ability to freely introduce side effects in arbitrary functions and expressions gives programmers a great amount of flexibility. For example, they can freely introduce print statements to check intermediate values of a computation when debugging an application.

The downside of this flexibility is that side effects can hide just about anywhere, or more precisely, in every function. A programmer using an external library does not know what side effects a library function invocation might cause; he has to either trust the documentation or look at the source code if available. If a function throws an exception, then a client can guard against failures by catching and handling it, but most side effects are not easily observable or reversible.

The inability to specify side effects is problematic not only for the clients of a library, but also for its authors. When a library function executes code that is passed as argument by the client, this code might have arbitrary side effects and executing it might put the library into an inconsistent state. One example can be found in Scala's futures library, which is designed for executing operations asynchronously. The contract of the library states that code passed for execution should never block its thread, however this contract is not enforced. Section 2.6.2 discusses this problem in detail and shows a possible solution.

Last but not least, information about side effects and purity can be exploited by compilers and runtime environments to optimize programs. For example, invocations of pure functions can be omitted if the returned value is never used. Such dead code is rarely written by programmers directly, however it appears in combination with other optimizations such as inlining. An example in Scala where pure code can be eliminated is when inlining a method defined in a singleton object. Constructors of singleton objects may have arbitrary side effects which are executed the first time the object or one of its members is accessed. When the compiler inlines a method defined in a singleton object, it might not know if the object has already been initialized. The compiler has to emit an invocation of the object initializer in addition to the inlined code, which ensures that the semantics are preserved. If the object initializer is known to have no side effects, this invocation can be omitted.

## **Controlling Side Effects in Impure Languages**

The goal of this dissertation is to provide programmers with a practical tool that allows them to control side effects in impure languages. Specifically, we propose a *type-and-effect system*, an

---

<sup>1</sup>In C and C++, the order of evaluation of function arguments is unspecified, but all arguments are evaluated before the function is invoked

## Chapter 1. Introduction

---

extension to an ordinary type system, which can express the effects of common programming idioms while remaining lightweight in annotation and easy to understand for programmers. We study the system using formal techniques and evaluate it with an implementation for the Scala programming language.

Type-and-effect systems were first proposed by Gifford and Lucassen [1986] with the goal to delimit the scope of effects on memory locations. The same technique can also be used to track other kinds of side effects, such as the ones described earlier in this chapter. The fundamental idea is to express the side effects of each function in its type signature, so that the type system can compute the effects of an expression using the signatures of invoked functions and knowledge about effects of built-in operations. We present a formal introduction to type-and-effect systems in Section 2.1.

Tracking side effects in the type system has a number of benefits. First of all, type systems are the most widespread technique for static code analysis and programmers are familiar with them. An extended type system does not introduce a new tool which needs to be integrated into the development workflow. Second, effect systems can be naturally incorporated into integrated development environments (IDEs), which gives developers immediate information about the side effects of the methods they are using and writing. Thanks to their inherent modularity — functions are analyzed separately using the type and effect annotations of other functions — type systems are known to scale well and work in the context of separate compilation. Finally, the annotation overhead introduced by a type system can be alleviated to some degree with local or global type inference. In a type system with global inference, types of values are computed using constraints that may depend on arbitrary parts of the program. For example, to compute the parameter type of a function, the type inference algorithm typically takes all invocations of the function into account. Type inference becomes challenging in the presence of either records or subtyping, as explained in [Pierce, 2002, Chapter 22]. Local type inference such as in Scala or C# on the other hand only uses local information to compute types of variables or functions. This typically means that parameter types need to be specified, but result types of methods, types of variables and fields and also type parameters in polymorphic method invocations can be computed.

With all these promising aspects of type-and-effect systems in mind, the natural question to ask is: Why are effect systems not more widespread? Currently there exists only one effect system in a widely used, impure programming language, namely the mechanism for verifying checked exceptions in Java. Even more, this particular system has earned a lot of critique about its verbosity and lack of expressiveness [Hejlsberg, 2003], [van Dooren and Steegmans, 2005], which in turn influenced language designers not to put effect systems for exceptions into new languages [Hejlsberg, 2003], [Odersky, 2013].

In this dissertation, we study the problems that hinder the adoption of effect systems in mainstream languages and propose new ideas that address these issues effectively. The high-level goal is to design a practical effect system with the right balance between nota-



tional and conceptual overhead, simplicity and expressive power. We believe that one of the necessary ingredients of a successful effect system is a lightweight syntax for expressing effect-polymorphic functions. Methods whose effect depends on their arguments are ubiquitous in both object-oriented and functional programming languages, as we show later on.

The proposed system is not only an effect checker that is given “as is” to programmers, but rather an extensible framework for effect checking in which new kinds of side effects, so called effect domains, can be integrated easily. In addition, the system is optional and can be easily disabled: effect annotations are simply ignored when effect checking is not enabled, which leaves programmers with the choice of using the tool or not. In this thesis, we discuss a number of concrete effect domains, including IO, checked exceptions and state modifications, which are available in the implementation for the Scala language.

The flexibility and generality of the Scala programming language and its platform makes it possible to implement the effect system without changing the language specification or the compiler. The effect system is implemented as a compiler plugin and effect annotations are expressed as standard type annotations which are simply ignored by the Scala compiler if the compiler plugin is not enabled. The effect annotations for effect-polymorphism and for state effects make extensive use of dependent types. Finally, refinement types are used to track the effects of anonymous functions and classes, which is essential for checking effect-polymorphic code.

## 1.1 Overview

The four following chapters in this dissertation describe different aspects of the effect systems presented in this thesis. Chapter 2 starts by formally introducing type-and-effect systems and motivating the need for lightweight effect-polymorphism. The effect system presented in this chapter uses a simple strategy for expressing effect-polymorphic functions which does not make use of explicit effect type parameters. We show that effect-polymorphism is independent of specific effect domains by embedding the language into a framework for effect checking which is extensible to new effect domains. The soundness proof is parametrized by monotonicity lemmas that are required to hold for each effect domain. This means that proving soundness of the system when adding a new effect domain does not require an inductive proof.

In Chapter 3 we introduce *relative effect annotations*, a simple scheme for annotating effect-polymorphism based on dependent types that scales well in object-oriented languages. While relative effects are intuitive and easy to write and understand, they are also expressive enough to capture common higher-order code patterns such as those found in the Scala collections library.

One of the most important effect domains is state modifications, however it is also notoriously

difficult to handle because state can be arbitrarily aliased in general. The effect system for purity presented in Chapter 4 is capable of expressing common programming patterns that use local state without exposing any observable side effects, like the use of an iterator or building a collection using a buffer. At the same time the type system remains remarkably simple to use and understand.

Finally, in Chapter 5 we show how the effect system is integrated into Scala and evaluate the expressiveness and annotation overhead using core examples from the Scala library. We explain the internals of the compiler plugin and show how effect checking is integrated into the type checking process.

## 1.2 Contributions

This thesis proposes solutions to several problems which make it difficult to integrate type-and-effect systems into impure programming languages. The main contributions of this thesis are the following:

- We introduce a lightweight annotation system for expressing effect-polymorphism that is intuitive and easy to use. We show that it can describe the behavior of common higher-order programming patterns.
- We present a generic effect system with lightweight effect-polymorphism that can check multiple effect domains at the same time, an extension of the work by Marino and Millstein [2009] that is discussed in Section 1.3.1. We show that practical default effects for non-annotated methods reduce the annotation overhead and at the same time facilitate integrating the system into existing languages.
- We show that dependent types are a concise and natural method to express effect annotations for certain effect systems. Our effect system makes use of dependent types to express effect-polymorphism and state modification effects.
- We present a new effect system for purity with respect to state modification that builds on existing work by Pearce [2011] and that can express purity of higher-order code which is common in functional languages with nested definitions like Scala. The modularity and flow-insensitivity of the effect system allow it to be integrated as an effect domain into the generic framework for effect checking.
- We implemented the generic effect system and the effect domains for IO, exceptions and purity as a compiler plugin for the Scala language. The implementation works with the official release of the Scala compiler and did not require changes to the Scala language since effects are expressed as standard annotations. The effect system supports local effect inference in the same way that Scala supports local type inference. Compiler plugins that are integrated into the type checking process can be enabled in the *presentation*

*compiler* mode of the Scala compiler which is used for interactive error reporting in IDEs. Programmers receive immediate feedback about side effects “as they type” which makes effect checking a helpful tool *during* development, in addition to the verification when compiling the source code.

### 1.3 Related Work

In this section we review existing techniques for controlling side effects that are used in either pure or impure languages. The discussion of existing work related to the topics covered in the individual chapters of this thesis is deferred to those chapters.

#### 1.3.1 Type-and-Effect Systems

*Type-and-effect systems* were originally designed by Gifford and Lucassen [1986] and later extended to support effect and region polymorphism by Lucassen and Gifford [1988] as a means to track effects on memory locations. The basic idea is to separate the store into regions, bind state allocations to specific regions and express effects on the store in terms of those regions. Such an effect system forms the basis of the FX programming language [Gifford et al., 1992]. The technique of expressing effects as part of a function’s type has since been used in other settings, including the type systems in this thesis.

Talpin and Jouvelot [1992b] introduce subeffecting and present an inference algorithm for types, regions and effects. The work by Talpin and Jouvelot [1992a] shows that effect inference solves the problem of unsound let-polymorphism in the presence of mutable storage cells, which is usually addressed with the conservative value restriction as noted by Pierce [2002], Chapter 22.7. This inference algorithm is the basis of the work on DDC by Lippmeier [2010], an extension of Haskell with mutable state that uses call-by-value semantics for effectful parts of programs.

Later work by Tofte and Talpin [1994] shows how type, region and effect inference can be used to provide a stack based implementation for programming languages with reference allocations and updates. This system has been implemented by Tofte et al. [January 2006] in the context of the MLKit programming language.

In programming languages with global type inference like Haskell and ML, the use of an effect system with global inference is a natural choice. The effect system presented in this dissertation on the other hand has a different focus: it is designed to integrate with object-oriented and functional programming languages with subtyping and without global type inference, languages like Scala or C#. This implies that programmers will be confronted with effect annotations and should be able to write and read them with reasonable effort. In such a setting, explicit region and effect parameters for constructors and polymorphic methods

## Chapter 1. Introduction

---

are syntactically too heavyweight. But even in languages with global effect inference, the size and complexity of effect annotations is relevant because they are visible to programmers, for instance in module signatures or error messages.

There are a number of computational effects for which type-and-effect systems have been designed, including exceptions [Gosling et al., 2013], purity [Pearce, 2011], atomicity [Abadi et al., 2008] or access to widgets in graphical user interfaces [Gordon et al., 2013]. The work by Marino and Millstein [2009] factors out the commonalities of various effect systems into a generic framework which is proven to be sound for arbitrary well-behaved effect domains. The effect system presented in Chapter 2 can be seen as an extension of this work with lightweight effect-polymorphism and the ability to combine multiple effect domains. One difference is that their system features tagging of runtime values and a whole-program analysis to reconstruct which tags can flow into a function argument. Our system on the other hand is designed to work modularly on the basis of effect annotations.

Like most effect systems, the generic effect system by Marino and Millstein [2009] records effects in an unordered fashion and accordingly expresses them as sets that form a lattice. Nielson and Nielson [1999] introduce *behaviors*, a richer representation for effects which also takes into account the order in which the effects take place. The temporal information in behaviors enables modeling advanced properties such as communication protocols or resource usage contracts.

### 1.3.2 Monads

Monads were originally introduced by Moggi [1991] and popularized by Peyton Jones and Wadler [1993] as the main technique for integrating effects into the Haskell language. Despite their great success, there are two well known issues with monadic effect handling that have been described in the literature. The first issue is that source code which uses multiple kinds of side effects has to combine multiple monads, which is not straightforward, as recently illustrated by Brady [2013]. The solution to monad composition in Haskell are monad transformers [Liang et al., 1995], which often require programmers to lift operations explicitly into the resulting monad instance.

The second issue is that introducing side effects into an existing function requires refactoring that function to monadic style, and also other code that uses it has to be adapted. As described in Chapter 1 of the thesis of Lippmeier [2010], the fact that monadic and pure code have incompatible types leads to code duplication: for example the function *map* in Haskell can only apply a pure function over the elements of a list. To apply an effectful function, the language provides a second implementation *mapM*.

Wadler [1998] showed that monads are equivalent to type-and-effect systems: a lattice of effects can be represented as a lattice of monads that are connected by monad morphisms.

The producers framework by Tate [2013] is a general theory that formalizes the semantics of sequential composition of “producer” effects. Their system subsumes various generalizations of monads, for example as indexed monads, which are equivalent to classical type-and-effect systems. However, producers are more expressive than existing approaches based on monads or effect systems with an effect lattice and can encode effects like locking, which depend on the ordering of a sequence of statements.

### 1.3.3 Alternative Systems for Controlling Effects

Algebraic effects [Bauer and Pretnar, 2012] are an alternative representation for effects that can express common monadic effects such as state, IO and exceptions, but not continuations. Defining new algebraic effects is lightweight and unlike monads, combining effects is straightforward, which encourages fine-grained effect definitions. Their system does not check effects statically. This means that programs might evaluate to an undefined state where an effect operation appears outside a handler, in which case execution gets stuck.

Brady [2013] presents an implementation of algebraic effects for the IDRIS language, which does not require any changes to the host language. New effects are defined as standard data types and given semantics by creating an instance of the `Handler` type class for the new type. Effect handlers are typically implemented using monads, the system provides a flexible abstraction that does not compete with monads but builds on them. Using the support for dependent types, the type system statically ensures that functions cannot use any effects which are not available in their signature. However, there is no support for effect inference.

The most straightforward way to introduce ordering of side effects in a programming language with undefined evaluation order is by manually threading a value as parameter through functions with effects. Linear types [Wadler, 1990] make this technique robust by ensuring that this *world* parameter cannot be duplicated or discarded. The type system with uniqueness types by Barendsen and Smetsers [1993] is based on ideas from linear types and has been implemented in the Clean programming language. The main disadvantage of this approach to ordering effects is that the programmer has to manually handle the *world* parameters, which can be tedious.

Terauchi and Aiken [2005] present a similar system for ordering effects in which *witness* values are passed as additional function arguments. Since multiple witness values can be created, the system is more flexible than the linear type systems and allows arbitrary ordering of non-interfering effects. A semantic condition on the use of witness values that can be checked with a static algorithm guarantees correctness of programs.

The Koka programming language by Leijen [2012] features an effect system that can express effect-polymorphism and also functions like exception handlers that mask effects. Combining effects of multiple domains is straightforward; effects are represented as simple labels, and each function has a set of effects, which can be either annotated by the programmer or inferred.

The system does not feature subeffecting, i.e., ordering between effect labels. Therefore the effect systems presented in this thesis, which use an effect lattice, could not be directly expressed in Koka.

### 1.3.4 Program Verification

The goals of program specification and verification systems overlap to some degree with those of effect systems. Method contracts typically denote the portion of the program state that a method operates on, called the *frame*, for example using a `modifies` clause in ESC/Java [Flanagan et al., 2002] or `Spec#` [Leino and Müller, 2010]. Like in effect systems, method contracts in object-oriented languages need to restrict behavior of overriding methods in order for the system to be sound<sup>2</sup>.

Program verification has a wider scope than effect systems, at the cost of increased complexity. Specifications have to be provided by the programmer and are often non-trivial to express: Leino [2010] comments his verifiable implementation of the Schorr-Waite graph algorithm with the phrases “32 lines of quantifier-filled loop invariants can be a mouthful” and “The hardest thing in writing the program is deciphering the verifier’s error messages. That task is not yet for non-experts”.

The main difficulties in verifying framing specifications are caused by the possibility of aliasing and the desire for abstraction and information hiding. In the case of `Spec#`, consistency of data is specified using ownership annotations [Leino and Müller, 2004], a system that enables modular specifications which interact well with subclassing and enable specifications for recursive data structures. Ownership enforces global constraints on aliasing that have to be maintained by programs throughout their entire lifetime. Ownership type systems are discussed in more detail in Section 4.5.2.

To abstract over private state, verification systems often provide the possibility to define *specification variables*, also called *ghost variables*, which describe the private state of a module. Kassios [2006] defines *dynamic frames* as specification variables whose values are sets of allocated locations and expresses framing conditions in terms of these variables. In addition to the modified or accessed state, specifications can express properties like capturing existing state or allocating a fresh object in the representation. Dynamic frames can express object relations like those described by ownership annotations but are more flexible because no programming restrictions are enforced. The Dafny language by Leino [2010] uses dynamic frames and expresses memory footprints in terms of `reads`, `modifies` and `fresh` clauses.

Banerjee et al. [2008] introduce region logic, a first-order Hoare logic which expresses frame conditions in terms of first class regions. The benefit of treating regions as ghost state is that it makes the logic compatible with existing automated tools that support first-order specification

---

<sup>2</sup>ESC/Java deliberately allows overriding methods to specify an unsound “also `modifies`” clause in favor of the enhanced flexibility

languages based on classical logic. Inspired by separation logic, modular reasoning is enabled by a static analysis of the *footprint* of a formula, which is the state that the formula allows to be accessed.

Smans et al. [2009] introduce *implicit* dynamic frames, a variant of the dynamic frames approach inspired by separation logic in which frame information is inferred from access assertions in pre- and postconditions. A method can only modify an existing location if that location is required to be accessible by its precondition. This approach leads to more concise method contracts and fewer proof obligations that must be discharged by the verifier.

In his seminal paper introducing separation logic, Reynolds [2002] shows that Hoare style verification systems “suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size”. Separation logic defines a new logical operation  $P * Q$ , called the separating conjunction, which asserts that formulas  $P$  and  $Q$  hold for disjoint portions of the heap. Valid specification  $\{P\}c\{Q\}$  are required to be *tight* which means that the formula  $P$  must describe all of the heap that  $s$  needs during execution. Tight specifications enable *local reasoning*, the ability to prove specifications of code using only those memory cells that the program accesses, which is expressed through the frame rule: if  $c$  does not modify free variables of  $R$ , then  $\{P\}c\{Q\}$  implies  $\{P * R\}c\{Q * R\}$ .

Parkinson and Bierman [2008] develop techniques to apply separation logic to object-oriented programming languages with inheritance. They split up the specification of a method into a *static* part, which is used for verifying the implementation, and a *dynamic* part used for verifying dynamically bound invocations. Dynamic specifications are written in terms of *abstract predicates* that can be overridden in subclasses, which leads to modular specifications that do not need to be reverified for each subclass that inherits a method.

Separation logic is a non-classical logic which requires custom theorem provers for automated verification. Piskac et al. [2013] introduce a translation from a decidable fragment of separation logic to a first-order theory, which enables using of the advanced tools that exist for first-order logic. Their article also discusses earlier work that pursues similar goals.

In a recent article, Dinsdale-Young et al. [2013] present the “Views” framework, a metatheory that can express various program analyses like type systems and program logics. The *view* of a thread is defined as an abstract *knowledge* about the state of the machine and the *rights* to change that state. The knowledge of one thread must be immune to operations of other threads. They present encodings of various existing systems in their framework, for example simple type systems, type systems with strong updates like linear types, and program logics like separation logic.





## Chapter 2

# A Generic Framework for Polymorphic Effect-Checking

This chapter presents a generic framework for polymorphic effect checking using lightweight annotations. The framework is independent of a specific effect domain.

Support for effect-polymorphic functions is essential for the expressiveness of an effect system: higher-order functions whose behavior depends on their argument functions are ubiquitous not only in functional programming languages, but equally in object-oriented languages where they appear in the form of the common “strategy” pattern [Gamma et al., 1995] and delegation in general. One famous example is the `map` function which transforms the elements of a collection using its argument function. The side effects of an invocation of `map` depend on the function which is passed as argument: if for example a pure function is passed to `map`, then the invocation does not have any side effects.

The effect system for checked exceptions in Java has earned a lot of critique for its verbosity and limited expressiveness [Hejlsberg, 2003], [van Dooren and Steegmans, 2005]. For example, identical exception declarations often need to be copied between methods, and the verbose syntax tempts programmers to use over-approximations instead of precisely expressing the exceptions of a method. Support for effect-polymorphism is a fundamental ingredient to solve these issues. However, as shown in Section 2.2.1, it is vital that effect-polymorphism can be expressed in a lightweight fashion which is easy to use and does not require unnecessary refactorings.

Towards this goal, the effect system in this chapter introduces a new kind of function type for denoting effect-polymorphic functions. In a monomorphic function type  $T \xrightarrow{e} U$ , the effect  $e$  denotes the *latent* effect, the effect that might occur when the function is invoked. For an effect-polymorphic function type  $T \xrightarrow{e} U$ , the latent effect consists of both  $e$  and the latent effect of its argument type  $T$ .

As mentioned in Section 1.2, the system is an extension of the generic effect system by Marino and Millstein [2009] with lightweight effect-polymorphism and the ability to combine multiple effect domains. The number of effect annotations required to annotate effects of multiple domains remains manageable thanks to intelligent defaults. The effect system is shown to be sound, and soundness is preserved when instantiating it to one or multiple well-behaved effect domains.

### 2.1 Introducing Type-and-Effect Systems

Traditional type systems such as the simply typed lambda calculus described in Pierce [2002] assign to every term a type that describes the value to which the term evaluates. A *type-and-effect system*, often just called an *effect system*, in addition assigns an effect to every term. This effect describes the side effects that may occur when the term is evaluated. Type-and-effect systems were first described by Gifford and Lucassen [1986] as a mechanism to delimit the scope of read, write and allocation effects on memory locations. The fundamental techniques in their system are however not tied to effects on memory locations and have been successfully extended to other kinds of computational effects, for instance IO or exceptions.

The type-and-effect system presented in this chapter abstracts over the concrete computational effects using effect variables. When instantiating the abstract system to a concrete effect domain, soundness of the resulting effect system follows from soundness of the generic system and the well-formedness of the effect domain. In Section 2.6 we present examples of instantiations to concrete effect domains.

We introduce type-and-effect systems using STLCE, a simply typed lambda calculus with effects described in Figure 2.1. Every function type  $T_1 \xrightarrow{e} T_2$  has an effect annotation  $e$  describing the effect that may occur when executing the function, called the *latent* effect. In a concrete effect system, the effect annotation on a function type would describe for instance the state that the function modifies, or the exceptions that the function might throw. STLCE represents effects as sets of atomic effects for simplicity. We introduce a refined representation for abstract effects in Section 2.1.2.

Figure 2.1 also presents the inference rules for STLCE. The typing statement of the form  $\Gamma \vdash t : T ! e$  assigns a type  $T$  and an effect  $e$  to term  $t$ . We first observe that the typing rules are completely standard modulo effect inference and effect annotations in function types.

In rule T-ABS, the latent effect of the function type assigned to the lambda abstraction is the effect inferred by typing the function body. The function abstraction itself has no effect: as explained in Chapter 1 we assume a call-by-value evaluation strategy which does not reduce nested redexes. Therefore, the term under the lambda abstraction is not reduced until the function is applied to an argument. We say that the effect of  $t$  is *delayed* by the function abstraction. Purity of a lambda abstraction can also be explained by the fact that lambda

## 2.1. Introducing Type-and-Effect Systems

$t ::= x$ $\quad   t t$ $\quad   v$ $v ::= (x : T) \Rightarrow t$	variable function application value function abstraction		$T ::= T \xRightarrow{e} T$ $e ::= \emptyset   e \cup e$ $\Gamma ::= \overline{x : T}$	function type effect typing environment
--	---	--	--	---

$\Gamma \vdash t : T ! e$

$\text{T-VAR} \frac{x : T \in \Gamma}{\Gamma \vdash x : T ! \emptyset}$	$\text{T-ABS} \frac{\Gamma, x : T_1 \vdash t : T_2 ! e}{\Gamma \vdash (x : T_1) \Rightarrow t : T_1 \xRightarrow{e} T_2 ! \emptyset}$
$\text{T-APP} \frac{\Gamma \vdash t_1 : T_1 \xRightarrow{e} T_2 ! e_1 \quad \Gamma \vdash t_2 : T_1 ! e_2}{\Gamma \vdash t_1 t_2 : T_2 ! e_1 \cup e_2 \cup e}$	

$t \longrightarrow t'$

$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2}$	$\frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2}$	$\frac{}{((x : T) \Rightarrow t) v \longrightarrow [v/x]t}$
---	---	---

Figure 2.1: Simply Typed Lambda Calculus with Effects (STLCE)

abstractions are values and therefore do not take any evaluation steps.

Typing rule T-APP computes the effect of a function invocation as the union of three effects: the effect  $e_1$  of evaluating the function term, the effect  $e_2$  of evaluating the argument term, and the latent effect  $e$  of the function type assigned to  $t_1$ . As an example, imagine  $t_1$  is an expression which writes to a log and returns a function that might throw an exception, and  $t_2$  is a term that reads from the console.

```
let t1 = { log "t1"; (x: String) ⇒ stringToInt x }
let t2 = readLine ()
```

The evaluation of the program “ $t_1 t_2$ ” has all three side effects: it writes to the log, it reads from the console, and it may throw an exception.

### 2.1.1 Effects Have “May” Semantics

Type-and-effect systems typically have “may” semantics for effect annotations, i.e., the effect of a term inferred by the type system *may* occur when the term is evaluated, but it does not have to. We illustrate this by extending STLCE with conditionals (the definition of booleans is omitted):

$$t ::= \dots | \text{if } t t t \quad \left| \quad \text{T-IF} \frac{\Gamma \vdash t_c : \text{Bool} ! e_c \quad \Gamma \vdash t_1 : T ! e_1 \quad \Gamma \vdash t_2 : T ! e_2}{\Gamma \vdash \text{if } t_c t_1 t_2 : T ! e_c \cup e_1 \cup e_2}$$

$$\frac{t_c \longrightarrow t'_c}{\text{if } t_c \ t_1 \ t_2 \longrightarrow \text{if } t'_c \ t_1 \ t_2} \quad \frac{}{\text{if true } t_1 \ t_2 \longrightarrow t_1} \quad \frac{}{\text{if false } t_1 \ t_2 \longrightarrow t_2}$$

Typing rule T-IF includes the effects of both branches in the resulting effect, even though only one of the two terms will be evaluated. For instance, the following function is assigned the effect of producing an error:

```
(x: Int) ⇒ (y: Int) ⇒ if (y == 0) (error "div by zero") (x / y)
```

Since the type-and-effect systems studied in this dissertation adopt “may” semantics for their effect annotations, they define an upper bound on the effects of a term, but they cannot express a lower bound. This restriction simplifies the effect system and the abstract representation of effects, as explained in Section 2.1.2.

There are some effect systems that cannot be expressed in our framework. For example, an effect system that ensures a locking strategy needs to be able to ensure that a locking effect *has* occurred in order to type check a critical section. Existing systems that support such advanced semantics for effects can be found for instance in the work on behaviors by Nielson and Nielson [1999] or in the recent work on producer effects by Tate [2013].

### 2.1.2 A Generic Representation for Effects

In order to design a generic effect system that can represent multiple kinds of side effects, we need to define an abstract representation for effects on which the system can operate. The choice of this representation has a direct impact on the expressiveness of the generic effect system. For instance, if the effect representation in the generic effect system does not register the order in which effects occur, an effect system that depends on this property cannot be expressed as an instance of the generic one.

In the framework presented in this chapter, the effects of an effect domain  $\mathcal{D}$  are represented as a semi-lattice consisting of the following elements:

- A set of atomic effects  $E_{\mathcal{D}}$
- A join operation computing the combination of two effects,  $\sqcup_{\mathcal{D}} : E_{\mathcal{D}} \times E_{\mathcal{D}} \Rightarrow E_{\mathcal{D}}$
- A sub-effect relation comparing two effects,  $\sqsubseteq_{\mathcal{D}} \subseteq (E_{\mathcal{D}} \times E_{\mathcal{D}})$
- A bottom element  $\perp_{\mathcal{D}} \in E_{\mathcal{D}}$  that satisfies  $\forall e \in E_{\mathcal{D}}. \perp_{\mathcal{D}} \sqsubseteq_{\mathcal{D}} e$  denoting purity, i.e., the absence of effects
- A top element  $\top_{\mathcal{D}} \in E_{\mathcal{D}}$  that satisfies  $\forall e \in E_{\mathcal{D}}. e \sqsubseteq_{\mathcal{D}} \top_{\mathcal{D}}$  denoting impurity, i.e., allows arbitrary effects, also called the *unknown* effect

The join operator should be commutative, associative and idempotent. Note that the sub-effect relation is implicitly defined by the join operation:  $e_1 \sqsubseteq_{\mathcal{D}} e_2$  holds if and only if  $e_1 \sqcup_{\mathcal{D}} e_2 = e_2$ . The reason we still include it when defining effect lattices is that it usually has a more efficient implementation which avoids building the join of the two effects. An explicit sub-effect relation also helps to clarify the presentation of an effect domain.

The consequence of representing effect domains as simple lattices is that flow-sensitive systems cannot be encoded in the generic effect system: the order in which effects happen is not recorded. Instances of the generic system also cannot distinguish if an effect happens only once or multiple times.

Richer representations for effects that also take into account the order in which effects take place are described for instance by Nielson and Nielson [1999] or Tate [2013]. However the added expressiveness for effects comes at the cost of additional complexity in the effect system. Annotations for effect-polymorphism would need to encode how the abstracted effect is embedded in the remaining effect of the function. Kneuss et al. [2013] do this by expressing effects as control flow graph summaries. Such effect annotations are verbose and difficult to write and understand. To make effect systems practical for every day programming, we found that classical “may” semantics with a lattice-based representation for effects is a good compromise between simplicity and the ability to model a relevant amount of the behavior of programs.

As an example of a concrete effect domain we present the lattice of an effect system tracking IO effects. It is the simplest possible effect system because there are only two states: every term either has the IO effect or not.

- Effect set  $E_{\mathcal{D}} = \{noIo, io\}$  with  $\perp_{\mathcal{D}} = noIo$  and  $\top_{\mathcal{D}} = io$
- Join operation  $e_1 \sqcup_{\mathcal{D}} e_2 = \begin{cases} io & \text{if } io \in \{e_1, e_2\} \\ noIo & \text{otherwise} \end{cases}$
- Sub-effect relation  $e_1 \sqsubseteq_{\mathcal{D}} e_2 = (e_1 = noIo) \vee (e_2 = io)$

As mentioned before, a valid effect lattice needs to adhere to the common lattice properties. Showing validity of the IO lattice is straightforward and therefore omitted.

## 2.2 Effect-Polymorphism

The effect system of STLCE introduced in Section 2.1 only supports monomorphic effects: the effect of invoking a function is always the same, no matter what arguments are passed into it.

In reality there are many examples where the effect of a function depends on the effects of its arguments. A famous example is the higher-order function `map` which transforms the elements

## Chapter 2. A Generic Framework for Polymorphic Effect-Checking

---

of a collection using an argument function. The effect of an invocation of `map` depends on the effect of the function which is passed to it.

The higher-order function  $h$  in the following example takes as argument a function with an arbitrary effect and invokes it:

$$\text{let } h = (f: \text{Int} \stackrel{\top}{\Rightarrow} \text{Int}) \Rightarrow f \ 1$$

In STLCe the function  $h$  has type  $(\text{Int} \stackrel{\top}{\Rightarrow} \text{Int}) \stackrel{\top}{\Rightarrow} \text{Int}$ , so the invocation  $h \text{ increment}$  where a pure function is passed to  $h$  is type checked to have effect  $\top$ , even though no effects occur at run-time<sup>1</sup>.

A common solution to add support for effect-polymorphism, as first described by Lucassen and Gifford [1988], is to extend the type system with effect type parameters. The function  $h$  would be written as

$$\text{let } h = [e :: \text{Effect}] \Rightarrow (f: \text{Int} \stackrel{e}{\Rightarrow} \text{Int}) \Rightarrow f \ 1$$

where  $e$  is a type parameter of kind *Effect*. The type  $[e :: \text{Effect}] \Rightarrow (\text{Int} \stackrel{e}{\Rightarrow} \text{Int}) \stackrel{e}{\Rightarrow} \text{Int}$  describes that the effect of  $h$  depends on the effect of its function parameter. The invocation “ $h \perp \text{ increment}$ ” instantiates the effect parameter to  $\perp$  and is therefore type checked as pure.

In the case of checked exceptions in Java, the effect types coincide with the types of values and therefore normal type parametrization can be used to express effect parametrization. The following example shows a list interface in Java with a monomorphic and an effect-polymorphic version of the `map` method:

```
public interface Function<T, U> {
    public U apply(T t) throws Exception;
}

public interface FunctionE<T, U, E extends Exception> {
    public U apply(T t) throws E;
}

public interface List<T> {
    public <U> List<U> map(Function <T, U> f) throws Exception;
    public <U, E extends Exception> List<U> mapE(FunctionE<T, U, E> f) throws E;
}
```

The monomorphic method `map` can only accept functions with arbitrary effects as argument if it declares the top effect, i.e., `throws Exception`. The second version is polymorphic in the exception type of its argument function. Note that adding an effect type parameter to `map` is not

---

<sup>1</sup>This example requires a type system with subtyping that allows using a pure function where an effectful function is expected. Extending STLCe with subtyping is straightforward and analogous to subtyping of the effect system in Section 2.5.1.

enough: the Function type also has to be extended with an explicit exception type parameter.

### 2.2.1 The Need for Lightweight Syntax

The previous example shows that explicit parametrization for effect-polymorphism results in code which is syntactically heavy and hard to understand. Another problem is that polymorphism is tied to one effect domain: adding a new effect system to Java would lead to additional effect type parameters.

For languages without global type inference like Scala or Java, the readability and brevity of declarations are of crucial importance for programmers. For this reason, effect-polymorphism in Java as shown in the previous example is rarely used in practice, even though higher-order code such as the strategy pattern is very common.

Checked exceptions in Java have been criticized for their verbosity and limited expressiveness, for instance by Hejlsberg [2003]. Interestingly, the two mentioned issues are related: writing effect-polymorphic code in Java is extremely verbose as shown by the previous example. For this reason developers often chose the more concise monomorphic exception declarations which do not express the behavior of the program precisely.

On the one hand, we believe that effect-polymorphism is indispensable to the success of an effect system. On the other hand, it is essential that writing effect-polymorphic functions is straightforward and lightweight so that the question of using polymorphism or not is not a question of tradeoffs for developers.

This chapter introduces a new pragmatic way to express effect-polymorphic code: writing an effect-polymorphic function does not require explicit effect parameters and is as lightweight as writing an ordinary, monomorphic function. Polymorphic functions are directly supported in the type system by a new kind of function type, introduced in the following section.

### 2.2.2 Effect-Polymorphic Function Types

The generic type-and-effect system presented in this chapter introduces a new kind of function type which, by definition, denotes effect-polymorphic functions. To differentiate between effect-polymorphic and ordinary, effect-monomorphic functions, we use two kinds of arrows in function types.

The double arrow  $\Rightarrow$  is used for ordinary function types and has the same meaning as in the type system for STLCe. If the effect annotation is omitted, the largest possible effect  $\top$  is assumed.

An effect-polymorphic function type is expressed with a simple arrow  $\rightarrow$ . Like ordinary function types, also polymorphic function types are annotated with a latent effect, however

## Chapter 2. A Generic Framework for Polymorphic Effect-Checking

---

the default effect is  $\perp$  when the annotation is omitted.

A simple effect-polymorphic function  $h$  is therefore written as:

```
let  $h = (f: \text{Int} \Rightarrow \text{Int}) \rightarrow f \ 1$ 
```

The type of  $h$  is  $(\text{Int} \Rightarrow \text{Int}) \rightarrow \text{Int}$ , which is equivalent to  $(\text{Int} \overset{\top}{\Rightarrow} \text{Int}) \overset{\perp}{\rightarrow} \text{Int}$  given the default effects defined above. The crucial property of an effect-polymorphic function type is that its latent effect consist of two components:

- The annotated effect, an effect that may occur when the function is invoked, independently of the argument. In the example of  $h$ , this effect is  $\perp$ .
- The effect of the argument.

For each invocation of a polymorphic function, the effect of the argument can be different, which results in a different overall effect. To illustrate this we define a pure function  $f$  and an effectful function  $g$ :

```
let  $f: (\text{Int} \overset{\perp}{\Rightarrow} \text{Int}) = (x: \text{Int}) \Rightarrow x + 1$   
let  $g: (\text{Int} \overset{\text{throws}(ex)}{\Longrightarrow} \text{Int}) = \text{throw } ex$ 
```

For each invocation of the function  $h$  the effect is computed based on the actual argument type. The invocation  $h \ f$  has no effect, while the invocation  $h \ g$  has the effect of throwing an exception.

To illustrate that the type system is able to express the effects of non-trivial functions, the following example shows an effect-polymorphic *map* function:

```
let  $map: \text{IntList} \overset{\perp}{\Rightarrow} (\text{Int} \Rightarrow \text{Int}) \rightarrow \text{IntList} =$   
  ( $l: \text{IntList}$ )  $\Rightarrow (f: \text{Int} \Rightarrow \text{Int}) \rightarrow l \ \text{match} \{$   
    case Nil  $\Rightarrow$  Nil  
    case Cons  $x \ xs \Rightarrow$  Cons  $(f \ x) (map \ xs \ f)$   
  }
```

Even though the signature of the *map* function is fully effect-polymorphic, there is only a single effect annotation  $\perp$ , which denotes purity of the outer function.

An even simpler system for effect-polymorphism could be designed using only a single kind of function type, but treating every higher-order function as effect-polymorphic. While this system would work for the *map* function from the previous example, Section 3.1 shows that effect-polymorphism is not necessarily tied to parameters of function type. Furthermore, we argue in Section 3.3.1 that effect-polymorphism is an important aspect of the documentation of a function and should therefore require an explicit annotation. For these reasons the system studied in this chapter uses a specific type for effect-polymorphic functions.



## 2.3 Abstracting Over Effect Domains

In Section 2.1.2 we define an abstract representation for effects in the form of a semilattice. In order to make the framework extensible to multiple effect domains, we additionally want to give each concrete effect system the possibility of customizing the effect of evaluating a term. For instance, an effect system for tracking exceptions declares that a throw expression introduces an effect and that a try expression can mask, or eliminate, effects.

In our framework, this information is provided in the form of a function  $eff_{\mathcal{D}}$  which returns the effect of evaluating a term, given the effects of its subterms. This function is closely related to the “adjust” function in the generic type-and-effect system of Marino and Millstein [2009].

The  $eff_{\mathcal{D}}$  function takes two arguments: a name indicating the syntactic form in question, and a list of effects of its subterms. By default it combines all the argument effects using the  $\sqcup_{\mathcal{D}}$  operator:

$$eff_{\mathcal{D}}(*, \bar{e}) = \bigsqcup_{\mathcal{D}} e_i$$

The default  $eff_{\mathcal{D}}$  function can be specialized by concrete effect domains. For instance in the domain of exceptions  $\mathcal{E}$  introduced in Section 2.6, the definition  $eff_{\mathcal{E}}(\text{THROW}(p)) = \text{throws}(p)$  assigns an effect to throw statements.

In the domain of IO, effects are introduced by calling pre-defined functions that have a latent *io* effect. There are no syntactic constructs that introduce or mask IO effects, therefore the  $eff_{\mathcal{D}}$  function is left unspecified and the framework will use the default definition.

### Monotonicity of $eff_{\mathcal{D}}$

The type system presented in Section 2.5 uses the  $eff_{\mathcal{D}}$  function in each inference rule to compute the effect of a term. In order for the type system to be sound,  $eff_{\mathcal{D}}$  needs to meet the following monotonicity requirement:

#### Lemma 2.1. Monotonicity.

For every effect domain  $\mathcal{D}$  and every syntactic form TRM,

1. if  $\forall e'_i \in \bar{e}', e_i \in \bar{e}. e'_i \sqsubseteq_{\mathcal{D}} e_i$ , then  $eff_{\mathcal{D}}(\text{TRM}, \bar{e}') \sqsubseteq_{\mathcal{D}} eff_{\mathcal{D}}(\text{TRM}, \bar{e})$ , and
2.  $eff_{\mathcal{D}}(\text{TRM}, e_1, \dots, e_{i1} \sqcup_{\mathcal{D}} e_{i2}, \dots, e_n) \sqsubseteq_{\mathcal{D}} eff_{\mathcal{D}}(\text{TRM}, e_1, \dots, e_{i1}, \dots, e_n) \sqcup_{\mathcal{D}} e_{i2}$

The first clause of the monotonicity lemma requires the  $eff_{\mathcal{D}}$  function to be monotonic in the sub-effect relation  $\sqsubseteq_{\mathcal{D}}$ . This means that given a term  $t$  of the form TRM, when replacing one of its subterms  $t_s$  with a subterm  $t'_s$  that has a smaller effect, the effect of  $t$  cannot grow.

Implementing effect masking remains possible, as we show in the effect domain of exceptions. The second part of the monotonicity lemma prevents the effect of a term to depend on the presence of a certain effect in its subterms. This restriction falls in line with the “may” semantics of the type-and-effect system explained in Section 2.1.1.

### 2.4 Combining Multiple Effect Domains

In order to check multiple kinds of effects at the same time, the effect lattices of each individual domain are combined into one multi-domain effect lattice. The elements of this combined lattice are tuples of effects from the individual domains:

$$E = \{e_{\mathcal{D}_1} \dots e_{\mathcal{D}_n} \mid e_{\mathcal{D}_i} \in E_{\mathcal{D}_i}\}$$

The  $\sqcup$ ,  $\sqsubseteq$  and  $eff$  operations are defined element-wise using the respective operation for each domain:

$$\begin{aligned} (e'_{\mathcal{D}_1} \dots e'_{\mathcal{D}_n}) \sqcup (e_{\mathcal{D}_1} \dots e_{\mathcal{D}_n}) &= (e'_{\mathcal{D}_1} \sqcup_{\mathcal{D}_1} e_{\mathcal{D}_1}) \dots (e'_{\mathcal{D}_n} \sqcup_{\mathcal{D}_n} e_{\mathcal{D}_n}) \\ (e'_{\mathcal{D}_1} \dots e'_{\mathcal{D}_n}) \sqsubseteq (e_{\mathcal{D}_1} \dots e_{\mathcal{D}_n}) &\iff (e'_{\mathcal{D}_1} \sqsubseteq_{\mathcal{D}_1} e_{\mathcal{D}_1}) \wedge \dots \wedge (e'_{\mathcal{D}_n} \sqsubseteq_{\mathcal{D}_n} e_{\mathcal{D}_n}) \\ eff(\text{TRM}, \overline{e_{\mathcal{D}_1} \dots e_{\mathcal{D}_n}}) &= eff_{\mathcal{D}_1}(\text{TRM}, \overline{e_{\mathcal{D}_1}}) \dots eff_{\mathcal{D}_n}(\text{TRM}, \overline{e_{\mathcal{D}_n}}) \end{aligned}$$

The top and bottom elements are defined as  $\top = (\top_{\mathcal{D}_1} \dots \top_{\mathcal{D}_n})$  and  $\perp = (\perp_{\mathcal{D}_1} \dots \perp_{\mathcal{D}_n})$ .

Deriving the Monotonicity Lemma 2.1 for the combined effect domain is straightforward given monotonicity of each individual domain.

#### 2.4.1 Annotating Multiple Effect Domains

When tracking effects from multiple domains the effect annotations on function types have to declare an effect for every domain that is being checked. For instance, if there are three effect domains  $\mathcal{D}_1$ ,  $\mathcal{D}_2$  and  $\mathcal{D}_3$ , a function type has the form  $T_1 \xrightarrow{e_{\mathcal{D}_1} e_{\mathcal{D}_2} e_{\mathcal{D}_3}} T_2$ .

Clearly this annotation scheme does not scale very well: effect annotations quickly become long and are hard to maintain. When adding a new effect domain, the annotation in every function type needs to be updated. Our system implements a simple solution to address this problem: it is based on the observation that in many cases, function types have either no effect, a small number of effects, or the topmost effect. To backup this claim we look at a few examples:

- Higher-order function like `map` typically accept as argument a function that can have any effect. Therefore, this argument is annotated with the topmost effect.

- The implementation of `map` itself is pure, there is no other effect than the effect of its argument.
- Pure functions are extremely common, for instance many operations on immutable datatypes are pure. The inference tool for JPure by Pearce [2011] found 40% of the methods in the Java standard library to be pure with respect to state modifications. Huang et al. [2012] show similar results for other libraries written in Java. Since Scala encourages immutable data structures and a functional style, the amount of purity is expected to be comparable or higher.
- Functions that do have side effects usually have effects in one or few effect-domains. For instance, a random generator is non-deterministic, operations on mutable data structures modify state, or functions from a file-API have IO-effects. In addition, these functions might have exceptional behavior. However, it is rather uncommon to have functions with side effects from many domains at the same time.

In order to simplify the multi-domain effect annotations we account for the above observations and introduce two specific effect annotations which, by definition, range over all effect domains:  $\top$  and  $\perp$ . When used as such, the two annotations have the expected meaning:  $\top = \top_{\mathcal{D}_1} \dots \top_{\mathcal{D}_N}$ , similarly for  $\perp$ .

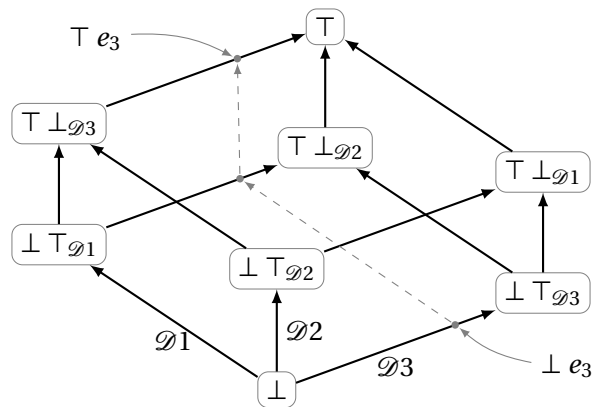


Figure 2.2: Effect Annotations in Multiple Domains

However, the crucial characteristic is that the multi-domain annotations can be combined with concrete effect annotations from individual effect domains. For instance, the type  $T_1 \xrightarrow{\perp e_{\mathcal{D}_i}} T_2$  denotes a function which may have effect  $e_{\mathcal{D}_i}$  in the domain  $\mathcal{D}_i$ , but is pure in all other domains. Similarly, combining the  $\top$  annotation with concrete effects restricts the allowed effect in certain domains. This behavior is illustrated in Figure 2.2, showing an example with three effect domains.

$t ::= x$	variable
$t t$	application
$v$	value
$v ::= (x : T) \Rightarrow t$	monomorphic abstraction
$(x : T) \rightarrow t$	effect-polymorphic abstraction
$T ::= T \xRightarrow{e} T$	function type
$T \xrightarrow{e} T$	effect-polymorphic function type
$e ::= \perp e_D \mid \top e_D \mid e_D$	effect annotation
$e_D ::= e_D e_D \mid \cdot$	concrete effects
$\Gamma ::= \overline{x : T}$	parameter context
$f ::= \epsilon \mid x$	polymorphism context

Figure 2.3: Language with Polymorphic Effects (LPE)

## 2.5 Static Semantics

This section presents a formalization of the effect system outlined in the previous section. It extends the type system of STLCE introduced in Section 2.1 by adding effect-polymorphic function types and subtyping. In Figure 2.3 we summarize the syntax of the formal language, which we name LPE for “Lightweight Polymorphic Effects”.

As introduced in Section 2.2.2, there are two kinds of functions: ordinary, monomorphic functions denoted using the double arrow  $\Rightarrow$ , and effect-polymorphic functions denoted with a simple arrow  $\rightarrow$ . The two kinds of arrows appear in function abstraction terms and in function types.

In a monomorphic function type  $T_1 \xRightarrow{e} T_2$  the latent effect is described by  $e$ . The effect of an effect-polymorphic function type  $T_1 \xrightarrow{e} T_2$  consists of two parts: the annotated effect  $e$  and the effect of its argument of type  $T_1$ . Only higher-order functions, which are functions that take another function as argument, can be effect-polymorphic. As explained in Section 2.5.2, this invariant is checked by the typing rule T-ABS-POLY which enforces the parameter type  $T_1$  to be a function type. The effect of the argument function is implicitly added to the total effect of an effect-polymorphic function.

The default effects for function types without effect annotations are explained in Section 2.2.2:  $T_1 \Rightarrow T_2$  is a equivalent to  $T_1 \xRightarrow{\top} T_2$ , and  $T_1 \rightarrow T_2$  is a equivalent to  $T_1 \xrightarrow{\perp} T_2$ .

### 2.5.1 Subtyping

The subtyping relation of our calculus is reflexive and transitive.

$$\text{S-REFL} \frac{}{T <: T} \quad \text{S-TRANS} \frac{T' <: S \quad S <: T}{T' <: T}$$

The subtyping rules covering the two kinds of function types in our system are identically structured.

$$\text{S-FUN-MONO} \frac{T_1 <: T'_1 \quad T'_2 <: T_2 \quad e' \sqsubseteq e}{T'_1 \xRightarrow{e'} T'_2 <: T_1 \xRightarrow{e} T_2} \quad \text{S-FUN-POLY} \frac{T_1 <: T'_1 \quad T'_2 <: T_2 \quad e' \sqsubseteq e}{T'_1 \xrightarrow{e'} T'_2 <: T_1 \xrightarrow{e} T_2}$$

In S-FUN-MONO, a function with a latent effect  $e'$  can only be a subtype of another function with effect  $e$  if  $e' \sqsubseteq e$ . As an example, we define a higher-order function that requires its argument to be pure:

$$\text{let } \text{pureHof} = (\text{f}: \text{Int} \xRightarrow{\perp} \text{Int}) \Rightarrow \text{f } 1$$

The subtyping rule will only allow pure functions to be passed into *pureHof*.

When comparing two effect-polymorphic function types in S-FUN-POLY, remember that we defined previously the latent effect of  $T_1 \xrightarrow{e} T_2$  to consist of two parts: the annotated effect  $e$  plus the latent effect of the parameter type  $T_1$ . This raises the question why the subtyping rule for polymorphic function types only compares the annotated effects. Assume we have two functions:

$$\begin{aligned} \text{let } \text{maybePure}: (\text{Int} \xRightarrow{\top} \text{Int}) \xRightarrow{\perp} \text{Int} = \dots \\ \text{let } \text{pure}: (\text{Int} \xRightarrow{\perp} \text{Int}) \xRightarrow{\perp} \text{Int} = \dots \end{aligned}$$

In general, an invocation of *maybePure* might have any effect, while an invocation of *pure* is always pure. However, the subtyping relation seems to contradict this observation: due to contra-variance of parameters, the type of *maybePure* is a subtype of the type of *pure*. To build an intuition why the subtyping rule is correct, we take a closer look at the two function types.

The type of *pure* says: “Give me a pure function from Int to Int, and I compute a result without producing a side effect.” For instance, in the body of a function  $m$

$$\text{let } m = (\text{pure}: (\text{Int} \xRightarrow{\perp} \text{Int}) \xRightarrow{\perp} \text{Int}) \Rightarrow \dots$$

the parameter function *pure* only accepts pure functions. Now assume that we use the function *maybePure* where a function of the type of *pure* is expected, e.g.,

$$m \text{ maybePure}$$

As explained before, this is allowed by the subtyping rules. The example shows that it is also sound, because in the body of method  $m$  only pure functions will be passed into *maybePure*. Due to effect-polymorphism, those invocations of *maybePure* have no effect.

## Chapter 2. A Generic Framework for Polymorphic Effect-Checking

---

In other words, the type of the function *maybePure* says: “If you give me a pure function, I *also* compute a result without producing a side effect!”

### 2.5.2 Typing Rules

Terms are assigned a type and an effect using a judgment of the form  $\Gamma; f \vdash t : T ! e$  where  $\Gamma$  maps variables to their type. The additional environment variable  $f$  is used for type checking effect-polymorphic methods. While its exact role is discussed later, remember for now that it holds either a parameter  $x \in \Gamma$ , or the special value  $\epsilon$  which is distinct from all parameter names.

Referencing a parameter does not have a side effect:

$$\text{T-VAR} \frac{x : T \in \Gamma}{\Gamma; f \vdash x : T ! \perp}$$

Next, we look at the typing rules for monomorphic function abstraction and application.

$$\text{T-ABS-MONO} \frac{\Gamma, x : T_1; \epsilon \vdash t : T_2 ! e}{\Gamma; f \vdash (x : T_1) \Rightarrow t : T_1 \xrightarrow{e} T_2 ! \perp}$$

$$\text{T-APP-MONO} \frac{\Gamma; f \vdash t_1 : T_1 \xrightarrow{e} T ! e_1 \quad \Gamma; f \vdash t_2 : T_2 ! e_2 \quad T_2 < T_1}{\Gamma; f \vdash t_1 t_2 : T ! \text{eff}(\text{APP}, e_1, e_2, e)}$$

The typing rule for abstraction infers the result type  $T_2$  and the latent effect  $e$  of a function. By using the value  $\epsilon$  in the environment for type checking the function body, we propagate the information that the term belongs to a monomorphic function.

The rule T-APP-MONO is a standard typing rule for function applications. The effects of the subterms and the latent effect of the function are combined using the *eff* function introduced in Section 2.3, which by default computes the join of its argument effects.

Before analyzing the typing rules for effect-polymorphic function abstractions and invocations, we take a look at the functionality of the extended typing environment  $\Gamma; f$ .

Suppose we are analyzing the effect of a simple effect-polymorphic function

$$\text{val } hof = (f : \text{Int} \xRightarrow{\top} \text{Int}) \rightarrow f \ 1$$

The computed type should be  $(\text{Int} \xRightarrow{\top} \text{Int}) \xrightarrow{\perp} \text{Int}$ , i.e., the function *hof* itself has effect  $\perp$ . The effect of invoking  $f$  can be ignored because it is already expressed in the function type through effect-polymorphism.

To achieve this special treatment of the argument function, the parameter  $f$  is placed in the extended environment as  $\Gamma; f$  when type checking the function body of an effect-polymorphic function.

$$\text{T-ABS-POLY} \frac{T_1 = T_a \xRightarrow{e_1} T_b \quad \Gamma, f : T_1; f \vdash t : T_2 ! e}{\Gamma; f' \vdash (f : T_1) \rightarrow t : T_1 \xrightarrow{e} T_2 ! \perp}$$

Note that the typing rule forces the argument type  $T_1$  to be a monomorphic function type, since only higher-order functions can be effect-polymorphic. We later explain why the argument function can only be monomorphic.

The following typing rule T-APP-PARAM implements the mentioned special treatment of the argument function  $f$ :

$$\text{T-APP-PARAM} \frac{f : T_1 \xRightarrow{e} T \in \Gamma \quad \Gamma; f \vdash t : T_2 ! e_2 \quad T_2 <: T_1}{\Gamma; f \vdash f t : T ! \text{eff}(\text{APP}, \perp, e_2, \perp)}$$

When applying a function  $f$  which is the parameter of an enclosing effect-polymorphic function, the latent effect of  $f$  is not taken into account.

The last element of the static semantics is the typing rule for invocations of effect-polymorphic functions.

$$\text{T-APP-POLY} \frac{\Gamma; f \vdash t_1 : T_1 \xrightarrow{e} T ! e_1 \quad \Gamma; f \vdash t_2 : T_2 ! e_2 \quad T_2 <: T_1}{\Gamma; f \vdash t_1 t_2 : T ! \text{eff}(\text{APP}, e_1, e_2, e \sqcup \text{latent}(T_2))}$$

There is a single but crucial difference between this rule and the rule T-APP-MONO for monomorphic function applications. The latent effect of the function  $t_1$  now consists of two components: the concrete effect  $e$  annotated in the function type, and the latent effect of the argument function  $t_2$  which is computed using  $\text{latent}(T_2)$ .

The rule T-APP-POLY plays a central role in the effect-polymorphic type system: it computes the effect of the actual argument type  $T_2$  for each invocation of  $t_1$ . Through subtyping this effect might be smaller than the effect of the function's parameter type  $T_1$ .

The parameter type  $T_1$  is known to be a monomorphic function type: this is enforced by the typing rule T-ABS-POLY. Since  $T_2 <: T_1$ , we know that also  $T_2$  is a monomorphic function type. Therefore, the auxiliary function computing the latent effect is simply defined as

$$\text{latent}(T_1 \xRightarrow{e} T_2) = e$$

The reason why only monomorphic functions are allowed as parameters of effect-polymorphic

$t ::= \dots$	
$\text{throw}(p)$	throwing an exception
$\text{try } t \text{ catch}(\overline{p}) t$	catching and handling exceptions
$T ::= \dots$	
$\text{Nothing}$	bottom type
$e_D ::= \dots$	
$\text{throws}(\overline{p})$	exception effect annotation
$p ::= p_1 \mid \dots \mid p_n$	exceptions

Figure 2.4: Extended Syntax for LPE with Exceptions

functions is that the typing rules become simpler without decreasing the expressiveness. Imagine that a polymorphic function takes another polymorphic function as argument:

```
val highHof = (f: (Int => Int) -> Int) -> f ((x: Int) => x + 1)
```

When looking at the type of *highHof*, the information that its argument *f* is applied to a *pure* is not represented in the signature. Therefore, there is no advantage in allowing effect-polymorphic functions as arguments.

Enforcing parameter types of effect-polymorphic functions to be monomorphic function types slightly simplifies the soundness proof for the type system. However, we assume that lifting this restriction would not render the type system unsound.

## 2.6 Examples of Concrete Effect Domains

This section presents two extensions of the core calculus that implement effect checking for concrete effect domains. Both extensions are orthogonal to the mechanisms for effect-polymorphism in the base language. Every concrete effect system that is added to the framework profits from effect-polymorphism without any additional effort: the extensions would be exactly the same in a monomorphic effect checking framework.

### 2.6.1 Exceptions

In order to add effect checking for exceptions, we first extend the base language with primitives that express the throwing and handling of exceptions. The additions to the language syntax are presented in Figure 2.4. We use a finite set of exceptions  $p_1 \dots p_n$  that can be thrown and caught, however the system could be easily extended to an open hierarchy of effects such as the exception types in languages like Scala or Java. An effect annotation  $\text{throws}(\overline{p})$  denotes that any of the exceptions in  $\overline{p}$  might be thrown. The effect lattice for the exception domain  $\mathcal{E}$



is defined in Figure 2.5.

$$\begin{array}{ll}
 E_{\mathcal{E}} = \{\text{throws}(\bar{p}) \mid \bar{p} \subseteq \{p_1, \dots, p_n\}\} & \text{throws}(\bar{p}) \sqsubseteq_{\mathcal{E}} \text{throws}(\bar{q}) \iff p \subseteq q \\
 \perp_{\mathcal{E}} = \text{throws}() & \text{throws}(\bar{p}) \sqcup_{\mathcal{E}} \text{throws}(\bar{q}) = \text{throws}(\bar{p} \cup \bar{q}) \\
 \top_{\mathcal{E}} = \text{throws}(p_1, \dots, p_n) & 
 \end{array}$$

Figure 2.5: Effect Lattice for Exceptions

To give a valid type to the throw primitive, we introduce the bottom type `Nothing` which is a subtype of every other type.

$$\text{S-NOTHING} \frac{}{\text{Nothing} <: T}$$

The typing rules for the two new syntactic forms are defined as follows:

$$\begin{array}{c}
 \text{T-THROW} \frac{}{\Gamma; f \vdash \text{throw}(p) : \text{Nothing}! \text{eff}(\text{THROW}(p))} \\
 \\
 \text{T-TRY} \frac{\begin{array}{l} \Gamma; f \vdash t_1 : T_1! e_1 \quad T_1 <: T \quad e_t = \text{eff}(\text{TRY}, e_1) \\ \Gamma; f \vdash t_2 : T_2! e_2 \quad T_2 <: T \quad e = \text{eff}(\text{CATCH}(\bar{p}), e_t, e_2) \end{array}}{\Gamma; f \vdash \text{try } t_1 \text{ catch}(\bar{p}) t_2 : T! e}
 \end{array}$$

Note that the type system for LPE does not have a subsumption rule. As noted in [Pierce, 2002, Chapter 16.4], adding a bottom type to such a type system requires an additional inference rule for function invocations when the function term is erroneous.

$$\text{T-APP-E} \frac{\Gamma; f \vdash t_1 : \text{Nothing}! e_1 \quad \Gamma; f \vdash t_2 : T_2! e_2}{\Gamma; f \vdash t_1 t_2 : \text{Nothing}! \text{eff}(\text{APP}, e_1, e_2, \perp)}$$

Finally, to complete the description of the new effect domain we have to inform the framework that throw expressions can add effects, while try expressions can mask effects. This is achieved by defining the function  $\text{eff}_{\mathcal{E}}$ :

$$\begin{array}{ll}
 \text{eff}_{\mathcal{E}}(\text{THROW}(p)) & = \text{throws}(p) \\
 \text{eff}_{\mathcal{E}}(\text{TRY}, e) & = e \\
 \text{eff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), e_1, e_2) & = \text{throws}((\bar{q} \setminus \bar{p}) \cup \bar{s}) \quad \text{where } e_1 = \text{throws}(\bar{q}) \\
 & \quad e_2 = \text{throws}(\bar{s})
 \end{array}$$

### Monotonicity of $\text{eff}_{\mathcal{E}}$

To ensure soundness of the effect domain we need to show that the monotonicity Lemma 2.1 holds for the definition of  $\text{eff}_{\mathcal{E}}$  given above.

## Chapter 2. A Generic Framework for Polymorphic Effect-Checking

---

*Proof (Monotonicity for  $\text{eff}_{\mathcal{E}}$ ).* For terms of the form TRY the function  $\text{eff}_{\mathcal{E}}$  is equivalent to the default  $\text{eff}_{\mathcal{D}}$  for which monotonicity is straightforward. In the case of THROW(P),  $\text{eff}_{\mathcal{E}}$  does not take any arguments, so monotonicity holds trivially. It remains to show monotonicity of  $\text{eff}_{\mathcal{E}}$  for terms of the form CATCH( $\bar{p}$ ).

For the first clause of the monotonicity lemma we need to show that  $e'_q \sqsubseteq_{\mathcal{E}} e_q \wedge e'_s \sqsubseteq_{\mathcal{E}} e_s$  implies  $\text{eff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), e'_q, e'_s) \sqsubseteq_{\mathcal{E}} \text{eff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), e_q, e_s)$ . If we take  $e'_q = \text{throws}(\bar{q}')$ ,  $e'_s = \text{throws}(\bar{s}')$ , similarly for  $e_q$  and  $e_s$ , and expand the definitions of  $\text{eff}_{\mathcal{E}}$  and  $\sqsubseteq_{\mathcal{E}}$  we obtain the goal

$$(\bar{q}' \setminus \bar{p}) \sqcup \bar{s}' \subseteq (\bar{q} \setminus \bar{p}) \sqcup \bar{s}$$

which can be shown using basic set theory.

For the second clause we need to show

$$\begin{aligned} \text{eff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), e_{q1} \sqcup_{\mathcal{E}} e_{q2}, e_s) &\sqsubseteq_{\mathcal{E}} \text{eff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), e_{q1}, e_s) \sqcup_{\mathcal{E}} e_{q2} \quad \text{and} \\ \text{eff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), e_q, e_{s1} \sqcup_{\mathcal{E}} e_{s2}) &\sqsubseteq_{\mathcal{E}} \text{eff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), e_q, e_{s1}) \sqcup_{\mathcal{E}} e_{s2} \end{aligned}$$

We define  $e_{q1} = \text{throws}(\bar{q}_1)$ ,  $e_s = \text{throws}(\bar{s})$ , etc., and obtain by expanding  $\text{eff}_{\mathcal{E}}$  and  $\sqsubseteq_{\mathcal{E}}$

$$\begin{aligned} ((\bar{q}_1 \cup \bar{q}_2) \setminus \bar{p}) \cup \bar{s} &\subseteq (\bar{q}_1 \setminus \bar{p}) \cup \bar{s} \cup \bar{q}_2 \quad \text{and} \\ (\bar{q} \setminus \bar{p}) \cup (\bar{s}_1 \cup \bar{s}_2) &\subseteq (\bar{q} \setminus \bar{p}) \cup \bar{s}_1 \cup \bar{s}_2 \end{aligned}$$

which are again straightforward to show. □

### 2.6.2 Asynchronous Operations

Most popular programming languages, including C# [Hejlsberg, 2010], F# [Syme et al., 2011], Scala [Haller et al., 2012] and Java [Lea, 2000], support constructs to start asynchronous computations and returning a “future”, i.e., a handle that allows to retrieve the result once the computation completes. For example in Scala, an asynchronous computation can be started as follows:

```
val f = future {
  // potentially long-running computation
}
```

When using the handle  $f$  to retrieve the result, the current thread blocks until the value is computed, or an unhandled exception is thrown:

```
val r = Await.result(f, Duration.Inf)
```

```

 $t ::= \dots$ 
  | future  $t$     asynchronous expression
  | block       blocking expression
  | blocking  $t$  delimiting blocking expression

 $e_D ::= \dots$ 
  | B | noB     blocking / non-blocking effect annotation

```

Figure 2.6: Extended Syntax for LPE with Exceptions

For efficiency the runtime environment typically uses a thread pool for executing the body of a future instead of creating a new thread [Lea, 2000]. However, when running on a fixed-size thread pool, calling blocking operations within the body of the future is problematic: it may lead to starvation and, in the worst case, even to a deadlock of the entire thread pool [Haller and Odersky, 2009]. Blocking occurs for example when awaiting the result of a future or when performing synchronous IO.

The API documentation for the ForkJoinTask class in the Java standard library [Oracle, 2013a] clearly states this restriction:

“Computations should avoid synchronized methods or blocks, and should minimize other blocking synchronization [...]. Tasks should also not perform blocking IO [...]”

In order to use a potentially blocking operation within a future, the programmer has to wrap it so that the runtime system can temporarily re-size the thread pool if necessary. In Scala this is written as follows:

```

val f = future {
  blocking {
    // potentially blocking code
  }
}

```

Using a specific effect system it is possible to enforce wrapping of potentially blocking operations within futures at compile-time. This eliminates an entire class of concurrency errors when using thread pools. The additions to the language syntax are presented in Figure 2.6. For simplicity there is only a single blocking expression `block`; in practice, many more expressions would be potentially blocking, e.g., functions for synchronization or blocking IO. The `future  $t$`  expression asynchronously runs expression  $t$  which must be non-blocking, i.e., pure in this effect system. The `blocking  $t$`  expression wraps a potentially blocking expression  $t$  masking the effect of the wrapped expression.

## Chapter 2. A Generic Framework for Polymorphic Effect-Checking

---

The fact that an expression is blocking is denoted by the effect annotation  $B$ . The effect lattice is equivalent to the one for IO effects presented in Section 2.1.2, therefore its definition is omitted.

The typing rules for the three new syntactic forms are defined as follows:

$$\begin{array}{c} \text{T-FUTURE} \frac{\Gamma; e \vdash t : T!e \quad e = \text{noB}}{\Gamma; f \vdash \text{future } t : T! \text{eff}(\text{FUTURE}, e)} \\ \\ \text{T-BLOCK} \frac{}{\Gamma; f \vdash \text{block} : T! \text{eff}(\text{BLOCK})} \\ \\ \text{T-BLOCKING} \frac{\Gamma; f \vdash t : T!e}{\Gamma; f \vdash \text{blocking } t : T! \text{eff}(\text{BLOCKING}, e)} \end{array}$$

The typing rule T-FUTURE enforces the term  $t$  to be non-blocking by requiring its effect to be  $\text{noB}$ . To complete the description of the new effect domain we define the  $\text{eff}_{\mathcal{B}}$  as:

$$\begin{array}{lcl} \text{eff}_{\mathcal{B}}(\text{FUTURE}, e) & = & e \\ \text{eff}_{\mathcal{B}}(\text{BLOCK}) & = & B \\ \text{eff}_{\mathcal{B}}(\text{BLOCKING}, e_1) & = & \text{noB} \end{array}$$

The  $\text{eff}_{\mathcal{B}}$  function expresses the fact that block expressions add a blocking effect, while blocking expressions mask a blocking effect.

## 2.7 Dynamic Semantics

In order to model the runtime behavior of LPE, we define a big-step operational semantics. A term  $t$  reduces in one step to either a value  $v$  or an error  $\text{throw}(p)$ , written  $t \downarrow \langle r, S \rangle$  where  $r ::= v \mid \text{throw}(p)$ . The set  $S$  contains the effects that occurred while evaluating the term. Every element  $e \in S$  is an atomic effect, i.e.,  $S \subseteq E$  where  $E$  is the multi-domain effect lattice defined in Section 2.4.

### 2.7.1 Extensible Effect Domains

Similar to the typing judgments, the evaluation rules are parametrized by an auxiliary function  $\text{dynEff}$  which computes the effect of evaluating a term based on the effects of its subterms. It is closely related to function  $\text{eff}$ , but it operates on sets of effects instead of atomic effects. The reason is that in contrast to the static semantics, the operational semantics does not approximate the occurrence of two distinct atomic effects by their join, but keeps both effects in the resulting set  $S$ .

By default the  $\text{dynEff}_{\mathcal{D}}$  function for an effect domain  $\mathcal{D}$  joins its argument effect sets:

$$\text{dynEff}_{\mathcal{D}}(*, \bar{S}) = \bigcup_i S_i$$

In the case of exceptions, the function  $\text{dynEff}_{\mathcal{E}}$  is defined as follows:

$$\begin{aligned} \text{dynEff}_{\mathcal{E}}(\text{THROW}(p)) &= \{\text{throws}(p)\} \\ \text{dynEff}_{\mathcal{E}}(\text{TRY}, S) &= S \\ \text{dynEff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), S_1, S_2) &= (S_1 \setminus \{\text{throws}(p_i) \mid p_i \in \bar{p}\}) \cup S_2 \end{aligned}$$

The actual  $\text{dynEff}$  function used in the dynamic semantics works on the multi-domain effect lattice introduced in Section 2.4. Remember that each set of dynamic effects is a subset of the multi-domain effect lattice,  $S \subseteq E = \{e_{\mathcal{D}_1} \dots e_{\mathcal{D}_n} \mid e_{\mathcal{D}_i} \in E_{\mathcal{D}_i}\}$ . In order to invoke the domain-specific  $\text{dynEff}_{\mathcal{D}}$  functions the tuples in this set need to be projected to the corresponding domain:

$$\text{dynEff}(\text{TRM}, \bar{S}) = \{e_{\mathcal{D}_1} \dots e_{\mathcal{D}_n} \mid e_{\mathcal{D}_i} \in \text{dynEff}_{\mathcal{D}_i}(\text{TRM}, \overline{\text{proj}_i(S)})\}$$

where  $\text{proj}_i(S) = \{e_{\mathcal{D}_i} \mid (e_{\mathcal{D}_1} \dots e_{\mathcal{D}_n}) \in S\}$ .

In order for the type system to be sound, the  $\text{eff}$  function needs to model  $\text{dynEff}$  conservatively and correctly. This requirement is explained in Section 2.8.1.

### 2.7.2 Evaluation Rules

When evaluating an application, if one of the two terms evaluates to  $\text{throw}(p)$  for some exception  $p$ , then so does the entire expression.

$$\begin{array}{c} \text{E-APP-E1} \frac{t_1 \downarrow \langle \text{throw}(p), S_1 \rangle}{t_1 \ t_2 \downarrow \langle \text{throw}(p), S \rangle} \\ \text{E-APP-E2} \frac{t_1 \downarrow \langle v_1, S_1 \rangle \quad t_2 \downarrow \langle \text{throw}(p), S_2 \rangle}{t_1 \ t_2 \downarrow \langle \text{throw}(p), S \rangle} \end{array}$$

In the evaluation rule for applications, we write  $[v/x]t$  for the term  $t$  with all occurrences of the variable  $x$  replaced by value  $v$ . We use the special arrow  $\mapsto$  to range over both, effect-polymorphic and monomorphic functions.

$$\text{E-APP} \frac{t_1 \downarrow \langle (x : T) \mapsto t, S_1 \rangle \quad t_2 \downarrow \langle v_2, S_2 \rangle}{t_1 \ t_2 \downarrow \langle r, S \rangle} \quad [v_2/x]t \downarrow \langle r, S_1 \rangle \quad S = \text{dynEff}(\text{APP}, S_1, S_2, S_1)$$

A throw expression does not evaluate, but the evaluation rule still computes the set of dynamic effects of the expression.

$$\text{E-THROW} \frac{S = \text{dynEff}(\text{THROW}(p))}{\text{throw}(p) \downarrow \langle \text{throw}(p), S \rangle}$$

The evaluation of a try-catch expression depends on the result obtained for the first subterm  $t_1$ . In case it evaluates to an error  $\text{throw}(p)$  and the exception  $p$  is handled by the catch clause, then the final result is the evaluation of the handler  $t_2$ .

$$\text{E-TRY-E} \frac{\begin{array}{l} t_1 \downarrow \langle \text{throw}(p), S_1 \rangle \quad p \in \bar{p} \\ t_2 \downarrow \langle r_2, S_2 \rangle \quad S_t = \text{dynEff}(\text{TRY}, S_1) \\ S = \text{dynEff}(\text{CATCH}(\bar{p}), S_t, S_2) \end{array}}{\text{try } t_1 \text{ catch}(\bar{p}) t_2 \downarrow \langle r_2, S \rangle}$$

The last evaluation rule applies to try-catch expressions in which the evaluation of  $t_1$  either does not raise an exception, or it raises an exception that is not handled by the  $\text{catch}(\bar{p})$  clause. In this case, the obtained result  $r_1$ , which might be an error, is propagated.

$$\text{E-TRY} \frac{t_1 \downarrow \langle r_1, S_1 \rangle \quad S = \text{dynEff}(\text{TRY}, S_1)}{\text{try } t_1 \text{ catch}(\bar{p}) t_2 \downarrow \langle r_1, S \rangle}$$

## 2.8 Effect Soundness

In this section we state two important theorems for the soundness of the type system presented in Section 2.5.2 with respect to the dynamic semantics introduced in the previous section.

In the static semantics, every expression has an effect  $e$ , while in the dynamic semantics, the evaluation of an expression yields a *set* of effects  $S$ . For notational convenience, we write  $S \leq e$  to express that every effect in  $S$  is smaller than  $e$ , i.e.,  $\forall e_s \in S. e_s \sqsubseteq e$ .

**Theorem 2.1.** Preservation.

If  $\Gamma; f \vdash t : T ! e$  is a valid typing statement for term  $t$  and the term evaluates as  $t \downarrow \langle r, S \rangle$ , then there is a valid typing statement  $\Gamma; f \vdash r : T' ! e'$  for  $r$  with  $T' < T$ .

**Theorem 2.2.** Effect soundness.

If  $\Gamma; f \vdash t : T ! e$  is a valid typing statement for term  $t$  and the term evaluates as  $t \downarrow \langle r, S \rangle$ , then  $S \leq e \sqcup \text{latent}(\Gamma(f))$ .

The effect soundness theorem states that every effect that occurs when evaluating a term  $t$  is represented in the typing derivation for  $t$ . Remember that in the typing rule for effect-polymorphic functions, T-ABS-POLY, the argument function  $f$  is propagated in the extended environment  $\Gamma; f$ . Invocations of  $f$  are thereafter treated as pure by typing rule T-APP-PARAM. Therefore, given a typing statement  $\Gamma; f \vdash t : T ! e$ , the effect that might occur when evaluating  $t$  consists of  $e$  and the latent effect of  $f$ ,  $\text{latent}(\Gamma(f))$ .

### 2.8.1 Consistency Requirement

In both semantics, we use an auxiliary function to compute the effect that occurs when evaluating a term. The preservation and soundness theorems are based on the assumption that the static *eff* function conservatively models the behavior of the *dynEff* function in the operational semantics.

This requirement is expressed through the following consistency lemma, which has to be verified for every effect domain.

**Lemma 2.2.** Consistency.

For every syntactic form TRM, list of dynamic effects  $\bar{S}$ , list of static effects  $\bar{e}$  and typing environment  $\Gamma; f$ ,

if  $\forall i. S_i \leq_{\mathcal{D}} e_i \sqcup_{\mathcal{D}} \text{latent}(\Gamma(f))$  then  $\text{dynEff}_{\mathcal{D}}(\text{TRM}, \bar{S}) \leq_{\mathcal{D}} \text{eff}_{\mathcal{D}}(\text{TRM}, \bar{e}) \sqcup_{\mathcal{D}} \text{latent}(\Gamma(f))$ .

*Proof (Consistency for  $\text{dynEff}_{\mathcal{D}}$ ).* For the default  $\text{dynEff}_{\mathcal{D}}$  and  $\text{eff}_{\mathcal{D}}$  functions we obtain

$$S = \text{dynEff}_{\mathcal{D}}(*, \bar{S}) = \bigcup_i S_i \quad (1)$$

$$e = \text{eff}_{\mathcal{D}}(*, \bar{e}) = \bigcup_i e_i \quad (2)$$

The goal is to show  $S \leq e \sqcup_{\mathcal{D}} \text{latent}(\Gamma(f))$ , which is equivalent to  $\forall e_s \in S. e_s \sqsubseteq_{\mathcal{D}} e \sqcup_{\mathcal{D}} \text{latent}(\Gamma(f))$ .

$$\begin{aligned} e_s \in S &\iff \exists i. e_s \in S_i && \text{by (1)} \\ &\implies \exists i. e_s \sqsubseteq_{\mathcal{D}} e_i \sqcup_{\mathcal{D}} \text{latent}(\Gamma(f)) && \text{by precondition } S_i \leq_{\mathcal{D}} e_i \sqcup_{\mathcal{D}} \text{latent}(\Gamma(f)) \\ &\implies e_s \sqsubseteq_{\mathcal{D}} e \sqcup_{\mathcal{D}} \text{latent}(\Gamma(f)) && \text{by (2)} \end{aligned}$$

□

*Proof (Consistency for  $\text{dynEff}_{\mathcal{E}}$ ).* For the domain of exceptions, the case THROW( $p$ ) is trivial, and the case TRY is covered by the default case shown above. For terms of the form CATCH( $\bar{p}$ ) we obtain

$$S = \text{dynEff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), S_1, S_2) = (S_1 \setminus \{\text{throws}(p_i) \mid p_i \in \bar{p}\}) \cup S_2 \quad (1)$$

$$e = \text{eff}_{\mathcal{E}}(\text{CATCH}(\bar{p}), \text{throws}(\bar{q}), \text{throws}(\bar{s})) = \text{throws}((\bar{q} \setminus \bar{p}) \cup \bar{s}) \quad (2)$$

The preconditions of the consistency lemma are

$$\forall e_s \in S_1. e_s \sqsubseteq_{\mathcal{E}} \text{throws}(\bar{q}) \sqcup_{\mathcal{E}} \text{latent}(\Gamma(f)) \quad (3)$$

$$\forall e_s \in S_2. e_s \sqsubseteq_{\mathcal{E}} \text{throws}(\bar{s}) \sqcup_{\mathcal{E}} \text{latent}(\Gamma(f)) \quad (4)$$

The goal is to show  $\forall e_s \in S. e_s \sqsubseteq e \sqcup \text{latent}(\Gamma(f))$ . Since  $S$  is a union of two sets there are two cases, from which we investigate the more challenging one. In step (5) we exploit the fact that dynamic effects  $e_s \in S$  are atomic. In the case of exceptions this means that  $e_s$  has the form  $\text{throws}(p_s)$  where  $p_s$  is a single exception.

$$\begin{aligned}
e_s &\in S_1 \setminus \{\text{throws}(p_i) \mid p_i \in \bar{p}\} \\
&\implies e_s \in S_1 \wedge e_s = \text{throws}(p_s) \wedge p_s \notin \bar{p} && (5) \\
&\implies e_s \sqsubseteq_{\mathcal{E}} \text{throws}(\bar{q}) \sqcup_{\mathcal{E}} \text{latent}(\Gamma(f)) && \text{by (3)} \\
&\implies e_s \sqsubseteq_{\mathcal{E}} \text{throws}(\bar{q} \setminus \bar{p}) \sqcup_{\mathcal{E}} \text{latent}(\Gamma(f)) && \text{since } p_s \notin \bar{p} \\
&\implies e_s \sqsubseteq_{\mathcal{E}} \text{throws}((\bar{q} \setminus \bar{p}) \cup \bar{s}) \sqcup_{\mathcal{E}} \text{latent}(\Gamma(f)) && \text{weakening}
\end{aligned}$$

□

**Multiple Domains** The consistency lemma holds for the multi-domain *dynEff* function introduced in Section 2.7.1, given consistency of *dynEff*<sub>∅</sub> for individual domains. The proof is straightforward and therefore omitted.

### 2.8.2 Soundness Proofs

This section sketches the proof for the soundness theorem. The full proofs for preservation and soundness can be found in Appendix A.

In addition to preservation, the proof for soundness uses a lemma showing that effects are preserved in typing statements under value substitution. This lemma comes in two flavors: for monomorphic and effect-polymorphic abstractions.

**Lemma 2.3.** Preservation under substitution for monomorphic abstractions.

If  $\Gamma, x : T_1; f \vdash t : T ! e_l$ ,  $f \neq x$  and  $\Gamma; g \vdash v : T_2 ! \perp$  with  $T_2 < T_1$ , then  $\Gamma; f \vdash [v/x]t : T' ! e'_l$  such that  $T' < T$  and  $e'_l \sqsubseteq e_l$ .

**Lemma 2.4.** Preservation under substitution for polymorphic abstractions.

If  $\Gamma, x : T_1; x \vdash t : T ! e_l$  and  $\Gamma; g \vdash v : T_2 ! \perp$  with  $T_2 < T_1$ , then  $\Gamma; \epsilon \vdash [v/x]t : T' ! e'_l$  such that  $T' < T$  and  $e'_l \sqsubseteq e_l \sqcup \text{latent}(T_2)$ .

The two lemmas state that the type and the effect of term  $t$  decrease when a free variable in  $t$  is replaced by a value with a conforming type.

*Proof (soundness for invocations of polymorphic functions).* The proof of Theorem 2.2 is carried out using induction on the evaluation rules for a term  $t$ . We look at the most interesting case E-APP that produces the following derivations:

$$\begin{aligned}
t &= t_1 t_2 \\
t_1 &\downarrow \langle (x : T'_1) \mapsto t_{11}, S_1 \rangle \\
t_2 &\downarrow \langle v_2, S_2 \rangle \\
[v_2/x]t_1 &\downarrow \langle r, S_l \rangle \\
S &= \text{dynEff}(\text{APP}, S_1, S_2, S_l)
\end{aligned}$$



There are multiple typing rules for type checking an application expression. We investigate the key case T-APP-POLY and obtain the following sub-derivations:

$$\begin{aligned} \Gamma; f \vdash t_1 : T_1 &\xrightarrow{e_1} T ! e_1 \\ \Gamma; f \vdash t_2 : T_2 &! e_2 \\ T_2 &<: T_1 \\ e &= \text{eff}(\text{APP}, e_1, e_2, e_1 \sqcup \text{latent}(T_2)) \end{aligned}$$

The goal is to show that in environment  $\Gamma; f$ , the static effect  $e$  correctly approximates the dynamic effects  $S$ , i.e.,  $S \leq e \sqcup \text{latent}(\Gamma(f))$ .

We see that  $t_1$  evaluates to a function abstraction. The preservation theorem states that the type of this resulting function is a subtype of  $t_1$ 's original type  $T_1 \xrightarrow{e_1} T$ . Since the value is a function abstraction, the subtyping rules restrict the type to be a polymorphic function type. Looking at the canonical forms, we observe that the value can only be a polymorphic function abstraction  $(x : T'_1) \rightarrow t_{11}$ , and we obtain the following typing derivation:

$$\Gamma, x : T'_1; x \vdash t_{11} : T' ! e'_1 \quad \text{with } T_1 <: T'_1, T' <: T \text{ and } e'_1 \sqsubseteq e_1$$

Applying preservation to the term  $t_2$ , we obtain  $v_2 : T'_2$  with  $T'_2 <: T_2$ . Using transitivity of subtyping, we obtain  $T'_2 <: T'_1$  and apply the substitution Lemma 2.4 to obtain

$$\Gamma; \epsilon \vdash [v_2/x]t_{11} : T'' ! e''_1 \quad \text{with } T'' <: T' \text{ and } e''_1 \sqsubseteq e'_1 \sqcup \text{latent}(T'_2)$$

By applying the induction hypothesis on the subterm  $[v_2/x]t_{11}$ , we obtain

$$\begin{aligned} S_1 &\leq e''_1 \sqcup \text{latent}(\Gamma(\epsilon)) \\ S_1 &\leq e'_1 \sqcup \text{latent}(T'_2) \quad \text{by } e''_1 \sqsubseteq e'_1 \sqcup \text{latent}(T'_2) \text{ and } \text{latent}(\Gamma(\epsilon)) = \perp \end{aligned}$$

Since  $T'_2 <: T_2$  we can easily verify that  $\text{latent}(T'_2) \sqsubseteq \text{latent}(T_2)$ . Together with the induction hypotheses on  $t_1$  and  $t_2$ , we now have the necessary conditions to apply the consistency Lemma 2.2:

$$\begin{aligned} S_1 &\leq e_1 \sqcup \text{latent}(\Gamma(f)) \\ S_2 &\leq e_2 \sqcup \text{latent}(\Gamma(f)) \\ S_1 &\leq e_1 \sqcup \text{latent}(T_2) \end{aligned}$$

We obtain the desired result:

$$\text{dynEff}(\text{APP}, S_1, S_2, S_1) \leq \text{eff}(\text{APP}, e_1, e_2, e_1 \sqcup \text{latent}(T_2)) \sqcup \text{latent}(\Gamma(f))$$

□

## 2.9 Conclusion

In this chapter we presented an extensible framework for polymorphic effect checking where multiple effect domains can be integrated modularly. As discussed in Section 1.3.1, the closest existing work related to this framework is the generic effect system by Marino and Millstein [2009]. Our system introduces a lightweight syntax for denoting effect-polymorphic functions and shows that the treatment of effect-polymorphic functions is sound and independent of

## Chapter 2. A Generic Framework for Polymorphic Effect-Checking

---

specific effect domains.

The effect system has certain limitations in expressiveness which are due to the simplistic annotation scheme for effect-polymorphic functions. Concretely, effect inference can be imprecise when the system is used either for curried functions, for functions with multiple parameters or in object-oriented programming languages where argument objects hold a large number of member methods. These issues are discussed and addressed in the next chapter.

## Chapter 3

# Dependent Types for Relative Effects Declarations

This chapter describes a type-and-effect system for lightweight effect-polymorphism which generalizes the type system of LPE presented in Chapter 2. The system in this chapter unifies the function types for effect-polymorphic and monomorphic functions into a single kind of function type.

Effect-polymorphism is expressed using *relative effect annotations* on function types: each function declares a list of parameters which contribute to its effect. For instance, the type  $(f : T \rightarrow T) \xrightarrow{f} T$  describes a higher order function which has exactly the effect of its argument function  $f$ .

For functions with multiple arguments, relative effect annotations allow programmers to define precisely in *which* of its arguments the function is effect-polymorphic. Methods that take multiple functions as arguments are ubiquitous in object-oriented languages: every object passed as argument to a method carries a potentially large number of member methods. Relative effect annotations are a lightweight way to express effect-polymorphism that integrates well with object-oriented languages. Specifically, annotating a method as effect-polymorphic does not require changes to the parameter type. For example, the signature of a higher-order method is written “ $(f : T \Rightarrow T) : T @\text{pure}(f)$ ”, which states that the method is pure up to the effect of its parameter  $f$ . The parameter type “ $T \Rightarrow T$ ” would be the same if the higher-order method was not polymorphic.

We implemented the type system with relative effect declarations for Scala in the form of a compiler plugin and applied it to the Scala collections library which makes extensive use of higher-order functions and effect-polymorphism.

### 3.1 Overview

In object-oriented programming languages every object carries a potentially large number of methods, which naturally leads to higher-order programming patterns. In his essay explaining differences between objects and data types, William R. Cook [2009] notes that

“the typical object-oriented program makes far more use of higher-order values than many functional programs.”

In the type system of the LPE language described in Chapter 2, methods are marked as either effect-polymorphic or not. When applying this type system to an object-oriented language, we need to define what it means for a function to be effect-polymorphic in its arguments. One possibility is to define that an effect-polymorphic function with a parameter `o` has the effect of a specific member method of object `o`, for instance `o.apply`. However, this solution is not flexible enough in practice: there are many examples of effect-polymorphic functions where the effect depends on a different member than `apply`, as we will see later in this section.

A different solution is to define that an effect-polymorphic function has the effect of all member methods of the function’s parameters. In many situations this coarse annotation system is problematic:

```
def m(a: A, f: A => B): B @effPoly = f(a)
```

If method `m` is marked effect-polymorphic, then each invocation of `m` implicitly has the effect of function `f`, but also the effect of all members of object `a`. The signature of method `m` does not reveal that no methods of `a` are invoked.

To overcome this issue we could define that effect-polymorphism only applies to parameters that have a function type. However, this solution is unsystematic and there are many examples of effect-polymorphism where the parameter type is not a function. For example, the method `println` has the IO effect and the effect of invoking `toString` on its argument. The argument object is typically not a function, it can be of any type.

Another limitation of LPE is that it cannot express effect-polymorphism of curried functions: functions can only be polymorphic in their own parameter, but not in a parameter of an outer function. Curried functions are not common in Scala because methods may have multiple parameters, however the same issue appears in the form of nested definitions:

```
def m(f: () => A): A @effPoly = {  
  def nested(): A /* @effPoly-in-f */ = f()  
  nested()  
}
```

This pattern is common in the implementation of the Scala collections library as show in Section 3.3.3.

### 3.1.1 Relative Effect Declarations

Relative effect annotations represent a clean solution to the limitations outlined above while remaining syntactically lightweight and easy to understand for programmers.

Like in the type system for LPE, each function type specifies the effects that may occur when the function is invoked, e.g., IO or raised exceptions. In this chapter, that effect is called the *concrete* effect of the function. In addition to the concrete effect, each function type specifies a *relative* effect in the form of a list of parameters. Relative effects are expanded at call site according to the type of the argument that is passed for a parameter. For example, the higher-order method `m` from the previous example is annotated as follows:

```
def m(f: () => A): A @pure(f) = {
  def nested(): A @pure(f) = f()
  nested()
}
```

The methods `m` and `nested` both have a relative effect `f` and no concrete effect. As we explain in Section 3.3.1, the relative effect `f` is an abbreviation for `f.apply`. To annotate a method which is effect-polymorphic in a specific member of its parameter, the relative effect annotation takes the form of a method selection:

```
def println(o: Any): Unit @pure(o.toString) @io = { ... }
```

Before explaining the integration of relative effects into Scala in Section 3.3, the following section introduces a formal definition of the type system with relative effects and examines the subtyping and typing rules.

## 3.2 Formalization

The type system with relative effect annotations is formalized using a language called REL which is presented in Figure 3.1. REL is based on LPE, the language with lightweight polymorphic effects from Chapter 2. It uses the the same multi-domain effect lattice as LPE, which is described in Section 2.4.

The REL language differs from LPE only in the way it expresses effect-polymorphism. Instead of using a dedicated type for effect-polymorphic functions, it introduces relative effect annotations on function types. A function type has the form  $(x : T) \xrightarrow[e]{\bar{x}} T$  where  $e$  is the concrete effect of the function. A relative effect annotation  $f \in \bar{x}$  is a reference to the parameter  $f$  and

## Chapter 3. Dependent Types for Relative Effects Declarations

---

$t ::= x$	variable
$\quad   t t$	application
$\quad   v$	value
$v ::= (x : T) \xrightarrow{\bar{x}} t$	function abstraction
$T ::= (x : T) \xrightarrow{e} T$	function type
$e ::= \perp e_D \mid \top e_D \mid e_D$	effect annotation
$e_D ::= e_D e_D \mid \cdot$	concrete effects
$\Gamma ::= \overline{x : T}$	parameter context

Figure 3.1: Language with Relative Effect Annotations (REL)

expresses the fact that the function is effect-polymorphic in  $f$ . For example, the function type  $(f : \text{Int} \rightarrow \text{Int}) \xrightarrow{f} \text{Int}$  describes a higher-order function which is polymorphic in the effect of its argument  $f$ .

Because annotations for effect-polymorphism are references to parameters, the parameter names have to be retained in function types. This implies that the type system of REL uses dependent function types, however it is a restricted form of dependent types in which a type can only depend on a term in its relative effect annotations.

The system admits two syntactic simplifications. First, parameter names that do not appear in any relative effect annotation can be omitted, so that for example  $\text{Int} \rightarrow \text{Int}$  is equivalent to  $(x : \text{Int}) \rightarrow \text{Int}$  for some name  $x$ . Second, similar to the types of LPE, effect annotations on function types can be omitted in which case the following default effects are used:

- Monomorphic function types are *impure* by default:  $T_1 \rightarrow T_2$  is equivalent to  $T_1 \xrightarrow{\top} T_2$
- Effect-polymorphic function types are *pure* by default: if  $\bar{f}$  is non-empty then  $T_1 \xrightarrow{\bar{f}} T_2$  is equivalent to  $T_1 \xrightarrow{\perp} T_2$

### 3.2.1 Subtyping

The subtyping rules for REL presented in Figure 3.2 are computed with respect to an environment  $\Gamma$  which maps variables to their types. The rules for reflexivity and transitivity are standard. We discuss the subtyping rule for function types in more detail.

For the parameter types the subtyping rule S-FUN is standard, the two types are compared in contravariant fashion. The following two subsections explain how the latent effects and the result types are compared.

$$\begin{array}{c}
 \boxed{\Gamma \vdash T' <: T} \\
 \text{S-REFL} \frac{}{\Gamma \vdash T <: T} \quad \text{S-TRANS} \frac{\Gamma \vdash T' <: S \quad \Gamma \vdash S <: T}{\Gamma \vdash T' <: T} \\
 \\
 \text{S-FUN} \frac{\Gamma \vdash T_1 <: T'_1 \quad \Gamma, x : T_1 \vdash (e', [x/x']\bar{x}) \leq (e, \bar{x}) \quad \Gamma, x : T_1 \vdash [x/x']T'_2 <: T_2}{\Gamma \vdash (x' : T'_1) \xrightarrow{x'} T'_2 <: (x : T_1) \xrightarrow{x} T_2} \\
 \\
 \boxed{\Gamma \vdash (e', \bar{x}') \leq (e, \bar{x})} \\
 \frac{e' \sqsubseteq e \quad \forall f \in \bar{x}'. \Gamma \vdash f \leq (e, \bar{x})}{\Gamma \vdash (e', \bar{x}') \leq (e, \bar{x})} \\
 \\
 \frac{f \in \bar{x}}{\Gamma \vdash f \leq (e, \bar{x})} \quad \frac{\Gamma(f) = (y : T_a) \xrightarrow{e_y} T_b \quad y \notin \bar{x} \quad \Gamma, y : T_a \vdash (e_y, \bar{y}) \leq (e, \bar{x})}{\Gamma \vdash f \leq (e, \bar{x})} \\
 \\
 \boxed{[x/x']T} \\
 \frac{T = (y : T_1) \xrightarrow{e_y} T_2 \quad y \notin \{x, x'\}}{[x/x']T = (y : [x/x']T_1) \xrightarrow{[x/x']e_y} [x/x']T_2} \\
 \\
 \boxed{[x/x']\bar{x}} \\
 [x/x']\bar{x} = \bar{y} \text{ where } y_i = \begin{cases} x & \text{if } x_i = x' \\ x_i & \text{otherwise} \end{cases}
 \end{array}$$

Figure 3.2: Subtyping for REL

### Comparing Effects

The effect of a function type consists of two components: a concrete effect  $e$  and a list of relative effects  $\bar{x}$ . The relation  $(e', \bar{x}') \leq (e, \bar{x})$  compares the effects of two function types: the concrete effect  $e'$  has to be a sub-effect of the effect  $e$  and each relative effect  $f \in \bar{x}'$  has to conform to the effect pair of the supertype  $f \leq (e, \bar{x})$ .

There are two possibilities for a relative effect  $f$  to conform to the effect pair  $(e, \bar{x})$ : either  $f$  exists as a relative effect in  $\bar{x}$ , i.e.,  $f \in \bar{x}$ , otherwise the expansion of the relative effect  $f$  has to conform to  $(e, \bar{x})$ . The expansion of a relative effect  $f$  is simply the effect pair of the function type of  $f$  in  $\Gamma$ . The parameter name  $y$  in the type of  $f$  has to be distinct from the relative effects in  $\bar{x}$ , which can be ensured by applying  $\alpha$ -renaming if necessary.

Note that for each concrete effect in the subtype, an equivalent concrete effect is required in the supertype. A concrete effect in the subtype cannot conform to a relative effect in the

### Chapter 3. Dependent Types for Relative Effects Declarations

---

supertype, as illustrated by the following example:

$$\cdot \vdash (f : T \xrightarrow{e} T) \xrightarrow{e} T <: (f : T \xrightarrow{e} T) \xrightarrow{f} T \quad \text{does not hold}$$

The function type on the left always has effect  $e$ , irrespective of the function it receives as argument. The function type on the right guarantees purity in the case a pure function is passed as argument. Indeed the subtype test fails when comparing the effects of the two functions:  $(e, \cdot) \not\leq (\perp, f)$ .

When comparing function types with matching relative effects, the subtyping rule does not take the expansion of these relative effects into account, even though they might differ:

$$\cdot \vdash (f : T \xrightarrow{\top} T) \xrightarrow{f} T <: (g : T \xrightarrow{e} T) \xrightarrow{g} T$$

This example is equivalent to the one illustrating the subtyping rules of LPE in Section 2.5.1. The parameter types conform since they are checked in contravariant order. The effects of the two function types are compared as

$$g : T \xrightarrow{e} T \vdash (\perp, [g/f]f) \leq (\perp, g) \iff g : T \xrightarrow{e} T \vdash (\perp, g) \leq (\perp, g)$$

The subtyping statement therefore holds, which might seem surprising: the subtype is a function which might have arbitrary effects, while the supertype is a function with at most effect  $e$ . The explanation is the same as in Section 2.5.1: it is safe to use a function of the subtype in places where a function of the supertype is expected.

As a last example, we illustrate how a relative effect in the subtype can conform to a concrete effect in the supertype:

$$\cdot \vdash (f : T \rightarrow T) \xrightarrow{\perp} (T \rightarrow T) <: (g : T \xrightarrow{e} T) \xrightarrow{\perp} (T \xrightarrow{e} T)$$

When comparing the two result types the parameter  $f$  is replaced by  $g$  in the subtype, as discussed in the next section on subtyping dependent function types. We obtain the following sub-derivation for their effects:

$$g : T \xrightarrow{e} T \vdash (\perp, g) \leq (e, \cdot)$$

For the relative effect  $g$  in the sub-effect there is no corresponding relative effect in the super-effect. Therefore, the effect of  $g$  is expanded according to the environment to  $(e, \cdot)$ , and we obtain the derivation

$$g : T \xrightarrow{e} T \vdash (e, \cdot) \leq (e, \cdot)$$

which trivially holds. It might again seem surprising that the subtyping relation holds: the subtype can return a function with a potentially larger effect than the supertype. The situation



can be explained very similarly as in the previous example: when the subtype is used in place of the supertype, the effect of the parameter function  $g$  is restricted to  $e$ , so the resulting function can only have effect  $e$ .

**A simpler, but unsound strategy** The subtyping rule presented in this section first compares relative and concrete effects separately and then expands relative effects *only* in the subtype, but never in the supertype. A simpler strategy to compare the effects of two function types would be to directly expand all relative effects. However, this strategy is unsound, as illustrated by the following example: assume function  $h$  has type  $(f : \text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$ , hence, the invocation  $h ((x : \text{Int}) \rightarrow x)$  type checks as pure. The following (incorrect) implementation of function  $h$  would throw an exception on every invocation, even if the argument function is pure:

```
let h = (f : Int → Int) →  
        f  
    let g = (m : Int → Int) → m 1  
        f  
    g ((x : Int) → throw e)
```

The subtyping rules presented in Figure 3.2 correctly reject the invocation of  $g$ : the argument type  $\text{Int} \xrightarrow{\text{throws } e} \text{Int}$  does not conform to  $g$ 's parameter type  $\text{Int} \xrightarrow{f} \text{Int}$  because the effects do not match:  $(\text{throws } e, \cdot) \not\leq (\perp, f)$ .

A modified subtyping rule for function types which expands all relative effects would accept the invocation of function  $g$ . The parameter of  $g$  requires the effect  $f$  which would be expanded to  $\top$ . The effects would be compared as “throws  $e \sqsubseteq \top$ ” and the invocation would type check, which shows that the modified rule is unsound.

### Subtyping Dependent Function Types

The support for dependent types increases the complexity of subtyping for function types because in each function type, the result type can refer to the function's parameter. When comparing two function types, it is necessary to unify the two parameters so that references which appear in the two result types are treated as equivalent.

Type systems with dependent types and subtyping have been studied by Aspinall and Compagnoni [1996]. Our subtyping rule S-FUN is conceptually equivalent to the rule for function types in their work.

In order to compare the two result types  $T'_2$  and  $T_2$ , the variable environment is extended by  $x$ , the parameter of the supertype. All occurrences of  $x'$ , the parameter of the subtype, are eliminated from the result type  $T'_2$  and replaced by the other parameter, so the rule compares the result types as  $[x/x']T'_2 <: T_2$ . The substitution  $[x/x']T$  requires the parameter name  $y$  of

### Chapter 3. Dependent Types for Relative Effects Declarations

---

the function type  $T$  to be distinct from  $x$  and  $x'$ , which is ensured by applying  $\alpha$ -renaming if necessary.

Note that by contravariance of the parameter types, the type of  $x$ , the parameter in the supertype, is a subtype of the type of  $x'$ . For instance, if the parameters have function types, the function  $x$  has an effect which is smaller than or equal to the effect of  $x'$ . When comparing the result types, the typing rule substitutes  $x$  for  $x'$  and therefore uses the parameter type with the more precise type. The following example illustrates why this decision is correct and shows that the opposite, using the variable  $x'$  with a less precise type, leads to an unnecessarily restrictive subtyping rule.

$$\cdot \vdash (f : T \rightarrow T) \stackrel{\perp}{\rightarrow}_f (T \rightarrow T) <: (g : T \xrightarrow{e} T) \stackrel{\perp}{\rightarrow} (T \xrightarrow{e} T)$$

Intuitively, this subtyping relation should hold because it is safe to use a function with the type on the left in places where the type on the right is expected. The supertype allows passing a function with effect  $e$  as argument and it returns a function with at most effect  $e$ . The subtype satisfies these constraints: it accepts as argument functions with effect  $e$  and in this case also returns a function with at most effect  $e$ .

The subtyping rule starts by comparing the parameter types and the effects:  $T \xrightarrow{e} T <: T \rightarrow T$  holds since the default effect for monomorphic functions is  $\top$  and both function are pure. Next the result types are compared as follows:

$$g : T \xrightarrow{e} T \vdash [g/f](T \rightarrow T) <: (T \xrightarrow{e} T)$$

This subtype test produces the effect check

$$g : T \xrightarrow{e} T \vdash (\perp, g) \preceq (e, \cdot)$$

which expands to  $(e, \cdot) \preceq (e, \cdot)$  and also holds.

Assume the subtyping rule would use the parameter  $f$  of the subtype instead of  $g$  when comparing the result types. This would lead to the following subderivation:

$$f : T \rightarrow T \vdash (T \rightarrow T) <: [f/g](T \xrightarrow{e} T)$$

The resulting effect test  $f : T \rightarrow T \vdash (\top, \cdot) \preceq (e, \cdot)$  does not hold, therefore the two strategies for subtyping are not equivalent.

Note that the second version of the subtyping rule is only imprecise, but not unsound: it rejects some valid subtypes, but it does not accept any subtype tests that do not hold. The reason is that a relative effect can be expanded to conform to a concrete effect as explained in the previous Section 3.2.1, but not the opposite. For a concrete effect in the subtype, there must be a matching concrete effect in the supertype. Since the modified typing rule uses

the parameter symbol with a potentially larger effect, it is more restrictive than the original version.

If we invert the original example, we see that the modified typing rule does not incorrectly accept subtype tests:

$$\cdot \vdash (g : T \xrightarrow{e} T) \perp (T \xrightarrow{e} T) <: (f : T \rightarrow T) \xrightarrow{f} (T \xrightarrow{f} T)$$

The rule S-SUB correctly rejects this subtype test when comparing the result types:

$$\begin{aligned} f : T \rightarrow T \vdash [f/g](T \xrightarrow{e} T) <: (T \xrightarrow{f} T) \\ f : T \rightarrow T \vdash (e, \cdot) \leq (\perp, f) \quad \text{does not hold, } e \not\leq \perp \end{aligned}$$

The modified rule in which  $g$  is used to compare the two result types also rejects it:

$$\begin{aligned} g : T \xrightarrow{e} T \vdash (T \xrightarrow{e} T) <: [g/f](T \xrightarrow{f} T) \\ g : T \xrightarrow{e} T \vdash (T \xrightarrow{e} T) <: (T \xrightarrow{g} T) \\ g : T \xrightarrow{e} T \vdash (e, \cdot) \leq (\perp, g) \quad \text{does not hold, } e \not\leq \perp \end{aligned}$$

### Dependent Method Types in Scala

The Scala programming language uses invariant instead of contravariant subtyping for parameter types [Odersky, 2013]. The subtyping relation for two method types can only hold if the parameter types of the two methods are identical. This implies that a method cannot be overridden in a subclass with a method that accepts more general parameter types. The following example illustrates this behavior:

```
class Base {
  def m(s: String) = "Base"
}

class Sub extends Base {
  override def m(o: Object) = "Sub"
}
```

The compilation of the second class fails with the error message “method  $m$  overrides nothing”. If we remove the override modifier the example does compile, but then the subclass defines an overloaded alternative for method  $m$  instead of overriding the one in the superclass.

Because subtyping for parameter types in Scala is invariant, the question of choosing the correct parameter symbol discussed in Section 3.2.1 is irrelevant. The subtyping relation can only hold if the two symbols have the same type, therefore any of the two symbols can be used for comparing the result types.

### 3.2.2 Typing Rules

The typing rules for REL are presented in Figure 3.3. Terms are type checked using a typing statement of the form  $\Gamma; \bar{f} \vdash t : T \mid e$  where  $\Gamma$  maps variables to their types and  $\bar{f}$  keeps track of the function parameters in which the current expression is effect-polymorphic. The environment  $\bar{f}$  has a similar purpose as the parameter environment in the typing rules of LPE in Section 2.5.2, but consists of list of parameters instead of just one.

The typing rule for variables is standard. In the typing rule for function abstractions, the annotated relative effect  $\bar{x}$  is used as the polymorphism environment for typing the function body. We illustrate the typing rule using a simple higher-order function:

$$\text{let } h = (f : \text{Int} \rightarrow \text{Int}) \xrightarrow{f} f \ 1$$

The body of  $h$  is type checked as  $f : \text{Int} \rightarrow \text{Int}; f \vdash f \ 1$ , where the polymorphism environment  $f$  records that the enclosing function is effect-polymorphic in  $f$ . Additionally, the typing rule T-ABS ensures that the function type is well formed by using the predicate  $\Gamma \vdash T$  which ensures that all relative effects refer to parameters that are in scope.

The rule T-APP-PARAM type checks function invocations where the function is a parameter listed in the polymorphism environment. In this case the latent effect of the function is excluded from the effect of the expression because it is already included in the relative effect annotation of the enclosing function. Therefore, the function body of the above example has no effects:

$$f : \text{Int} \rightarrow \text{Int}; f \vdash f \ 1 : T \mid \text{eff}(\text{APP}, \perp, \perp, \perp)$$

The last typing rule T-APP is used for all remaining function invocations. It computes the latent effect of the invoked function using the auxiliary function  $\text{latent}_{\Gamma; \bar{f}}(T)$ . The latent effect of a function type  $(x : T_1) \xrightarrow{\bar{x}} T_2$  is the maximal effect that might occur when a function of that type is invoked. It consists of the concrete effect  $e$  and the latent effects of the functions  $f \in \bar{x}$  which are part of the relative effect of the function type.

However not all relative effects are included in the effect of the function application: the relative effects of the invoked function which also belong to the polymorphism environment of the current expression can be ignored. This is achieved by filtering out the relative effects of the environment and considering only functions  $f \in (\bar{x} \setminus \bar{f})$  when computing the joined relative effect  $e_p$ . The following example explains this aspect of the typing rule:

$$\begin{aligned} \text{let } h &= (f : \text{Int} \rightarrow \text{Int}) \xrightarrow{f} \\ &\quad \text{let } g = (x : \text{Int}) \xrightarrow{f} f \ x \\ &\quad g \ 1 \end{aligned}$$

$$\begin{array}{c}
 \boxed{\Gamma; \bar{f} \vdash t : T! e} \qquad \text{T-VAR} \frac{\Gamma(x) = T}{\Gamma; \bar{f} \vdash x : T! \perp} \\
 \\
 \text{T-ABS} \frac{(\Gamma, x : T_1); \bar{x} \vdash t : T_2! e \quad \Gamma \vdash (x : T_1) \xrightarrow{\bar{x}} T_2}{\Gamma; \bar{f} \vdash (x : T_1) \xrightarrow{\bar{x}} t : (x : T_1) \xrightarrow{\bar{x}} T_2! \perp} \\
 \\
 \text{T-APP-PARAM} \frac{f \in \bar{f} \quad \Gamma; \bar{f} \vdash f : (x : T_1) \xrightarrow{\bar{x}} T! \perp \quad \Gamma; \bar{f} \vdash t_2 : T_2! e_2 \quad \Gamma \vdash T_2 <: T_1}{\Gamma; \bar{f} \vdash f t_2 : [T_2/x]_{\Gamma; \bar{f}} T! \text{eff}(\text{APP}, \perp, e_2, \perp)} \\
 \\
 \text{T-APP} \frac{\Gamma; \bar{f} \vdash t_1 : (x : T_1) \xrightarrow{\bar{x}} T! e_1 \quad \Gamma; \bar{f} \vdash t_2 : T_2! e_2 \quad \Gamma \vdash T_2 <: T_1 \quad e_p = \sqcup_{f \in (\bar{x} \setminus \bar{f})} \text{latent}_{\Gamma; \bar{f}}((\Gamma, x : T_2)(f))}{\Gamma; \bar{f} \vdash t_1 t_2 : [T_2/x]_{\Gamma; \bar{f}} T! \text{eff}(\text{APP}, e_1, e_2, e \sqcup e_p)} \\
 \\
 \boxed{\Gamma \vdash T} \qquad \frac{\Gamma \vdash T_1 \quad \Gamma, x : T_1 \vdash T_2 \quad \forall f \in \bar{x}. f \in (\Gamma, x : T_1)}{\Gamma \vdash (x : T_1) \xrightarrow{\bar{x}} T_2} \\
 \\
 \boxed{\text{latent}_{\Gamma; \bar{f}}(T)} \qquad \frac{e_p = \sqcup_{f \in (\bar{x} \setminus \bar{f})} \text{latent}_{\Gamma; \bar{f}}((\Gamma, x : T_1)(f))}{\text{latent}_{\Gamma; \bar{f}}((x : T_1) \xrightarrow{\bar{x}} T_2) = e \sqcup e_p} \\
 \\
 \boxed{[T_x/x]_{\Gamma; \bar{f}} T} \qquad \frac{x = y}{[T_x/x]_{\Gamma; \bar{f}} \left( (y : T_1) \xrightarrow{\bar{y}} T_2 \right) = (y : [T_x/x]_{\Gamma; \bar{f}} T_1) \xrightarrow{\bar{y}} T_2} \\
 \\
 \frac{x \notin \bar{y} \wedge x \neq y \quad T_x = (z : T_a) \xrightarrow{\bar{z}} T_b \quad e_p = \begin{cases} \text{latent}_{\Gamma; \bar{f}}(T_a) & \text{if } z \in \bar{z} \\ \perp & \text{otherwise} \end{cases}}{[T_x/x]_{\Gamma; \bar{f}} \left( (y : T_1) \xrightarrow{\bar{y}} T_2 \right) = (y : [T_x/x]_{\Gamma; \bar{f}} T_1) \xrightarrow{(\bar{y} \setminus \{x\}), (\bar{z} \setminus \{z\})} [T_x/x]_{\Gamma; \bar{f}} T_2}
 \end{array}$$

Figure 3.3: Typing Rules for REL

### Chapter 3. Dependent Types for Relative Effects Declarations

---

Since function  $g$  has type  $\text{Int} \xrightarrow{f} \text{Int}$ , the invocation  $g\ 1$  has the effect of  $f$ . However, because the enclosing function  $h$  is *also* effect-polymorphic in  $f$ , the relative effect  $f$  can be ignored in the invocation of  $g$ .

Another crucial detail in T-APP is that the latent effect of the parameter function  $x$  is computed using the argument type  $T_2$  instead of the parameter type  $T_1$ . Therefore, the effect of a function invocation depends on the effect of the argument expression, which is the gist of effect-polymorphism. We illustrate this typing rule by applying the higher-order function  $h$  to the (pure) identity function:

$$\Gamma; \cdot \vdash h((x : \text{Int}) \rightarrow x) \quad \text{where } \Gamma = h : (f : \text{Int} \rightarrow \text{Int}) \xrightarrow{h} \text{Int}$$

The argument function has type  $\text{Int} \xrightarrow{\perp} \text{Int}$ , therefore the typing rule T-APP computes the latent effect of  $h$  as

$$\text{latent}_{\Gamma; \cdot}((\Gamma, f : \text{Int} \xrightarrow{\perp} \text{Int})(f)) = \text{latent}_{\Gamma; \cdot}(\text{Int} \xrightarrow{\perp} \text{Int}) = \perp$$

The effect-polymorphic definition of  $h$  and its invocation can be expressed in the type system of LPE presented in Chapter 2. The additional expressiveness of REL stems from the fact that functions cannot only be effect-polymorphic in their own argument, but also in the arguments of enclosing functions. For instance, the following higher-order function  $m$  which extends the behavior of an existing function has type  $(f : \text{Int} \rightarrow \text{Int}) \xrightarrow{\perp} (\text{Int} \xrightarrow{f} \text{Int})$ :

$$\text{let } m = (f : \text{Int} \rightarrow \text{Int}) \rightarrow (x : \text{Int}) \xrightarrow{f} f\ x$$

For each invocation of  $m$ , the effect of the resulting function depends on the effect of the argument function passed to  $m$ . This behavior is implemented in both typing rules for function applications: T-APP-PARAM and T-APP. In the result type of the invoked function  $T$ , references to the function's parameter are eliminated using the substitution function  $[T_2/x]_{\Gamma; \bar{f}} T$ .

Recall that dependent types only occur in the form of relative effect annotations on function types. The substitution  $[T_2/x]_{\Gamma; \bar{f}} T$  expands all relative effects  $x$  in type  $T$  to the latent effect of the function type  $T_2$ .

The typing rules implement effect-polymorphism by using the argument type  $T_2$  which might be a subtype of the function's parameter type  $T_1$ . An invocation of the higher-order function  $m$  with a pure identity function as argument is type checked as follows:

$$\Gamma; \cdot \vdash m((x : \text{Int}) \rightarrow x) \quad \text{where } \Gamma = m : (f : \text{Int} \rightarrow \text{Int}) \xrightarrow{\perp} (\text{Int} \xrightarrow{f} \text{Int})$$

The final type of the invocation is computed using the substitution

$$[\text{Int} \xrightarrow{\perp} \text{Int}/f]_{\Gamma}; (\text{Int} \xrightarrow{f} \text{Int}) = \dots = \text{Int} \xrightarrow{\perp} \text{Int}$$

which yields a function type with no side effects.

### 3.3 Relative Effect Declarations in Scala

This section explains how the type system with relative effects is integrated into the Scala programming language. The main difference between Scala and the REL language from the previous section is that Scala is an object-oriented language. Every parameter of a function carries a potentially large number of methods, while in REL only single functions can be passed as argument to other functions.

The effect of an effect-polymorphic function is specified in terms of its argument functions. The fact that objects collect multiple functions into a record does not change the need for effect-polymorphism. For instance, the effect of a logging function which calls `toString` on its argument object `o` depends on the effect of that `toString` method:

```
def log(msg: String, o: Any): Unit = {
  println(msg + o.toString) // IO effect, and the effect of o.toString
}
```

However the annotation scheme for relative effects needs to be adjusted to support the object-oriented language: instead of just referring to the parameters of the function, a relative effect annotation needs to specify in *which* of the parameter methods the function is effect-polymorphic. A relative effect expression in Scala is therefore a method selection where the receiver is a parameter of the polymorphic method, for example `o.toString` in the above example.

#### 3.3.1 Syntax for Relative Effect Annotations

The syntax for annotating effects in Scala is explained in detail in Chapter 5. For the purpose of discussing relative effect annotations we need to know that effect annotations are standard Scala annotations on the result types of method declarations. The effect annotation `@pure` ranges across all effect domains and corresponds to the multi-domain purity annotation  $\perp$  introduced in Section 2.4. In Scala, the `@pure` annotation is also used for annotating effect-polymorphism: the annotation accepts an arbitrary number of argument expression where each expression denotes a relative effect of the function. Since effect-polymorphic functions are pure by default as explained in Section 3.2, using the `@pure` annotation for denoting effect-polymorphism is a natural choice. The previous example is annotated as follows:

## Chapter 3. Dependent Types for Relative Effects Declarations

---

```
def log(msg: String, o: Any): Unit @pure(o.toString) @io
```

Each argument expression of the `@pure` annotation is required to be of the form `p.m( $\bar{a}$ )...( $\bar{a}$ )` where `p` is either a parameter of the annotated method, a parameter of an enclosing method or the self parameter `this`. The method `m` is a member of object `p`, and the term `%` is a pre-defined function of type `Nothing`.

Because the relative effect annotations are standard Scala annotations, their arguments are type checked like any other expression written in Scala. If the method `p.m` takes parameters, then the annotation `@pure(p.m)` leads to a type error with the message “missing arguments for method m”. The annotation only passes the Scala type checker if the method selection is extended with arguments. Since the function `%` has type `Nothing` it conforms to any other type an can therefore be used as argument expression. The effect system only takes the parameter `p` and the selected method `m` into account; the concrete argument expressions are not relevant. The following example illustrates the relative effect annotation:

```
def invoke(f: Int => Int): Int @pure(f.apply(%)) = {
  f(10)
}
```

If the method `p.m` has multiple overloaded alternatives, overloading resolution can be guided to choose the desired method by adding type ascriptions to the argument expressions `%`:

```
class A {
  def f(t: T): A
  def f(s: U): A

  def op(): A @pure(this.f(% : T)) = { ... }
}
```

Note that using an underscore instead of the pre-defined `%` function is currently not supported. The reason is that the Scala type checker transforms an expression of the form `p.m(_)` into an anonymous function  $(x: T_x) \Rightarrow p.m(x)$ . The current implementation of the effect system does not recognize anonymous function expressions as relative effect annotations. This is a technicality that can be changed.

### Relative Effects for apply Methods

Higher-order methods which are effect-polymorphic in their argument function are common in Scala. The classical example is the method `map` which transforms the elements of a collection using the function passed as argument. A function in Scala is represented as an object of type `FunctionN[T1, ..., TN, R]` where `N` is the number of parameters<sup>1</sup>. The function is invoked by

---

<sup>1</sup>The syntactic type  $(T_1, \dots, T_N) \Rightarrow R$  is equivalent



calling the member method `apply` which takes  $N$  parameters of types  $T_1, \dots, T_N$  and returns an object of type  $R$ .

Scala applies an implicit conversion to method invocation expressions if the selected function does not denote a method. In this case a selection of the member method `apply` is automatically inserted by the compiler. For example, the expression `f(1)` is transformed to `f.apply(1)` if the value  $f$  is of type `Int => Int`.

The relative effect annotation of a higher-order method takes the form `f.apply( $\bar{\%}$ )`, as shown in the method `invoke`. To simplify the relative effect annotation for this common case, the `apply` method is used by default if the relative effect does not explicitly select a method of the parameter. The method `invoke` can therefore be annotated as

```
def invoke(f: Int => Int): Int @pure(f) = f(10)
```

This special case is not specific to function types: for parameters of any type, the relative effect  $p$  is equivalent to `p.apply( $\bar{\%}$ )` with the necessary number of `%` arguments. If the object  $p$  defines multiple overloaded `apply` methods, the annotated method is effect-polymorphic in *all* of the `apply` members of object  $p$ . This can also be annotated explicitly by providing multiple relative effect expressions, e.g., “`@pure(p.apply(% : T), p.apply(% : U))`” for two overloaded methods with parameter types  $T$  and  $U$ .

#### Effect-Polymorphism is not Inferred

As described in Section 5.1.1, the effect of a method is inferred if its return type is inferred. In the examples in this chapter, we assume that only the effect domain for IO is enabled.

```
def f = 1 // inferred type of f: Int @noIo
def g = print("hi") // inferred type of g: Unit @io
```

Methods with an inferred effect are always effect-monomorphic, they do not have any relative effect annotations. The type system does not attempt to identify method calls which have a parameter of the current method as receiver:

```
def invoke(h: Int => Int) = h(1) // inferred return type of invoke: Int @io
```

The method `invoke` is inferred to be impure because the `apply` method in the type `Int => Int` can have arbitrary effects, i.e., the invocation of  $h$  is impure. In order to make the method effect-polymorphic, the programmer needs to provide an explicit return type and a relative effect annotation. The only exception to this rule is made for nested methods as we explain in Section 3.3.3.

There are two reasons why relative effect annotations are not inferred. First, inferring a relative effect for every method invocation with a parameter as receiver would lead to a large effect

## Chapter 3. Dependent Types for Relative Effects Declarations

---

and also be misleading: a relative effect should document effect-polymorphism and not other effects. Second, using a relative effect annotation to express a non-polymorphic effect can lead to imprecise effect inference. The reason is explained in Section 3.4: relative effect annotations do not encode information about the argument which is passed to the parameter method invocation.

### 3.3.2 Refined Types for Effect-Polymorphism

The gist of relative effect annotations is that the effect of a method might be more precise at call site than at declaration site. In the following definition, the parameter method `a.op` can have arbitrary effects:

```
abstract class A {
  def op(): Unit
}

def m(a: A): Unit @pure(a.op) = {
  a.op()
}
```

For each invocation of `m`, the effect is computed using the method `op` of the actual argument passed to `m`. For instance, the invocation of `m` in the example below has effect `@noIo`:

```
class B extends A {
  def op(): Unit @noIo = ()
}

def t(b: B): Unit @noIo = {
  m(b) // effect @noIo
}
```

Exploiting effect-polymorphism requires the argument type to override (or implement) a method from the original parameter type with a smaller, more specific effect. The method `op` has the unknown effect in class `A`, but only `effect @noIo` in the subclass `B`. It is not necessary to have a named subclass in order to make use of effect-polymorphism:

```
def t(): Unit @noIo = {
  val p = new A { def op(): Unit @noIo = () }
  m(p) // effect @noIo
}
```

In this example the Scala compiler assigns the refinement type “`A { def op(): Unit @io }`” to the value `p` because the return type of `op` has a more precise effect than in the parent type `A`. Section 5.4.3 explains how the Scala type checker is extended to take effects into account. When computing the effect of the invocation `m(p)`, the effect of method `op` is looked up in the

refinement type of  $p$ , which yields `@noIo`.

The return type and the effect annotation of method `op` in the above example can be inferred, as we show in Section 5.1.1. The example can be shortened as follows:

```
def t(): Unit @noIo = {
  m(new A { def op() = () })
}
```

Note that the example is a typical instance of the “strategy” pattern described by Gamma et al. [1995]. In languages that support higher-order functions, this pattern is usually replaced by a function parameter in the method definition and a function literal at the call site:

```
def m(op: () => Unit): Unit @pure(op) = {
  op.apply()
}

def t(): Unit @noIo = {
  m(() => ())
}
```

Effect checking in this last example uses the exact same mechanisms that were introduced with the other examples in this section. The Scala type checker assigns a refinement type to the function literal which specifies the precise effect of the function. The full type of the function literal can be annotated if desired:

```
val f: Function0[Unit] { def apply(): Unit @noIo } = () => ()
```

The refinement specializes the effect of the `apply` method, which is allowed to have arbitrary effects in the trait `Function0`. When computing the effect of the invocation `m(f)`, the relative effect `@pure(op)`, which is equivalent to `@pure(op.apply())`, is expanded to `@noIo`.

Function types with refinements are relatively verbose and hard to read. In practice we encountered only few situations where function types with explicit effects have to be written by programmers. For function literals the refined types are inferred and methods with function parameters typically accept functions with arbitrary effects, i.e., do not require a refinement. However, if the verbose syntax turns out to be problematic we plan to introduce syntactic sugar for function types with specific effects.

#### 3.3.3 Relative Effects for Nested Definitions

A nested method definition can declare relative effects which refer to parameters of its enclosing method. This feature is required to correctly express the effect of methods which are nested within an effect-polymorphic method:

### Chapter 3. Dependent Types for Relative Effects Declarations

---

```
def invoke(f: Int => Int): Int @pure(f) = {
  def impl(): Int @pure(f) = f(10)
  impl()
}
```

The invocation of `impl` type checks without any concrete effect because its relative effect  $f$  is also a relative effect of the enclosing method `invoke`. If the relative effect annotation is omitted in the nested method, the example does not type check:

```
def invoke(f: Int => Int): Int @pure(f) = {
  def impl(): Int @pure = f(10) // fails
  impl()
}
```

This definition leads to an effect mismatch error because the invocation of  $f$  might have arbitrary effects, which the signature of `impl` does not allow. The body of method `impl` is type checked in a context without any relative effects.

Marking the nested method `impure` does not work either:

```
def invoke(f: Int => Int): Int @pure(f) = {
  def impl(): Int = f(10) // ok
  impl() // fails
}
```

In this case the *invocation* of the nested method `impl` fails because by its signature it might have arbitrary effects, while the effect annotation of `invoke` only allows the effect of function  $f$ .

If the return type of the nested method is omitted entirely, the return type and the latent effect will be inferred. However we defined in Section 3.3.1 that methods with an inferred effect are never effect-polymorphic. This means the example would not compile if the return type of the nested method is omitted: the method `impl` is inferred to have the topmost effect and the invocation of `impl` would fail just as in the last example.

To address this problem the type system makes an exception for nested methods. If the effect of a nested method is inferred, then it inherits the relative effect annotations from the next enclosing method. The example therefore type checks without a type and effect annotation on the nested method `impl`:

```
def invoke(f: Int => Int): Int @pure(f) = {
  def impl() = f(10) // return type of impl: Int @pure(f)
  impl() // ok
}
```

**Relative Effects in Nested Classes and Function Literals**

Nested methods are not necessarily defined as direct children of an outer method, but they can also be members of classes or objects defined within a method:

```

1  abstract class A {
2    def op(): Unit
3  }
4
5  def m1(a1: A): Unit @pure(a1.op) = {
6    a1.op()
7  }
8
9  def m2(a2: A): Unit @pure(a2.op) = {
10   class NestedA extends A {
11     def op(): Unit @pure(a2.op) = {
12       a2.op()
13     }
14   }
15   m1(new NestedA)
16 }

```

To compute the effect of the invocation of method `m1` in line 15, its relative effect `@pure(a1.op)` is expanded using method `op` defined in the argument type, i.e., `NestedA`. This method has itself a relative effect `@pure(a2.op)` which happens to be in the current polymorphism environment: the outer method has the same relative effect. Therefore the invocation of `m1` is treated as pure.

The example involving a nested class might seem far fetched, but this situation occurs frequently when using higher-order functions. After translating the above example to use higher-order functions it looks less obscure:

```

1  def m1(op1: () => Unit): Unit @pure(op1) = {
2    op1()
3  }
4
5  def m2(op2: () => Unit): Unit @pure(op2) = {
6    val f = () => op2()
7    m1(f)
8  }

```

Similar as in the last example of Section 3.3.2, the function literal `f` in line 6 is assigned a refinement type which specializes the effect of the function's `apply` method:

```
val f: (() => Unit) { def apply(): Unit @pure(op2) } = () => op2()
```

The function inherits the relative effect annotations from the outer method `m2`, as explained in Section 3.3.3. This programming pattern is ubiquitous in the design of the Scala collections

```
1 trait TraversableLike[+A, +Repr] {
2   def foreach(f: A => Unit): Unit @pure(f)
3
4   def newBuilder: Builder[A, Repr] @pure
5
6   def filter(p: A => Boolean): Repr @pure(p) = {
7     val b = newBuilder
8     for (x <- this)
9       if (p(x)) b += x
10    b.result
11  }
12 }
```

Figure 3.4: Trait TraversableLike

library which we describe in the following section.

### Relative Effects in the Scala Collections Library

The Scala collections library in its current form was introduced in Scala 2.8 [Odersky and Moors, 2009]. Each collection class inherits from trait `TraversableLike` which has an abstract method named `foreach`. Using `foreach`, the trait defines implementations for all of the common collection operations such as `isEmpty`, `filter` or `map`. Figure 3.4 presents the relevant parts of trait `TraversableLike`.

Having shared implementations across multiple collection types is great for maintainability, but it is challenging with respect to typing: when invoking `filter` on a `Set[Int]`, the programmer expects to obtain a result of type `Set[Int]`, similarly for other collection types. To abstract over the representation type, the trait `TraversableLike` has a type parameter `Repr` which represents the collection type in addition to the type parameter `A` which denotes the collection's element type.

The collection class `Set[A]` extends `TraversableLike[A, Set[A]]` and its method `filter` has return type `Set[A]`. The implementation of `filter` creates a new builder object, appends all elements that satisfy the predicate `p` to the builder, and retrieves the resulting collection. The method `newBuilder` returns a builder object which creates a collection of type `Repr`. This way, the generic method `filter` returns a set in class `Set[A]`, a list in class `List[A]` and so on for other collection types.

The method `foreach` is annotated to be pure up to the effect of its argument, i.e., `@pure(f)`. Override checking therefore enforces all implementations of `foreach` to be pure as well, as explained in Section 5.4.3. To discuss effect checking of method `filter` we consider an effect system that tracks IO effects. An effect system that can express the state modifications on the

builder object is introduced in Chapter 4.

The for comprehension in lines 8 and 9 is transformed by the Scala compiler to an equivalent invocation of `foreach`:

```
this.foreach(x => if (p(x)) b += x)
```

Since the `append` method `+=` of the builder is pure (it has no IO effect), the function literal has only the effect of the predicate invocation `p(x)`. The function is defined within method `filter` which is effect-polymorphic in `p`, therefore it inherits the relative effect annotation `@pure(p)` and obtains the refined type `(A => Unit) { def apply(x: A): Unit @pure(p) }`.

Method `foreach` has the effect of the `apply` method of its argument function, which is the relative effect `@pure(p)` for the invocation within `filter`. This relative effect can be ignored since `filter` itself has the same relative effect, and the entire for comprehension is pure. Because the invocations `newBuilder` and `b.result` also have no side effects, the implementation of `filter` can be annotated `pure`.

## 3.4 Expressiveness of Relative Effects

The examples in the previous section show that the type system presented in this chapter can express effect-polymorphism of common code patterns such as those in the Scala collections library. Annotating existing code is straightforward and does not require any refactorings such as the introduction of effect parameters. The syntactic overhead thus remains minimal.

There are however examples of effect-polymorphic code which cannot be correctly annotated with relative effect annotations. This section discusses the expressiveness of the type system and gives an intuition on the practical impact of its limitations.

The experience gained from applying relative effects to existing Scala code suggests that the limitations presented in this section are not critical in practice. However the effect system has not yet been made available to a broader audience of Scala programmers, which will provide important data for a comprehensive assessment of the practicality of the type system.

### Abstracting Over Functions With Relative Effects

If the parameter function of an effect-polymorphic method is itself effect-polymorphic, then a relative effect annotation usually leads to an over-approximation of the effect:

```
class A {  
  def run(f: A => Unit): Unit @pure(f) = f(this)  
}
```

### Chapter 3. Dependent Types for Relative Effects Declarations

---

```
def doRun(a: A, g: A => Unit): Unit @pure(a.run(%)) = {  
  a.run(g)  
}
```

The effect of `doRun` is annotated as the effect of the method `run` of its parameter `a`. However, the relative effect annotation does not encode *which* function is passed as argument to `run`. This leads to an over-approximation in the following invocation:

```
doRun(new A, a => ())
```

To compute the effect of the invocation, the relative effect `a.run(%)` of `doRun` is expanded to the relative effect `f` of method `run`. At this point the type system does not know which function the parameter `f` binds to, so it assumes the unknown effect.

This limitation is only relevant when a relative effect abstracts over a method that is effect-polymorphic and also gets overridden with a more specific effect in subclasses. In the example, the relative effect annotation `@pure(a.run(%))` only makes sense when the `run` method of class `A` is expected to be overridden with a more specific effect. In our practical experience we did not encounter such code: effect-polymorphic methods are typically overridden with the same effect, e.g., the method `foreach` in the `collections` library.

If a method is *not* expected to be overridden with a more specific effect, then there is no advantage to abstract over its the effect. In the example, purity can be verified correctly if method `doRun` is not effect-polymorphic in `a.run`:

```
def doRun1(a: A, g: A => Unit): Unit @pure(g) = {  
  a.run(g)  
}  
doRun1(new A, a => ())
```

The type system infers the invocation of `doRun1` to be pure.

The reason for the over-approximation in effect inference of the initial example is that the relative effect annotation `@pure(a.run(%))` does not encode the fact that the function passed as argument to `run` is pure. If this additional precision turns out to be essential in practice, the expressiveness of the type system could be enhanced by including the argument type in the relative effect annotation:

```
type PureFun = (A => Unit) { def apply(x: A): Unit @pure }  
def doRun(a: A, g: A => Unit): Unit @pure(a.run(%: PureFun)) = ...
```



## Polymorphism With Functions in Data Structures

Relative effect annotations are syntactically restricted to methods of parameters, for instance they cannot refer to methods of objects that that are *obtained* through a parameter. This leads to situations where effect-polymorphism of a method cannot be expressed, for instance when the parameter function is curried (assume that the single arrow  $\rightarrow$  denotes pure functions, i.e., a function type  $T \rightarrow S$  is equivalent to  $(T \Rightarrow S)$  { `def apply(x: T): S @pure }`):

```
def appZeros(f: Int -> Int => Int): Int @pure(f) = {
  f(0)(0)
}
```

Since function types in Scala associate to the right, function `f` has type “`Int -> (Int => Int)`”. The relative effect annotation `f`, which is equivalent to `f.apply(%)`, relates to the pure outer function which returns another function, and *not* to the inner function returning an integer. For that reason the definition of `appZeros` leads to an effect mismatch error: the relative effect annotation allows the effect of `f.apply`, which is pure anyway, but the unknown effect of `f(0).apply` is not annotated in the signature.

For curried functions the effects typically only happen at the innermost level, for example:

```
val div = (x: Int) => (y: Int) => {
  if (y == 0) error("div by 0") else x / y
}
```

Effect-polymorphism of curried functions cannot be expressed with relative effect annotations. This is a minor issue in Scala because programmers usually write functions with multiple parameters instead of curried functions. When using a function with two arguments, the relative effect annotation works as expected:

```
def appZeros(f: (Int, Int) => Int): Int @pure(f) = {
  f(0, 0)
}
```

Besides curried functions, there are other examples of effect-polymorphic code that cannot be expressed with relative effect annotations for the same underlying reason. The problem is that methods can only be polymorphic in members of their parameters, but not in members of objects *obtained through* their parameters. This case occurs for instance when functions are stored in a data structure that is passed as argument:

```
def mApply(x: Int, mf: Option[Int => Int]): Int =
  if (mf.isEmpty) x
  else mf.get(x)
```

## Chapter 3. Dependent Types for Relative Effects Declarations

---

The effect of `mApply` is the effect of the function stored in the optional value passed as argument, but this cannot be expressed using relative effect annotations. In a type system with parametric polymorphism for effects, the effect of method `mApply` would be expressed by an effect type parameter that is used as the effect of the optional parameter function.

Another example whose effect cannot be expressed with relative effect annotations for similar reasons is the observer pattern: the effect of the subject notifying the observer depends on the effect of the observer.

```
class Subject {
  var obs: Observer = _
  def register(o: Observer) { obs = o }
  def notifyObserver() { obs.update(this) }
}
trait Observer {
  def update(s: Subject)
}
```

In method `notifyObserver`, the invoked method `this.o.update` is not a direct member of the `this` parameter, so a relative effect annotation cannot refer to `update`. In an effect system with explicit parameters the `Subject` class could be parametrized by the effect of its observer.

Instead of introducing explicit effect type parameters, relative effect annotations could be generalized to arbitrary paths. This would allow expressing the effect-polymorphism of the examples in this section. For example, the relative effect of method `notifyObserver` would be `@pure(this.o.update)`. The difficulty with this idea is that it is not clear how the identity of objects can be tracked systematically. For example, the class `Option` in the Scala standard library is abstract and does not have any field members. Given an object `o` of type `Option`, the system should be able to show that the expressions “`o.get`” and “`o match { case Some(v) => v }`” denote the same value. Such a system would be considerably more complex than the relative effect annotations introduced in this chapter.

### Effect Masking

In the type system presented in this chapter, effect masking can only be expressed at the level of type-and-effect inference rules, but not in the effect annotations of a method.

With by-name parameters and multiple argument lists, Scala provides support for user-defined control structures that are as elegant as built-in language features for the programmer. For instance, users can define a custom operator for exception handling:

```
def myTry[T](f: => T)(c: PartialFunction[Throwable, T]) =
  try { f }
  catch { case e: Throwable if c.isDefinedAt(e) => c(e) }
```

```
myTry { 1 / 0 }
      { case _: java.lang.ArithmeticException => 0 }
```

The effect of an invocation of `myTry` is the effect of the parameter function `f`, but without the exceptions handled by the partial function `c`.

The annotation system for polymorphic effects could be extended to support effect masking, for instance by adding a `mask` parameter to relative effect annotations. The effect annotation `@pure(f, mask = throws[E])` would describe a method that catches the exception `E` of its parameter function `f`.

For the effect primitive `myTry`, that annotation system alone is not sufficient: the effect mask cannot be defined statically but it depends on the exceptions that are handled by the partial function `c`. Expressing effect masking in this case requires a further extension to the type system which tracks the definition domain of partial functions.

There are effect domains which would profit from the ability to express effect masking even without advanced extensions to the type system, for instance the effect system for blocking operations presented in Section 2.6.2. Another example is the `breakable` operator defined in the Scala library that is discussed in Section 5.2.2. Effect annotations that support static effect masks are supported in other projects like the work on Anchored Exceptions [van Dooren and Steegmans, 2005] or the Koka language [Leijen, 2012].

### 3.5 Related Work

Effect-polymorphism using effect type parametrization was first introduced by Lucassen and Gifford [1988] and is described in Section 2.2. The relative effect annotations introduced in this chapter eliminate most of the syntactic overhead involved with explicit effect parameters while keeping the ability to express the behavior of common programming patterns.

The idea of expressing the exceptions of a method in terms of the exceptions of some other method has been explored in the work on *anchored exception declarations* by van Dooren and Steegmans [2005]. To solve the verbosity and limited expressiveness of Java's checked exceptions, they introduce a second kind of exception declaration, called an *anchored* exception declaration. Anchored exception declarations have the form "like `<InvocationExpression>`" and co-exist with ordinary, concrete throws clauses. The expression in an anchored exception declaration is an arbitrary function invocation in which type names can also be used as terms, for instance to resolve an overloaded method. For example, "like `p.a().b()`" allows the exceptions of method `b()` when accessed through `p.a()` and the annotation "like `A.m()`" allows the exceptions of method `m()` from any value of type `A`. Their system supports effect-polymorphism by substituting argument expressions for parameter names at call sites, which potentially refines the method selection of an anchored exception declaration to a more

specific method with fewer exceptions. Anchored exception declarations that do not use parameters, e.g., “like `A.m()`”, are mere aliases of exception lists. Finally, anchored exception declarations can express effect masking. The relative effect annotations introduced in this chapter are simpler than anchored exception declarations and allow specifying effect-polymorphism across multiple effect domains.

Kneuss et al. [2013] present a whole-program effect inference algorithm for memory effects which works well for programs with callbacks such as programs using higher-order functions. Effects are represented as control flow graphs that may contain invocations of methods on parameters, which is equivalent to relative effect annotations. They employ heuristics to decide whether the effect of a method invocation should be considered directly or represented as a relative effect, which they call a *delayed* invocation. Their system is also discussed in Section 4.5.3 as related work of our purity type system that we describe in the next chapter.

### 3.6 Conclusion

In this chapter we introduced *relative effect annotations*, a lightweight and intuitive syntax based on dependent types for denoting effect-polymorphic functions. Relative effects precisely express effect-polymorphism of functions that take multiple functions as arguments. Such higher-order functions are common in object-oriented programming languages where each object holds a potentially large number of member methods.

The type system with relative effects is explained using a lambda calculus with effects and dependent types. We evaluate the type system with an implementation for the Scala language and show that relative effect annotations can express common patterns involving higher-order code such as those found in the Scala collections library.

In the future we plan to extend relative effect annotations with the ability to express effect masking. This allows expressing the behavior of certain methods directly in their signature instead of requiring a dedicated typing rule, e.g., the `breakable` operator defined in the Scala library.

Section 3.4 shows examples of effect-polymorphism that can be expressed with explicit effect type parameters, but not with relative effect annotations. In case these issues turn out to be relevant in practice, we plan to work on a hybrid strategy for expressing effect-polymorphism that unifies the benefits of relative effect annotations and traditional parametric polymorphism.

## Chapter 4

# A Type-and-Effect System for Purity

This chapter presents a type-and-effect system which tracks purity of functions with respect to state modification effects. We implemented the system as an effect domain in the generic framework for polymorphic effect checking introduced in Chapter 3. The effect system can express purity of common programming patterns that make use of local state such as the use of an iterator, nevertheless effect annotations are lightweight and intuitive.

### 4.1 Introduction

One of the differences between impure and purely functional programming languages is support for mutable state. Efficient implementations of algorithms and data structures often use mutable state internally even though they appear pure or immutable to their clients. For example, the `Vector` class in the Scala standard library, an immutable random access data structure, is implemented using a trie of arrays which are copied and modified on transformation operations.

The desire to control the scope of memory effects is the driving force behind most existing work on type-and-effect systems, including the original effect system by Gifford and Lucassen [1986]. Memory effect systems have been used to schedule non-interfering expressions to execute in parallel [Lucassen and Gifford, 1988], [Bocchino et al., 2009] or to provide a stack-based implementation of a programming language with mutable state [Tofte and Talpin, 1994]. Knowledge about side effects and purity in particular can be used not only for optimizations and improved scheduling, but also for making programs safer and easier to understand. The fundamental difficulty in tracking memory effects stems from aliasing. Existing approaches for control aliasing and memory effects are discussed in Section 4.5.

The effect system in this chapter is based on ideas introduced by Pearce [2011] in `JPure`, a purity system for Java. It builds on the observation that if we modify state that is allocated

*within* a function, then clients of that function cannot observe these modifications. Similar to JPure, effects are computed using a modular, intraprocedural analysis based on effect annotations. We use the same concept of *locality* to denote private state which is part of an object's representation.

Compared to JPure, our effect system is flow-insensitive, which makes it suitable for higher-order languages such as Scala or C#. Flow-insensitivity also enables the effect system to be integrated with the generic effect system introduced in Chapters 2 and 3. We introduce a generalized notion of *freshness* that allows annotating the precise locality of a function's return value. This enables our system to correctly express the behavior of getter methods, which are ubiquitous in Scala. Finally, our system allows effect annotations of nested methods to refer to parameters and variables from the enclosing scope, which is vital for expressing effects in higher-order code.

## 4.2 Overview

This section gives an informal introduction to our purity effect system. We use the annotation syntax from the Scala implementation presented in more detail in Section 4.4.

### 4.2.1 Purity and Modification Effects

The type-and-effect system for checking purity is based on the observation that a method which does not modify state that existed before its invocation appears pure to its callers. In other words, a pure method is allowed to modify state which is allocated within the method, but no other program state. It can for example mutate temporary objects like iterators or initialize fields of freshly created object structures.

This definition of purity is common. It is also used by Sălcianu and Rinard [2005] in their purity system based on pointer analysis, in ownership-based systems such as the Universe Types by Dietl et al. [2007] or in the Java Modeling Language (JML, Leavens et al. [2006]).

We first consider a simple effect system that only has two effect annotations: pure and impure. A method is pure if it does not modify any existing state and impure otherwise. In this system, the method next of an iterator has to be marked impure because it modifies the state of the iterator. We consider a method that searches a specific element in a list using an iterator:

```
def contains(l: List[Int], e: Int): Boolean = {
  val it = l.iterator
  while (it.hasNext)
    if (it.next() == e) return true
  return false
}
```

The method `contains` does not have any observable side effect: the invocation `l.iterator()` returns a new iterator, `hasNext()` does not modify any state, and `next()` only modifies the state of the freshly allocated object `it`. However, the type signature of the impure method `next` does not specify that only the iterator is modified, thus the simple effect system requires the method `contains` to be annotated `impure`.

To overcome this limitations our purity type system supports more expressive effect annotations that specify which parameters a method is allowed to modify. The trait `Iterator` is annotated as follows:

```
trait Iterator[+T] {
  def hasNext: Boolean @mod()
  def next(): T @mod(this)
}
```

Method `hasNext` is annotated `@mod()`, which denotes purity; it does not allow any existing state to be modified. The effect annotation `@mod(this)` on `next` allows modifications to the state of object `this`.

Using this effect annotation, the effect system has enough information to know that the invocation of `next` in the body of `contains` only modifies the iterator object and not any other program state. However, in order to conclude that the implementation of `contains` is pure, the system also needs to know that the iterator object `it` is freshly allocated and non-aliased. We discuss in Section 4.2.3 how freshness is annotated and tracked, but first we explain the effect annotation `@mod(this)` more precisely.

### 4.2.2 Ownership and Locality

The effect annotation `@mod(this)` of method `next` in the iterator class allows modifications to the state of the iterator. For instance, a list iterator can be implemented using a single field:

```
class ListIterator[+T](list: List[T]) extends Iterator[T] {
  private[this] var l = list
  def hasNext = l.nonEmpty
  def next(): T @mod(this) = {
    val r = l.head
    l = l.tail
    r
  }
}
```

For each instance of class `ListIterator`, the state modified by method `next` is private to that instance and not shared with any other object. In a context where an iterator object is known to be freshly allocated, such as the body of method `contains` from Section 4.2.1, the invocation of `next` cannot modify any existing state.

## Chapter 4. A Type-and-Effect System for Purity

---

An effect annotation `@mod(o)` can be interpreted not only as a permission to modify the state of object `o`, but also as a conditional purity annotation. If the object `o` is known to be fresh, then an expression with effect `@mod(o)` is pure.

The system outlined so far fails to express the purity of another common programming pattern which uses only locally scoped effects: if an object holds internal state that is never shared with other objects or instances, modifications of that state can be encapsulated as modifications of the owner object. One example is the `Builder` type used in the Scala collections library [Odersky and Moors, 2009] which supports appending elements and retrieving the resulting collection.

```
trait Builder[-Elem, +To] {
  def +=(e: Elem): Unit @mod(this)
  def result: To @mod()
}
```

The class `ArrayBuffer` implements the builder interface using an internal array to store the appended elements:

```
class ArrayBuffer[A:ClassTag] extends Builder[A, ArrayBuffer[A]] {
  @local private[this] var array: Array[A] = new Array(initSize)
  private[this] var size = 0

  def +=(elem: A): Unit @mod(this) = {
    ensureSize(size + 1)
    array(size) = elem
    size += 1
  }
  def result: ArrayBuffer[A] @mod() = this
}
```

Like in the iterator example, if a fresh array buffer is used within a method to add elements and eventually obtain a collection, the effects on that builder and its array are not observable from the outside.

In order to support objects with internal data structures the system is extended with a simple notion of ownership. By marking a field of an object as `@local`, the programmer defines the internal state of the object accessible through that field to be owned by the outer object. A method annotated with effect `@mod(o)` is not only allowed to modify the fields of `o`, but also the fields of objects owned by `o`.

The *locality* of an object is defined as the transitive closure of all objects reachable through fields annotated `@local`. The type system ensures that an object type checks as being fresh only if all objects in its locality are also fresh, therefore modifications of the locality of a fresh object cannot change any existing program state.



### 4.2.3 Freshness and Result Localities

The effect annotation `@mod(o)` on a method ensures that *if* the value passed as argument for `o` in a specific invocation is known to be fresh, *then* that invocation only modifies fresh state.

In the example given in Section 4.2.1, purity of the method `contains` depends on the fact that the iterator `it` is known to be fresh, so modifying its fields does not modify any state that existed before. But how does the type system know that an object is fresh? The answer involves a third kind of annotation, the locality annotation `@loc`, which specifies the locality of the object that a method returns.

If a method always returns a freshly allocated object whose locality cannot be accessed through any previously existing state, then that method is called *fresh* and is annotated with the empty locality `@loc()`. The most prominent examples of such methods are factory methods, but there are other important fresh methods such as `List.iterator()` which always creates a new iterator<sup>1</sup>.

There are also methods which only return a fresh object if some of its parameters are fresh. The trivial example is the identity method:

```
def id[T](x: T): T @loc(x) = x
```

If a fresh object is passed to `id` then the resulting object is also fresh. Conditional freshness annotations are required to correctly express freshness of getters for local fields, like the method `getCounter` in line 8 of the following example:

```
1 class Counter {
2   private var x = 0
3   def inc(): Int @mod(this) = { x += 1; x }
4   def get: Int @mod() = x
5 }
6 class HasCounter {
7   @local var counter = new Counter
8   def getCounter: Counter @loc(this) = counter
9 }
```

As explained in the previous section, the effect system ensures that an object can only type check as fresh if all objects in its locality are also fresh. Therefore the method `getCounter` returns a fresh object if the receiver instance of type `HasCounter` is known to be fresh, as illustrated by the following example:

```
def test: Int @mod() = {
  val hc = new HasCounter // hc is fresh
  val c = hc.getCounter // c is also fresh
  c.inc() // modifies only fresh state
```

<sup>1</sup>In the case of an empty list, the same empty iterator can be reused. This case is discussed in Section 4.4.4.

```
}
```

At each call site, effects on parameters are translated by the effect system according to the localities of the corresponding arguments. In the following example, the effect `@mod(this)` of method `inc` in class `Counter` is translated to `@mod(hc)`:

```
def incHc(hc: HasCounter): Int @mod(hc) = {
  val c = hc.getCounter // c has locality @loc(hc)
  c.inc()                // modifies the locality of c
}
```

Getters are common in many programming languages, but even more so in Scala where every field access is performed through an accessor method [Odersky, 2013]. The ability to specify the locality of a method is therefore indispensable in order to apply the purity type system to Scala. If the locality of an object is unknown, we use the annotation `@loc(any)`. Examples of objects with an unknown locality are global objects and objects obtained by reading a non-local field. If an object of unknown locality is modified, then the expression has effect `@mod(any)` which denotes impurity.

### 4.2.4 Effects of Field Updates

The effect of updating a field of an object depends on whether the field is annotated `@local` or not. For non-local fields, the effect of an assignment is expressed as `@mod(o)` where `o` is the object that contains the field. The increment method of class `Counter` in Section 4.2.3 is an example of a non-local field update, it has effect `@mod(this)` since it changes the field `this.x`.

The difference with `@local` fields is that after the assignment, the stored object is part of the locality of the field's owner. Remember that an object can only be considered fresh if all objects in its locality are known to be fresh. When storing an object in the locality of a fresh object, freshness is only maintained if also the stored object is fresh.

In a flow-sensitive type system the localities of the objects involved in an assignment could simply be updated to reflect the fact that they have been merged. Since our effect system is not flow-sensitive we do not have this possibility, instead we keep track of assignments to `@local` fields using modification effects.

The idea is based on the observation that the `@mod` annotation expresses *conditional purity*: an expression with effect `@mod(x, y)` is pure if the localities `x` and `y` are known to be fresh. In order to find the correct effect annotation for an assignment expression we can therefore ask which objects need to be fresh for the assignment to be pure, i.e., to not modify any existing state. We consider the following example:

```
val hc = new HasCounter
hc.counter = someCounter
```

```
hc.counter.inc()
```

Since the object `hc` is known to be fresh, the assignment to the `@local` field `hc.counter` does not modify any existing state. The effect of incrementing the counter in the last line depends on the locality of the object `someCounter`, expressed as `@mod(someCounter)`. But the flow-insensitive type system is not capable of computing this effect: the object `hc` is fresh, and since `hc.counter` is part of its locality, the last line is assumed to be pure.

We address this issue by including both the owner of the changed field *and* the newly stored object in the effect of an assignment to a local field. Consequently, the assignment expression in the second line has effect `@mod(hc, someCounter)`. Note that this effect is an over-approximation: if the third line is omitted and the counter is not incremented, the example does *not* have any observable side effect, but the effect system still infers `@mod(someCounter)`. Section 4.4.2 investigates this issue in more detail and explains the reasons for using a flow-insensitive type system in the first place.

In Scala, assignments to local fields are performed through setter methods. Considering the above discussion, the effect annotation of a setter includes not only the modified object `hc` but also the stored object `c`:

```
def setC(hc: HasCounter, c: Counter): Unit @mod(hc, c) = {
  hc.counter = c
}
```

The effect annotation can again be understood as a conditional purity annotation: an invocation of `setC` is pure if the localities of both parameters `hc` and `c` are fresh.

### 4.2.5 Freshness Depends on Purity

In this section we investigate the relation between freshness and modification effects introduced in Section 4.2.4. The main conclusion is that the result of an expression can only be considered fresh if the expression does not have any side effects.

We illustrate this observation with a factory method that accepts an initial value for a local field of the constructed object. The method returns a fresh object, but the argument object is stored in the result:

```
def mkHC(c: Counter): HasCounter @mod(c) @loc() = {
  val hc = new HasCounter()
  hc.counter = c
  hc
}
```

As discussed in Section 4.2.4, the assignment effect includes both variables `@mod(hc, c)`. The

locality of the method body is  $@loc(hc)$ . Since the local variable  $hc$  is out of scope for the signature of the method, references to it are replaced by its initial locality, i.e., the empty locality  $@loc()$ . This leads to the effect and locality annotations in the method signature.

The freshness annotation  $@loc()$  might seem surprising: the returned object can only be typed as fresh if the parameter  $c$  is also fresh, otherwise it has non-fresh state in its locality. However, in combination with the effect  $@mod(c)$  the freshness annotation is safe. The reason is that the effect system can only consider an object as fresh in a pure context, i.e., in the absence of side effects. In order for an invocation of  $mkHC$  to be pure, the argument for parameter  $c$  is required to be fresh, and in this case the resulting object consists of only fresh state.

Objects can only be considered fresh in pure contexts. The reason is that any expression which *does* have a side effect  $@mod(o)$  can create aliases between the modified object  $o$  and previously fresh state: fresh state can be captured in the locality of  $o$ , or the object  $o$  can be captured in the locality of some fresh state.

Note that annotating the method  $mkHC$  with the result locality  $@loc(c)$  is equivalent and does not introduce any imprecision. We can say that the  $@loc$  annotation of a method implicitly contains all localities from the method's  $@mod$  effect.

### 4.3 Formalization

This section formally presents the type-and-effect system for purity outlined in Section 4.2. It uses the language PUR, a lambda-calculus with mutable records, presented in Figure 4.1. The formal language is in A-normal form (ANF, Flanagan et al. [1993]), hence all intermediate terms are named. In Section 4.3.3, we show an example that explains why the typing rules require terms to be in ANF.

The language does not feature arbitrary mutable references (as described by Pierce [2002] in Chapter 13); instead assignments are limited to the fields of records. These records represent a simple model of objects with mutable fields in object-oriented languages like Scala, which is sufficient to highlight the fundamental properties of the type system. The main simplification in PUR with respect to Scala is that the formal language does not support mutable local variables. Section 4.4.1 explains how assignments to local variables are handled in our implementation of the purity type system for Scala.

To define the ownership constraints introduced in the previous section, each field in a record literal can be optionally annotated as “local”. Similarly, record types register which fields of an object are local.

Function types consist of a parameter name and type, a latent effect  $e$  describing the localities the function might modify, a locality annotation  $loc$  that designates the locality of the returned value and a return type.

$t$	$::= \text{let } x = p \text{ in } t$	let-bound expression
	$x.l := y$	assignment
	$x$	variable
$p$	$::= (x : T) \rightarrow t$	abstraction
	$\overline{x y}$	application
	$\{\{\text{local}\} l = x\}$	record construction
	$x.l$	selection
	$t$	term
$T$	$::= (x : T) \xrightarrow{e}_{loc} T$	function type
	$\{\{\text{local}\} l : T\}$	record type
$e$	$::= \overline{x} \mid \text{any}$	effect annotation
$loc$	$::= \overline{x} \mid \text{any}$	locality annotation
$\Gamma$	$::= \overline{x : T \circ loc}$	variable typing environment

Figure 4.1: Language with Purity Effects (PUR)

The effect and locality annotations are either a list of variables  $\overline{x}$  or the unknown locality “any”. The symbol “ $\emptyset$ ” denotes the empty list: methods with an empty effect annotation are pure, an empty locality annotation describes methods that return fresh objects. A method with the “any” effect might modify any existing object and create arbitrary aliases in the heap. The locality annotation “any” describes methods that return objects with an unknown locality.

Effects and result localities each form a lattice with “any” as the top element, the join operator  $\sqcup$  is defined as follows:

$$\text{any} \sqcup e = e \sqcup \text{any} = \text{any} \quad \overline{x} \sqcup \overline{y} = \overline{x, y}$$

The following example illustrates the syntax of terms and types. It creates a counter and an increment function, updates the counter and returns its value (integers are assumed to be part of the language). The underscore “\_” is used as a substitute to avoid introducing unused variable names.

```

let c    = {x = 1}      in
let inc  = (_ : {}) →
                let v = c.x  in
                c.x := v + 1 in
let _    = inc {}      in
let r    = c.x         in
r

```

Since there is no Unit type in the PUR language, assignment expressions evaluate to the

assignee. Consequently, function *inc* has type “ $(\_ : \{\}) \xrightarrow{c} \{x : \text{Int}\}$ ” with latent effect  $c$  and return locality  $c$ . The invocation of *inc* has effect  $c$ , which is masked once  $c$  gets out of scope, so the overall example is typed as a pure expression.

Before introducing the typing rules in Section 4.3.2 and analyzing the example in more detail, we introduce the subtyping relation for PUR.

### 4.3.1 Subtyping

The subtyping relation presented in Figure 4.2 is reflexive and transitive.

The subtyping rule for function types S-FUN compares the two parameter types in contravariant fashion. For comparing the effects and result types we face the same problem as in the REL language in Chapter 3: the parameter names of the two function types need to be unified in some way.

Section 3.2.1 shows that in the case of REL, it is important to use the parameter symbol of the supertype when comparing the effects and the result types. For consistency we use the same technique for the PUR language, although in this case it does not matter which of the two parameters is used. The reason is that parameters are only compared symbolically, but never expanded according to their types as in the case of REL.

The subtyping rule S-FUN requires the effect, the result locality and the result type of the subtype to be more precise than in the supertype. This allows to use a function that always returns a fresh object to be used in places where a function with an arbitrary result locality is expected. The rule prevents a non-fresh function to be passed when a fresh function is required, as in the following example:

$$\begin{array}{l} \text{let } \mathit{init} = (f : \{\} \xrightarrow{\emptyset} \emptyset \{x : \text{Int}\}) \rightarrow \\ \quad \text{let } c = f \{\} \quad \quad \text{in} \\ \quad c.x := 1 \quad \quad \text{in} \\ \quad x \end{array}$$

The function *init* is pure because it only modifies object  $c$  which is fresh: the constructor  $f$  returns a fresh object. The subtyping rule ensures that *init* can only be invoked with fresh functions as arguments.

Subtyping for record types is mostly standard. The subtype needs to define at least the fields of the supertype but is allowed to include others. This is called *width subtyping* in [Pierce, 2002], Chapter 15.2. Because the fields of records are mutable, *depth subtyping* would be unsound (cf. Pierce [2002], Chapter 15-5), therefore the field types need to be in invariant order. Finally, the two record types need to agree on the “local” annotations on their common fields.

$$\begin{array}{c}
 \boxed{T' < T} \\
 \text{S-REFL} \frac{}{T < T} \qquad \text{S-TRANS} \frac{T' < S \quad S < T}{T' < T} \\
 \\
 \text{S-FUN} \frac{T_1 < T'_1 \quad [x/x']e' \sqsubseteq e \quad [x/x']loc' \leq loc \quad [x/x']T'_2 < T_2}{(x' : T'_1) \xrightarrow{e'}_{loc'} T'_2 < (x : T_1) \xrightarrow{e}_{loc} T_2} \\
 \\
 \text{S-REC} \frac{\bar{l} \subseteq \bar{l}' \quad \forall i. l'_i = l_i \Rightarrow (T'_i < T_i) \wedge (T_i < T'_i) \wedge ([\text{local}] l'_i = [\text{local}] l_i)}{\{[\text{local}] l' : T'\} < \{[\text{local}] l : T\}} \\
 \\
 \boxed{[loc_x/x]T} \\
 \frac{}{[loc_x/x]\{[\text{local}] l : T\} = \{[\text{local}] l : [loc_x/x]T\}} \\
 \\
 \frac{T = (y : T_1) \xrightarrow{e}_{loc} T_2 \quad y = x}{[loc_x/x]T = (y : [loc_x/x]T_1) \xrightarrow{e}_{loc} T_2} \\
 \\
 \frac{T = (y : T_1) \xrightarrow{e}_{loc} T_2 \quad y \neq x}{[loc_x/x]T = (y : [loc_x/x]T_1) \xrightarrow{[loc_x/x]e}_{[loc_x/x]loc} [loc_x/x]T_2} \\
 \\
 \boxed{loc' \leq loc} \\
 \frac{}{loc' \leq \text{any}} \qquad \frac{\bar{x}' \subseteq \bar{x}}{\bar{x}' \leq \bar{x}} \\
 \\
 \boxed{[loc_x/x]loc} \quad \frac{(loc = \text{any}) \vee (x \notin loc)}{[loc_x/x]loc = loc} \quad \frac{x \in \bar{x}}{[\text{any}/x]\bar{x} = \text{any}} \quad \frac{x \in \bar{x}}{[\bar{x}'/x]\bar{x} = (\bar{x} \setminus x), \bar{x}'} \\
 \\
 \boxed{e' \sqsubseteq e}, \boxed{[x/x']e} \text{ similar to } loc' \leq loc, [x/x']loc
 \end{array}$$

Figure 4.2: Subtyping for PUR

### 4.3.2 Typing Rules

The typing statement for an expression of the PUR language takes the form  $\Gamma \vdash p : T \circ loc ! e$ . It assigns a type  $T$ , a locality  $loc$  and an effect  $e$  to the expression  $p$ .

An effect  $\bar{x}$  can be understood as a requirement for the expression to be pure: if all variables in  $\bar{x}$  hold fresh objects, then the expression can only modify fresh state and does not have an observable effect. Similarly, a locality  $\bar{x}$  says that the expression evaluates to a fresh value if all variables in  $\bar{x}$  are fresh.

To illustrate how effects and localities are tracked in the typing rules we analyze the example

$$\boxed{\Gamma \vdash p : T \circ loc ! e}$$

$$\begin{array}{c}
 \text{T-PARAM} \frac{x : T \circ loc \in \Gamma}{\Gamma \vdash x : T \circ loc ! \emptyset} \\
 \\
 \text{T-SUB} \frac{\Gamma \vdash p : T' \circ loc' ! e' \quad T' <: T \quad loc' \leq loc \quad e' \sqsubseteq e}{\Gamma \vdash p : T \circ loc ! e} \\
 \\
 \text{T-ABS} \frac{\Gamma, x : T_1 \circ x \vdash t : T \circ loc ! e}{\Gamma \vdash (x : T_1) \rightarrow t : (x : T_1) \xrightarrow{e}_{loc} T \circ any ! \emptyset} \\
 \\
 \text{T-APP} \frac{\Gamma \vdash f : (x : T_1) \xrightarrow{e}_{loc} T_2 \circ any ! \emptyset \quad \Gamma \vdash a : T_1 \circ loc_a ! \emptyset}{\Gamma \vdash f a : [loc_a/x] T_2 \circ [loc_a/x] loc ! [loc_a/x] e} \\
 \\
 \text{T-LET} \frac{\Gamma \vdash p : T_1 \circ loc_1 ! e_1 \quad \Gamma, x : T_1 \circ x \vdash t : T_2 \circ loc_2 ! e_2}{\Gamma \vdash \text{let } x = p \text{ in } t : \text{elim}(x, T_2) \circ [loc_1/x] loc_2 ! e_1 \sqcup [loc_1/x] e_2} \\
 \\
 \text{T-REC} \frac{\overline{\Gamma \vdash x : T \circ loc ! \emptyset} \quad loc_r = \bigsqcup_i \begin{cases} loc_i & \text{if local } l_i \\ \emptyset & \text{otherwise} \end{cases}}{\Gamma \vdash \{\overline{[local] l = x} : \overline{[local] l : T}\} \circ loc_r ! \emptyset} \\
 \\
 \text{T-SELECT} \frac{\Gamma \vdash x : \{\overline{[local] l : T}\} \circ loc_x ! \emptyset \quad loc = \begin{cases} loc_x & \text{if local } l_i \\ any & \text{otherwise} \end{cases}}{\Gamma \vdash x.l_i : T_i \circ loc ! \emptyset} \\
 \\
 \text{T-ASSIGN} \frac{\Gamma \vdash x : \{\overline{[local] l : T}\} \circ loc_x ! \emptyset \quad \Gamma \vdash y : T_i \circ loc_y ! \emptyset \quad e = \begin{cases} loc_x \sqcup loc_y & \text{if local } l_i \\ loc_x & \text{otherwise} \end{cases}}{\Gamma \vdash x.l_i := y : \{\overline{[local] l : T}\} \circ loc_x ! e}
 \end{array}$$

$$\boxed{\text{elim}(x, T) = \text{elim}(x, T, any)}$$

$$\frac{T = \{\overline{[local] l : T'}\}}{\text{elim}(x, T, loc) = \{\overline{[local] l : \text{elim}(x, T', loc)}\}}$$

$$\frac{T = (y : T_1) \xrightarrow{e}_{loc} T_2 \quad y \neq x \quad loc'_x = \text{if } (loc_x = any) \emptyset, \text{ else any}}{\text{elim}(x, T, loc_x) = (y : \text{elim}(x, T_1, loc'_x)) \xrightarrow{[loc_x/x]e}_{[loc_x/x]loc} \text{elim}(x, T_2, loc_x)}$$

Figure 4.3: Typing Rules for PUR



from Section 4.3:

$$\begin{array}{l}
 \text{let } c \quad = \{x = 1\} \quad \text{in} \\
 \text{let } inc \quad = (\_ : \{\}) \rightarrow \\
 \quad \quad \quad \text{let } v = c.x \quad \text{in} \\
 \quad \quad \quad c.x := v + 1 \quad \text{in} \\
 \text{let } \_ \quad = inc \{\} \quad \text{in} \\
 \text{let } r \quad = c.x \quad \text{in} \\
 r
 \end{array}$$

In the typing rule for object literals T-REC, the final locality joins all localities of the values stored in local fields. In other words, an object literal is fresh if all local fields contain fresh values.

The object literal  $\{x = 1\}$  is therefore fresh: the locality of the number 1 is irrelevant since it is stored in a non-local field. By typing rule T-LET, the typing environment for the let-body is extended with  $c : \{x : \text{Int}\} \circ c$ .

In the body of function *inc*, the assignment to  $c.x$  has effect  $c$ : for an assignment to a non-local field, the rule T-ASSIGN sets the effect to be the locality of the modified object. The increment function thus has type  $(\_ : \{\}) \xrightarrow{c} \{x : \text{Int}\}$ .

The types and effects for the rest of the program are straightforward. The invocation of *inc* has effect  $c$  and the resulting value  $r$  has type  $\text{Int}$ . The application of typing rule T-LET for the outermost let binding finally defines the overall type and effect of the program. Variable  $c$  gets out of scope and is substituted by its initial locality, and we obtain the pure typing  $\text{Int} \circ \text{any} ! \emptyset$ :

$$\frac{\Gamma \vdash \{x = 1\} : \{x : \text{Int}\} \circ \emptyset ! \emptyset \quad \Gamma, c : \{x : \text{Int}\} \circ c \vdash \text{body}_c : \text{Int} \circ \text{any} ! c}{\Gamma \vdash \text{let } c = \{x = 1\} \text{ in } \text{body}_c : \text{elim}(c, \text{Int}) \circ [\emptyset / c] \text{any} ! [\emptyset / c] c}$$

The role of the meta-function “elim” is discussed later in Section 4.3.2.

To illustrate how ownership and local field updates are tracked in the type system we define a constructor function *newHC* that returns a fresh object containing a counter:

$$\begin{array}{l}
 \text{let } newHC \quad = (\_ : \{\}) \rightarrow \quad \text{type } (\_ : \{\}) \xrightarrow{\emptyset} \{\text{local } c : \{x : \text{Int}\}\} \\
 \quad \quad \quad \text{let } k = \{x = 0\} \text{ in} \\
 \quad \quad \quad \text{let } r = \{\text{local } c = k\} \text{ in} \\
 r
 \end{array}$$

For the record literal  $\{\text{local } c = k\}$ , typing rule T-REC assigns locality  $k$  since the value  $k$  is stored in a local field. When the variable  $k$  gets out of scope, the rule T-LET applies the substitution  $[\emptyset / k]$  and the function body is typed as fresh. The function *newHC* therefore has type  $(\_ : \{\}) \xrightarrow{\emptyset} \{\text{local } c : \{x : \text{Int}\}\}$ .

We introduce two type aliases: type  $K = \{x : \text{Int}\}$  and type  $H = \{\text{local } c : K\}$ . The following

## Chapter 4. A Type-and-Effect System for Purity

---

function is a setter which updates the counter field in an object of type  $H$ :

$$\text{let } \text{setC} = (hc : H) \rightarrow \text{type } (hc : H) \xrightarrow{\emptyset}_{\text{any}} (k : K) \xrightarrow{hc, k}_{hc} H \\ \text{let } g = (k : K) \rightarrow hc.c := k \text{ in} \\ g$$

Because the setter assigns to a local field, its effect includes the localities of both the modified object  $hc$  and the stored object  $k$ . We examine a program which creates a new object holding a counter, changes the counter value and finally resets it using the setter:

$$\text{let } h = \text{newHC } \{ \} \quad \text{in} \\ \text{let } \_ = (\text{let } z = h.c \text{ in } z.x := 2) \quad \text{in} \\ \text{let } s = \text{setC } h \quad \text{in} \\ \text{let } r = s \{x = 0\} \quad \text{in} \\ r$$

The effect of the assignment  $z.x := 2$  in the second line is  $z$ . By rule T-SELECT the initial locality of  $z$  is  $h$  because it is defined as a selection of a local field of object  $h$ . When the variable  $z$  gets out of scope, the typing rule T-LET replaces it by its initial locality, therefore the second line has the effect  $h$ .

The setter function  $s$  is defined as a partial application of the curried setter  $\text{setC}$ . The typing rule T-APP computes the type of  $s$  by applying the substitution  $[h/c]$  to the result type of  $\text{setC}$ , which yields the function type  $(k : K) \xrightarrow{h, k}_h H$ . The invocation of the setter  $s$  to the fresh object  $\{x = 0\}$  therefore has the effect  $[\emptyset/k](h, k) = h$ .

The rule T-LET computes the effect of the let binding for variable  $h$ , which is the overall effect of the example, by substituting references to  $h$  by its initial locality, i.e.,  $\emptyset$ . Consequently the example type checks as pure.

### Closures With Effects on Captured Variables

In typing rule T-LET, the meta-function “elim” is used to eliminate references to the bound variable in the result type, or more precisely, in effect and locality annotations of function types in the result type. In the following example the function  $g$  has type  $(\_ : \{ \}) \xrightarrow{\emptyset}_c \{x : \text{Int}\}$ :

$$\text{let } f = \text{let } c = \{x = 0\} \text{ in} \\ \text{let } g = (\_ : \{ \}) \rightarrow c \text{ in} \\ g$$

Even though  $f$  is defined to be the same as  $g$ , its type cannot be the same because the local variable  $c$  is not in scope. The intuitive solution is to substitute the initial locality  $\emptyset$  for  $c$  in the type of  $g$  when variable  $c$  gets out of scope. This solution is however unsound: the function  $f$

does not return a fresh object on each invocation, instead it always returns the same object. The following example illustrates this:

```

let  $a = f$  {}      in
let  $o = \{\text{local } l = a\}$   in
let  $b = f$  {}      in
 $b.x := 2$ 

```

If the function type of  $f$  has a fresh result locality, the last two lines of the example type check as pure:  $b$  has the initial locality  $\emptyset$ , and the effect on  $b$  from the last line is masked. However, given the definition of  $f$  above, the last line *does* modify existing state, namely the locality of object  $o$ .

This problem is addressed by the meta-function “elim”, which eliminates references to captured variables from effect and locality annotations in the typing rule T-LET. In covariant positions, references to the variable are replaced by “any”, in contravariant by  $\emptyset$ . The type of function  $f$  is therefore  $(\_ : \{\}) \xrightarrow{\emptyset}_{\text{any}} \{x : \text{Int}\}$  and the last two lines of the above example are typed as impure.

### 4.3.3 Typing PUR Requires ANF

The reason for using terms in A-normal form in the REL language is related to the fact that variable names are used as abstract localities to describe the effects of a term. These variable names can be tracked with little effort in the type system. In a language with non-ANF terms we could write the following definition:

$$\text{let } f = ((o : \{x : \text{Int}\}) \rightarrow (\_ : \{\}) \rightarrow o) \{x = 1\}$$

The function  $f$  returns the same object on every invocation, so using the result locality  $\emptyset$  in its type is unsound as shown in the previous section.

Using the safe type  $(\_ : \{\}) \xrightarrow{\emptyset}_{\text{any}} \{x : \text{Int}\}$  for function  $f$  leads to an over-approximation of the inferred effect in some situations:

```

let  $r$  = let  $f = ((o : \{x : \text{Int}\}) \rightarrow (\_ : \{\}) \rightarrow o) \{x = 1\}$   in
          let  $o = f$  {}                                          in
           $o.x := 2$ 

```

In this example,  $r$  evaluates to the fresh object  $\{x = 2\}$  and its initialization has no observable effects. However, given the above function type for  $f$ , the locality of  $o$  is “any” and consequently the assignment in the last line type checks as impure. In ANF, the object  $\{x = 1\}$  would be bound to a local variable and the last line would have the effect of modifying that specific object.

Note that in the typing rules T-ABS and T-LET, the new variable  $x$  is always entered into the typing environment with locality  $x$  for typing the subterm. This is required for soundness of the type system, as shown by the following example:

```
let a = {x = 1}      in
let o = {local f = a} in
a.x := 2
```

If the variable  $a$  is entered into the typing environment with its initial locality,  $a : \{x : \text{Int}\} \circ \emptyset$ , the assignment is typed as pure even though it modifies the locality of  $o$ .

When type checking a source program, a variable binding  $x$  in the typing environment always has locality  $x$ , so the localities could be removed from the typing environment in principle. However the localities in typing environments are used to correctly type check intermediate terms that occur during evaluation of terms, as shown in the ongoing work on proving soundness for the type system of PUR [Rytz et al., 2013].

### 4.4 Implementation of the Purity System for Scala

All examples from the introduction of Section 4.2 are valid Scala programs and use the syntax for effect annotations supported by the implementation. The purity type system for Scala is implemented in the form of a compiler plugin, as discussed in Chapter 5.

The overview section highlights the important aspects of the purity type system except for one. Unlike the formal language REL, mutable state in Scala comes in two flavors: mutable fields of objects and local variables defined inside methods. The type-and-effect system of REL models purity with respect to mutable fields, but the implementation in Scala also has to take into account updates to local variables. The next section describes how these assignment effects are handled in the type system.

#### 4.4.1 Assignment Effects

Since Scala supports both functional and imperative code, method implementations that use local variables have to be supported by any practical tool for effect analysis. The following example shows a method that computes the length of a list in imperative style:

```
def length[T](l: List[T]): Int @mod() = {
  var c = 0
  var ls = l
  while (ls != Nil) {
    c = c + 1
    ls = ls.tail
  }
}
```

```
    }  
    c  
  }
```

The method has no observable side effect, it only updates the local variables allocated within the method body.

At first glance, it seems that assignments are irrelevant for the effect of a method because by definition, updates to local variables are not observable from the outside. This is however not entirely correct: it is true that the assignments themselves cannot be observed from the outside, however assignments can change the locality of a variable and impact the scope of subsequent object modifications:

```
def incSmaller(a: Counter, b: Counter): Unit @mod(a, b) = {  
  var c = a  
  if (b.get < a.get)  
    c = b  
  c.inc()  
}
```

In this example the local variable *c* might either point to *a* or *b*, therefore the effect of the method includes both of its parameters.

The type system presented so far cannot express the effect of assignments to local variables: the annotation `@mod(c)` denotes the effect of modifying fields in the locality of object *c* and not the effect of re-assigning the variable. For this reason, the effect system introduces a specific effect annotation `@assign(c)` which denotes the effect of re-assigning a local variable *c*.

As explained above, the goal of tracking assignment effects is to know the potential localities that a variable might point to. This is achieved by attaching to each assignment effect the locality of the object stored in the variable. In the example method `incSmaller`, the initial assignment has effect `@assign(c, a)` and the second assignment has effect `@assign(c, b)`. Joining these two effects yields the overall assignment effect `@assign(c, a, b)` for the method body which reads as “variable *c* is assigned objects from the localities *a* and *b*”.

Assignment effects are consulted when a variable gets out of scope to substitute references to that local variable. In the example, the effect of invoking `c.inc()` is `@mod(c)`. This effect is not valid for the signature of method `incSmaller` because the variable *c* is not in scope in the signature. The locality *c* is therefore replaced by the locality that the variable might point to, which yields the desired effect annotation `@mod(a, b)`.

Assignment effects form a lattice with the following properties:

- `@assign()` is the bottom element denoting purity
- `@assign(any)` denotes impurity

## Chapter 4. A Type-and-Effect System for Purity

---

- The subeffect relation for a specific variable  $\text{@assign}(x, \text{loc}_1) \sqsubseteq \text{@assign}(x, \text{loc}_2)$  holds if  $\text{loc}_1 \leq \text{loc}_2$
- Effects are annotated as a sequence of  $\text{@assign}$  annotations each describing assignments to one local variable. The subeffect relation  $\text{@assign}(x, \text{loc}_x) \sqsubseteq \text{@assign}(y, \text{loc}_y)$  holds if  $\forall i. \exists j. \text{@assign}(x, \text{loc}_x)_i \sqsubseteq \text{@assign}(y, \text{loc}_y)_j$

A complete effect annotation in the purity domain consists of both a state effect  $\text{@mod}$  and an assignment effect  $\text{@assign}$ . For syntactic convenience, the system accepts effect annotations which only specify one of the two effects, in which case it will use the respective purity annotation  $\text{@mod}()$  or  $\text{@assign}()$  for the missing domain.

Since assignment effects are accumulated throughout the scope of a variable and only applied at the end, the localities of variables are tracked in flow-insensitive manner. This implies that the effect computed for a block of statements is an over-approximation if it depends on the order of assignments, as shown in the following example:

```
def counterGame(a: Counter, b: Counter): Int @mod(a, b) = {
  var c = a
  c.inc()
  if (b.get < c.get)
    c = b
  c.get
}
```

We can see that the counter  $b$  is never modified, but the system computes the effects  $\text{@mod}(c)$  and  $\text{@assign}(c, a, b)$  which expand to  $\text{@mod}(a, b)$  as in the previous example.

The implementation of the purity effect system uses assignment effects not only to track assignments to local variables but also to remember the initial localities of non-mutable local value bindings:

```
def incSmaller(a: Counter, b: Counter): Unit @mod(a, b) = {
  val c = if (a.get < b.get) a else b
  c.inc()
}
```

The `if` expression has locality  $\text{@loc}(a, b)$ , which is the join of the localities of its two branches. Therefore the initial assignment of the local value  $c$  has effect  $\text{@assign}(c, a, b)$ . To obtain the effect for the method signature, the local effect  $\text{@mod}(c)$  is expanded according to the assignment effect of variable  $c$ , which yields the expected  $\text{@mod}(a, b)$ .

In the examples shown so far, the assignment effects only occur within the body of a method and are inferred by the effect system. Top-level method cannot have assignment effects in their signatures because there cannot be any local variables defined their environment which they could modify. For this reason, assignment effects are usually transparent to the programmer.

Assignment effect annotations can only appear in signatures of *nested* definitions which act on their environment, like in the following example:

```
def outer: Int @mod() {
  var i = 0
  def inc(): Unit @assign(i, any) = {
    i = i + 1
  }
  inc()
  i
}
```

Note that the locality assigned to variable `i` is “any”: since primitive values are immutable, their locality is irrelevant, there cannot be any modification effects on a primitive value. Section 4.4.4 shows that the unknown locality “any” can safely be used for all objects that have immutable types.

### 4.4.2 Flow-Insensitivity to Support Higher-Order Code

The main reason why we designed the purity type system to be flow-insensitive is to make it compatible with nested definitions, higher-order functions and the generic framework for polymorphic effect checking introduced in Chapters 2 and 3.

Section 2.1.2 explains why a flow-sensitive effect system cannot easily be integrated with effect-polymorphism. The effect of each statement in a block of code potentially depends on the effects of previous statements. In an effect-polymorphic method which abstracts over the effect of a parameter, the remaining effect of that method has to be expressed in terms of the abstract effect, which complicates the annotation scheme. Computing the effect of an invocation of a polymorphic method is also more complex: effects cannot simply be joined because their order is significant.

The situation is similar for definitions of nested methods that have effects on their environment. In a flow-sensitive system, the effect of such a nested method depends on the type of the accessed outer variable. Since this type can change in the course of execution of the outer method, we need to express the effect of the nested method as a function of the variable’s type (or over-approximate it).

Our flow-insensitive annotation scheme for purity effects naturally extends to nested method definitions and higher-order functions. The annotations can simply refer to localities defined in the enclosing scope, e.g., `@mod(c)` in the following example:

```
def test(): Int @mod() = {
  val c = new Counter()
  def up(): Int @mod(c) = {
    c.inc()
  }
}
```

```
    }  
    up()  
}
```

In Section 4.2.4, we observed that flow-insensitivity of the purity effect system occasionally leads to an over-approximation of a method's effect:

```
def f(b: B) = {  
    val a = new A()  
    a.localField = b  
}
```

The body of method `f` only modifies the fresh object `a` and is therefore pure, but the type system infers effect `@mod(a, b)` for the assignment and therefore `@mod(b)` for the method. A flow-sensitive type system could keep track of the localities of the involved variables and be more precise.

Some of the precision of a flow-sensitive type system can be regained by introducing more expressive effects: an effect annotation `@store(hc, someCounter)` could express the fact that object `someCounter` is stored in the locality of `hc`, but only the locality of `hc` is modified.

In our evaluations presented in Section 4.4.4 and Chapter 5, we did not encounter any issues related to the over-approximation of assignment effects in the flow-insensitive system with basic `@mod` effects. Furthermore, the proposed `@store` annotation cannot fully recover the expressiveness of a flow-sensitive type system as shown by the following example. It is unclear if the additional complexity is justified.

```
def f1(a: A, b: B) = {  
    a.store(b)  
    a.modify()  
}  
  
def f2(a: A, b: B) = {  
    a.modify()  
    a.store(b)  
}
```

In a flow-insensitive system both methods have the effect `@store(a, b) @mod(a)`, but the order in which these effects occur is not encoded. A caller of method `f2` must assume that the locality of object `b` can be modified.

Note that the flow-insensitivity does not restrict assignments to an object's locality. Neither does it require the programmer to declare the locality that an object may point to. As explained in Section 4.2.4, the system allows assigning arbitrary values to the locality of an object and uses `@mod` effects to keep track of the assignments that occur in the scope of the variable.

### 4.4.3 Polymorphic Purity Effects

Since the purity effect system is flow-insensitive and both of its effects, `@mod` and `@assign`, form a lattice, the system can be integrated with the framework for polymorphic effect check-



ing presented in Chapter 3. This ensures correct effect inference for common higher-order programming patterns that involve local state.

As an example, we consider the implementation of a method that computes the length of a list using the higher-order method `foreach`:

```
def length[T](l: List[T]): Int @mod() = {
  var c = 0
  l.foreach(x => c = c + 1)
  c
}
```

Effect analysis of the method `length` works similar to the examples presented in Section 3.3.2: method `foreach` has the effect of its argument function, and the function literal has the refined type

```
(T => Unit) { def apply(x: T): Unit @assign(c, any) }.
```

The overall effect of the method body is therefore `@assign(c, any)`, which is masked once the variable `c` gets out of scope.

Polymorphic effect checking works in the same way for `@mod` effects. One example is the method `filter` of the Scala collections library presented in Section 3.3.3. More examples are discussed in Chapter 5.

Note however that effect-polymorphism does not abstract over the result locality `@loc` of a parameter method. Locality annotations are not effect annotations, instead they describe the freshness of the objects that a method returns. Effect-polymorphic methods do not abstract over the freshness of their parameter function, as illustrated in the following example:

```
type CounterFactory = (() => Counter) {
  def apply(): Counter @mod(any) @assign(any) @loc()
}
def buildNew(f: CounterFactory): Counter @pure(f) @loc() = {
  val c = f()
  c.inc()
  c
}
```

Because `buildNew` is effect-polymorphic in the `apply` method of its argument function, it can be marked as `@pure` even though the `apply` method may have arbitrary effects. The freshness annotation `@loc()` on the `apply` method is required for the example to type check. Without it, the value obtained by invoking `f()` would have an unknown locality, the modification `c.inc()` would have an unknown effect and the returned value `c` would not be fresh.

### 4.4.4 Examples and Limitations

In this Section, we evaluate the expressiveness of our purity effect system by analyzing the implementations of the core data structures in the Scala collections library.

Figure 4.4 shows the trait `TraversableLike` which implements the basic operations for immutable collections such as `map` or `filter`. Note that these operations are inherited by mutable collection classes and remain immutable operations: for instance, calling `map` on a mutable set returns a new mutable set with the transformed values and leaves the original set unchanged.

```
trait CanBuildFrom[-From, -Elem, +To] {
  def apply(): Builder[Elem, To] @pure @loc()
}

trait TraversableLike[+A, +Repr] {
  def foreach(f: A => Unit): Unit @pure(f)

  def newBuilder: Builder[A, Repr] @pure @loc()

  def filter(p: A => Boolean): Repr @pure(p) = {
    val b = newBuilder
    for (x <- this)
      if (p(x)) b += x
    b.result
  }

  def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That])
    : That @pure(f) = {
    val b = bf()
    for (x <- this) b += f(x)
    b.result
  }
}
```

Figure 4.4: Base Trait of Scala Collections

To infer the effect of method `filter` the type system first considers the invocation of method `newBuilder`. This method is pure and returns a fresh object which is assigned to `b`. The body of the `for` comprehension has the effect `@mod(b)` from the invocation of `+=` and it inherits the relative effect `@pure(p)` (see Section 3.3.3). The `for` comprehension desugars to an invocation of the effect-polymorphic method `foreach`, so it has the effect of its body. Since the invocation `b.result` is pure and the effect `@mod(b)` on the fresh object `b` can be masked, only the relative effect `@pure(p)` remains.

The implementation of method `map` uses one additional component, the implicit parameter of type `CanBuildFrom`, also called the “builder factory”. The builder factory allows method `map` to

be polymorphic in the resulting collection type, as explained in [Odersky and Moors, 2009, Chapter 6]: invoking the method `map` on a `BitSet` should return another `BitSet` if the result type `B` is `Int`. In all other cases, the return type of `map` is `Set[B]`.

In order to verify purity of the implementation of `map`, the `apply` method in class `CanBuildFrom` is required to return a fresh object, similar to the method `newBuilder` which is used in the implementation of method `filter`.

### Freshness of Immutable Values

Immutable values are common in Scala. For example, the Scala collections library provides various immutable collection types. Also strings and primitive values like integers are immutable.

In our purity type system, the locality assigned to an immutable value is irrelevant because it cannot be affected by effects. This implies that objects of immutable types can safely be assigned the unknown locality `@loc(any)`: there is no advantage in marking an immutable object fresh because the object cannot be modified and there are no effects that freshness could mask.

The advantage of using the unknown locality for immutable objects is that it enables structural sharing. As an example we look at the implementation of method `dropWhile` in the parent trait `TraversableLike`:

```
trait TraversableLike[+A, +Repr] {
  def dropWhile(p: A => Boolean): Repr @pure(p) = {
    val b = newBuilder
    var go = false
    for (x <- this) {
      if (!go && !p(x)) go = true
      if (go) b += x
    }
    b.result
  }
}
```

The method skips all elements until `p(x)` becomes false and then creates a new collection containing the remaining elements. In class `List`, which extends `TraversableLike`, the method can be implemented more efficiently: thanks to immutability, it is not necessary to copy the list and we can safely return a tail of the current list.

```
class List[+A] extends TraversableLike[A, List[A]] {
  override def dropWhile(p: A => Boolean): List[A] @pure(p) = {
    @tailrec def loop(xs: List[A]): List[A] @pure(p) =
      if (xs.isEmpty || !p(xs.head)) xs // no copying required
  }
}
```

```
        else loop(xs.tail)
      loop(this)
    }
  }
```

Because the object returned by method `dropWhile` in class `List` is aliased from the current list `this`, the method cannot be annotated `fresh`. For type soundness, overriding rules enforce that the result type of an overriding method is a subtype of the result type in the superclass. Consequently the method `dropWhile` cannot be annotated `fresh` in the base trait `TraversableLike` as this would rule out the optimized implementation in class `List`<sup>2</sup>.

For mutable collections on the other hand, the operations defined in trait `TraversableLike` are expected to return freshly allocated, non-aliased results. Using the purity type system we can statically check this implementation constraint by annotating the methods as `fresh`. In order to be able to annotate the method it has to be overridden in the mutable collection implementation:

```
class HashSet[A] extends TraversableLike[A, HashSet[A]] {
  override def dropWhile(p: A => Boolean): List[A] @pure(p) @loc() = {
    // ...
  }
}
```

The problem is that the override cannot simply invoke `super.dropWhile(p)`: because the implementation in the parent is not marked `fresh`, that invocation leads to an effect type error, namely the localities do not match.

To avoid code duplication, the implementation of `dropWhile` in parent trait `TraversableLike` can be refactored to delegate to a protected helper method `dropWhileImpl` which is annotated `fresh`. In the subclass `HashSet`, the override can then invoke the fresh method `dropWhileImpl` and type check as `fresh`.

While the type system in its current form is capable of expressing the freshness properties of the mentioned methods, the fact that code has to be refactored is not ideal. One direction to improve the situation is to enhance the expressiveness of freshness annotations, possibly with freshness parameters.

Another idea is based on the observation that the locality of immutable objects is irrelevant, since they cannot be mutated. This implies that treating any immutable object as `fresh` is safe, even if there exist unknown aliases to it: freshness of an object can only be used to mask modification of that object. Since immutable objects cannot be changed, marking them `fresh` cannot lead to any additional effect masking.

---

<sup>2</sup> For annotating `dropWhile` `fresh` in the base trait `TraversableLike`, the method `result` in trait `Builder` would be required to have result locality `@loc(this)`. While this is desirable for builders of mutable collections, it is too restrictive for builders of immutable collections and leads to the issues as described in the next subsection, 4.4.4.

With this observation, the method `dropWhile` in class `List` could safely be annotated `fresh` because it returns an immutable object. To enable the type system to reason about immutability of objects, techniques similar to those introduced by Tschantz and Ernst [2005] could be incorporated.

### Immutable Objects are Used as Fresh

The fact that immutable objects can safely be shared is not only used in the context of immutable data structures, but in some cases also for mutable classes. For example, because iterators are mutable objects, the method `iterator` of a collection class is required to return a fresh object on each invocation, i.e., it is annotated `fresh`.

For empty collections it is safe to always return the same global object `Iterator.empty` because this object does not have any mutable state. One example is the class `immutable.HashMap` which only represent empty maps. Non-empty maps are instances of one of its subtypes:

```
class HashMap[A, +B] {  
  def iterator: Iterator[(A,B)] @pure @loc() = Iterator.empty  
}
```

This definition produces an effect type error because the returned object is not fresh, therefore an effect cast is required. We could solve this issue with the extension mentioned in the previous section that enables the type system to track immutability of objects.

### Collections Containing Fresh Objects

Most of the examples discussed in this chapter describe implementations of pure methods that use local mutable state. For example, a mutable builder is used in `filter` and `map` to collect the elements of the resulting collection. In the examples discussed up to this point, the algorithm only modifies the builder itself, but never the elements stored inside the builder. To verify purity of the implementation of `filter`, the type system needs to know the freshness of the builder, but the locality of the accumulated elements is not relevant.

There are however situations in which an algorithm stores freshly allocated, mutable objects in a data structure and modifies them later on. One example is the implementation of method `groupBy` in trait `TraversableLike`:

```
1 trait TraversableLike[+A, +Repr] {  
2   def groupBy[K](f: A => K): immutable.Map[K, Repr] @pure(f) = {  
3     val m = mutable.Map.empty[K, Builder[A, Repr]]  
4     for (elem <- this) {  
5       val key = f(elem)  
6       val bldr = m.getOrElseUpdate(key, newBuilder)
```

## Chapter 4. A Type-and-Effect System for Purity

---

```
7     bldr += elem
8   }
9   val b = immutable.Map.newBuilder[K, Repr]
10  for ((k, bldr) <- m)
11    b += ((k, bldr.result))
12  b.result
13 }
14 }
```

For each element of the current collection, line 5 computes the key in the resulting map using the parameter function `f`, and line 7 adds the element to a builder which corresponds to that key. The builders for each key are freshly created using `newBuilder` when necessary and stored in the local mutable map `m`.

However, the locality of the builder object `bldr` obtained in line 6 is unknown: the method `getOrElseUpdate` in the map class is not annotated `fresh` because that would not allow the function to return any existing object. For this reason, the effect system infers the effect `@mod(any)` for the invocation `bldr += elem`.

With the type system presented in this chapter, the issue can only be circumvented by casting the locality of `bldr` to `@loc()`. In the future, we plan to address the problem by allowing locality annotations to appear on type parameters. For instance, an optional value which is only allowed to store freshly allocated values of type `T` would have type `Option[T @loc()]`. A similar solution is used in the work by Dietl et al. [2007] on Generic Universe Types where type parameters can be instantiated according to the ownership of the stored value. For instance, the type `rep Set<rep T>` denotes a set, owned by the current object `this`, which can only contain objects owned by `this`.

### Caches

Cache update operations are side effects that should be transparent to clients of a function. For example, the implementation of a recurring calculation could be improved by caching results. However, the process of updating a cache is a side effect on program state that is not allocated within the function and the function cannot be considered pure by our effect system.

There are multiple examples of caching in the context of the Scala collections library, for instance in the implementation of immutable hash maps. Elements of the hash map are key-value pairs represented as instances of the class `HashMap1`:

```
class HashMap1[A,+B](key: A, hash: Int, value: B, var kv: (A,B))
```

The variable `kv`, initially set to `null`, is a cache used to improve the performance of traversals of the hash map. Method `foreach` takes a function of type `(A, B) => Unit` as argument. In order to invoke the argument function, a new object of type `Tuple2` has to be allocated for each

key-value pair in the map. Since most operations on collections are implemented in terms of `foreach`, these tuples are cached to avoid re-creating them for each traversal. This implies that method `foreach` has not only the side effect of its argument function, but additionally the effect of modifying the cache.

The purity type system presented in this chapter cannot verify the fact that modifications to the cache are not observable for clients that use an immutable hash map, and it requires an effect cast on method `foreach`. The problem is that the cache is not local state which is allocated and modified within a method; instead, the same state is accessed and modified across multiple method calls.

A second example for a collection implementation that uses caching is the class `Vector` in which the elements are stored in nested arrays as explained by Bagwell and Rompf [2011]. Every access to an element involves an array lookup at each nesting level. Since vector elements are often accessed in sequence, the lookup operation caches a pointer to the array at each level. If the index of a subsequent lookup uses the same trace of arrays, the intermediate lookup operations do not need to be performed again. Like in the previous example, the effect of writing the caches in the lookup method cannot be masked automatically and has to be handled with a cast.

For convenience, the use of caches could be simplified by introducing an annotation `@cache` which designates state that is used only for caching. The type system could then allow updates to caches in pure contexts. This solution, which is equivalent to using effect casts, hence unsound, is used in the work on reference immutability by Tschantz and Ernst [2005]. Verifying that caches cannot change the outcome of operations is beyond the scope of the type system presented in this chapter and requires advanced static analysis with global knowledge.

### Concrete Collection Classes

Most immutable data structures in Scala use structural sharing so that update operations return a new data structure which shares parts of the representation with the original instance.

One example are tree sets and tree maps based on immutable red-black trees that admit arbitrary sharing. Hash sets, hash maps and vectors on the other hand are implemented using nested mutable arrays, so called hash tries. Update operations create a copy of the arrays that need to be changed to represent the updated data structure, however the unchanged arrays are shared between multiple instances. Since state modifications are performed on arrays that are cloned within a method (copy-on-write), the modification effects can be masked and the effect system is able to verify purity of update operations.

A special case is the `List` class which is implemented using two subtypes, the object `Nil` and the class for list nodes `::`. Surprisingly, the fields that store the current element and the tail of the list in class `::` are not stable values, but variables:

## Chapter 4. A Type-and-Effect System for Purity

---

```
final case class ::[B](private var hd: B, private[scala] var tl: List[B])
extends List[B] {
  def head : B = hd
  def tail : List[B] = tl
}
```

One reason is that the list class uses a custom, more compact serialization format and therefore needs the ability to write the fields on deserialization.

There is another reason for the `tl` field to be mutable: it enables implementing the mutable `ListBuffer` class with constant-time `prepend`, `append` and `result` methods. The class appends elements to the list that is being constructed by creating new `::` nodes and updating the `tl` field of the current list. The `result` method returns a pointer to the constructed list, but also marks the list buffer as “exported”. When an element is appended to an exported list buffer, the internal list is first cloned in order to avoid side effects on the previously exported list.

The operations on the list buffer only modify its internal state, but since list nodes have unknown localities, the effect system cannot express this property. The implementation of `ListBuffer` requires a number of effect cast annotations.

Mutable collections are typically implemented using private state which is not shared across instances. Ownership annotations enable the effect system to show that the effect of an update is contained within an instance of a mutable collection.

Mutable hash sets and hash maps are implemented using hash tables, which store the elements of the collection in an array indexed by hash codes. The field holding the array is annotated `@local` which enforces that an array cannot be shared across instances. The update operations perform modifications on that array and are annotated `@mod(this)`. If the array needs to be resized to accommodate more elements, a new array is created, initialized and assigned to the local field in the hash table. This is also allowed by the effect annotation `@mod(this)`.

Mutable linked lists have a variable field in each node that points to the next list element. This field is annotated `@local`, which implies that all nodes reachable from the current node are part of its locality.

```
class LinkedLst[T] {
  var elem: T = _
  @local var next: LLst[T] = this

  def append(that: LinkedLst[T]): Unit @mod(this, that) = {
    if (isEmpty) {
      elem = that.elem
      next = that.next
    } else {
      if (next.isEmpty)
        next = that
      else
    }
  }
}
```



```

        next.append(that)
    }
}

```

The effect of `append` includes both objects `this` and `that` because the argument object is stored in the locality of the current list. This is explained in Section 4.2.4.

## 4.5 Related Work

The type system presented in this chapter is strongly influenced by JPure, a purity system for Java by Pearce [2011]. Our notions of *locality* and *freshness* are equivalent to theirs.

In contrast to our system, JPure tracks the freshness of local variables in a flow-sensitive manner. While being more precise in some situations, a flow-sensitive system is difficult to integrate into a language with nested methods and higher-order code. Based on our experience with Scala, we believe that the additional precision is not essential in practice. Note that flow-insensitivity does not restrict the values that can be assigned to a local variable:

```

var x = a
if (condition) x = b
x.modify()

```

Even though the local variable `x` initially has locality `a`, assigning `b` to it is allowed. The effect system uses the effect `@assign(x,b)` to keep track of the locality of `x`. More precisely, the overall effect of the three statements is `@assign(x,a) @assign(x,b) @mod(x)`. When variable `x` gets out of scope, this effect is translated to `@mod(a, b)`.

The flow-insensitivity leads to imprecise effect inference in some situations. Consider the above example where the last two lines are switched:

```

var x = a
x.modify()
if (condition) x = b

```

Effect inference applies the exact same procedure as in the original example and the computed effect is still `@mod(a, b)` even though `b` is not modified.

Another difference when comparing with JPure is that our system allows effect and locality annotations to refer to parameters and variables from the enclosing scope. This is essential for integration in higher-order languages. Effect annotations in JPure can only refer to a method's parameters.

A method annotated `@Fresh` in JPure will always return a freshly allocated object, while all

other methods return objects with unknown locality (annotated `@loc(any)` in our system). Our type system allows defining the locality of the returned object more precisely, which enables expressing the behavior of getters for local fields. If an object is known to be fresh, then the values stored in its local fields are guaranteed to be fresh as well. The getter of a local field returns a fresh object *if* the owner object is fresh, which is expressed as `@loc(this)` in our system. In JPure the getter is annotated non-fresh, therefore modifying a local field of a fresh object has the unknown effect if the field is accessed through its getter.

In addition the purity effect system, JPure provides an inference tool that analyzes existing libraries for pure methods and generates effect and freshness annotations. They applied their inference algorithm to a subset of the Java library and show that 40% of the analyzed methods are pure.

### 4.5.1 Regions

In the history of type-and-effect systems, the most widely used solution to control aliasing are *regions*. In a such an effect system, the store is split up in multiple distinct areas, so called regions, and each reference is allocated within a specific region. Side effects are expressed as read, write and allocation effects on those regions. Effect systems based on regions include the original polymorphic effect system by Lucassen and Gifford [1988], the work on type, region and effect inference by Talpin and Jouvelot [1992b] and more recently the effect system for deterministic parallel Java (DPJ) by Bocchino et al. [2009]. The process of hiding non-observable effects on locally allocated state was given the name *effect masking* in the article of Lucassen and Gifford [1988].

The annotation overhead of region based effect systems is typically large because functions that modify or allocate state need to be extended with region parameters. For languages like Scala, global type, region and effect inference is not a viable solution because it would be a departure from the fundamental concepts of the language. Furthermore, type inference in the presence of subtyping is known to be complex and leads to long type signatures and error messages that are difficult to understand. Hoang and Mitchell [1995] discuss the difficulties that subtyping causes to type inference. Another challenge for type inference is overloading, as explained by Smith [1994].

### 4.5.2 Ownership Types

A different approach to control aliasing is presented in the work on *ownership types*, first introduced by Clarke et al. [1998]. In an ownership type system, the *representation* of an object, a subset of the state that is accessible through its fields, is *owned* by the object. Access to the representation of an object is only allowed through that object, but not directly from the outside. The type system prevents aliases from the outside into the representation of an object.

Clarke and Drossopoulou [2002] introduce an effect system based on ownership types that can show non-interference of expressions whose effects are distinct.

Similar to ownership types, our system uses `@local` annotations to denote the internal representation of objects. The fundamental difference is that ownership type systems enforce a global aliasing discipline which ensures that private state can never be accessed from the outside. This enables a strong guarantee: when modifying the representation of an object at any point in time, the system ensures that objects outside the representation remain unchanged. Our effect system allows arbitrary aliasing and does not have any knowledge about the global structure of the heap. Instead it exploits local knowledge about the freshness of objects, which only holds if the entire representation of the object is known to be fresh. Modifications to fresh objects in a function are masked because they are not observable by its clients.

The strict encapsulation in ownership types has proven to be too restrictive, i.e., common programming patterns cannot be expressed, for example the use of an iterator. Boyapati et al. [2003] present a variant of ownership types in which objects of a nested class have privileged access to its ancestors in the ownership tree and objects they own. Their system can express constructs like iterators that share owned state while remaining modularly checkable.

In their work on universe types, Dietl et al. [2007] take a different approach to lift the restrictions of ownership type systems. The original ownership type system enforces the *owner-as-dominator* discipline in which each reference to an object has to go through its owner. Universe types propose a more permissive *owner-as-modifier* discipline where arbitrary references are allowed, however modifications to an object are only allowed through its owner. The system features pure methods which are, like in our effect system, methods that do not modify any state that existed before their execution. Similar to ownership types, universe types provide a stronger guarantee than our system but are coupled with a global aliasing discipline.

### 4.5.3 Pointer Analysis

There exist a number of purity systems based on pointer analysis. The goal of pointer analysis, or points-to analysis, is to determine the set of objects that a variable or field might point to. A recent article by Sridharan et al. [2013] presents an overview on the state of the art in pointer analysis for object-oriented programs.

Sălcianu and Rinard [2005] present a purity system based on a combined pointer and escape analysis. Their analysis can distinguish objects allocated within a method from objects that existed before the method was invoked. Consequently, non-observable effects like the use of an iterator can be masked. Based on the results of the analysis, their system generates regular expressions that characterize the heap locations a method modifies.

Kneuss et al. [2013] present a static analysis for side effects that is precise even in the presence of callbacks, which are common in languages with higher-order functions. Analyzing higher-order code is challenging because the number of targets for the invocation of a parameter function is potentially very large. Their system represents effects as control flow graph summaries which may contain *delayed* method calls. These summaries essentially express effect-polymorphism similar to relative effects presented in Chapter 3.

Purity systems based on pointer analysis have a different focus than type-and-effect systems. Pointer analyses are interprocedural and typically analyze entire programs at once. There are attempts to make program analyses more modular. The recent work by Ali and Lhoták [2013] shows how libraries can be analyzed separately from programs that are using them. Instead of creating custom signatures, their system generates lightweight replacement binaries that can be used with existing analysis tools instead of the original libraries. Another example is the modular effect analysis by Cherem and Rugina [2007] which analyzes libraries separately and generates effect and aliasing summaries. Type systems are modular at the level of individual methods. Verification of a method implementation is based solely on the signature of invoked methods. Modular verification is known to scale well to large programs, while scaling can be challenging for whole program analyses.

Another difference is that the effect systems presented in our thesis focus on lightweight effect annotations that programmers are willing to write and able to understand. The raw effect information obtained through alias analysis is too verbose to be presented to programmers.

The main strength of program analysis is that it does not require any effect annotations and can be used to perform effect inference. The inference tool for JPure by Pearce [2011] makes use of existing techniques for program analysis to generate effect annotations for large libraries such as the Java standard library.

### 4.5.4 Other Related Work

*Javari* by Tschantz and Ernst [2005] is a type system for tracking *reference immutability* for Java. If a field or local variable is annotated `readonly`, then that reference cannot be used to modify any of the objects that are transitively reachable through the reference. The system however allows other references to the same object to exist, which can be mutable references. Reference immutability can ensure that a method does not modify its argument, or that an object or array returned by a method can never be modified by its clients. This guarantee can prevent unnecessary copying of exposed state.

Huang et al. [2012] present a purity type system based on reference immutability. Their inference system scales to large programs and is able to discover a significant number of pure methods in existing Java libraries using an effect inference tool. The type system cannot express the notion of returning a freshly allocated object, therefore patterns like the use of an iterator are in principle impure. In the implementation they use a small number of special

cases to deal with some of these situations.

Finifter et al. [2008] present a purity system for Java that uses a stronger notion of purity: in addition to the absence of side-effects, a pure method is required to be deterministic. When a pure method is invoked with equal arguments, then it always has to return the same result. Equality of objects can be defined by the user; the notion of determinism is parameterized by the equality definition. The parameters of a pure method are marked *immutable*, which ensures that no object obtained through a parameter can be modified within the method. The system allows pure methods to modify freshly allocated state: when invoking an impure method from the body of a pure method, the argument objects are required to be mutable. Since all parameters of the pure method are immutable, these arguments are necessarily freshly allocated. Unlike our work, their system does not track freshness across methods.

The state monads described by Launchbury and Peyton Jones [1994] can encapsulate local state within the implementation of an externally pure algorithm. The technique of using monads to control effects is described in Section 1.3.2.

There is a large body of related work in the area of program verification, which we discuss in Section 1.3.4.

## 4.6 Conclusion

In this chapter we presented a type-and-effect system for verifying purity of methods that do not modify program state that existed before their execution. The effect system builds on ideas from JPure, a purity system for Java by Pearce [2011].

State modification effects are difficult to track because of aliasing. Instead of tracking aliases in a program globally, our purity system only identifies state that is freshly allocated within a method and masks effects on that fresh state. To identify fresh state, the effect system uses *ownership* annotations on fields and *locality* annotations on methods that describe the freshness of returned objects. The effect system is flow-insensitive, which makes it suitable for languages with higher-order functions and nested definitions such as Scala.

We integrated the effect system as an effect domain into the generic framework for effect checking from Chapters 2 and 3. The implementation for Scala can verify purity of programming patterns that mix higher-order code with side effects, such as using an iterator or copying a collection using a mutable builder.

In the future we plan to complete the preliminary work proving soundness of the effect system [Rytz et al., 2013]. We plan to improve the expressiveness of the purity system by enabling locality annotations on type arguments, which allows tracking fresh objects stored in data structures as explained in Section 4.4.4. Finally, an inference tool that generates effect annotations for existing code would reduce the initial effort of adopting our effect system.

## Chapter 4. A Type-and-Effect System for Purity

---

Such a tool can be based on the work by Kneuss et al. [2013] which performs global effect inference for purity and effect-polymorphism.

## Chapter 5

# Effect Checking in Scala

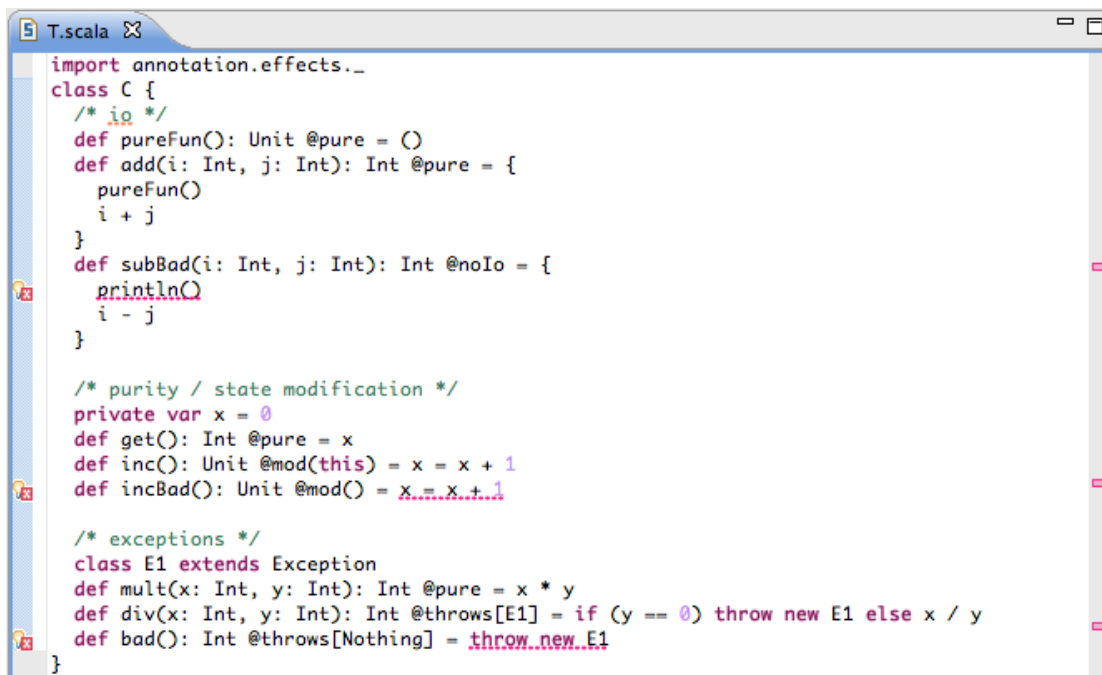
We combined the type-and-effect systems presented in this dissertation into an implementation in the form of a compiler plugin for the Scala programming language. Specifically, this effect system implements the generic effect system for multiple effect domains presented in Chapter 2 where effect-polymorphic functions are expressed using relative effect annotations like in Chapter 3 and it incorporates the effect system for purity presented in Chapter 4.

All code examples presented in the previous chapters are valid Scala programs and use the syntax for effect annotations supported by the implementation. This chapter explains how programmers can use the effect system to verify side effects in their code, how it can be extended to support new effect domains and how effect checking is integrated into the Scala compiler.

### 5.1 Programming With Effects

The implementation of our effect system for Scala has the form of a compiler plugin compatible with an official release of the Scala compiler (version 2.10.2 at the time of writing). The plugin can be obtained from its project page which gives access the source code, up-to-date information and an issue tracker for bug reporting; the project page is located at <https://github.com/lrytz/efftp>.

The Scala compiler has multiple modes of operation, all supported by the compiler plugin. The batch compilation mode is used for normal compilation on the command line, when using a build tool or when building a project within a development environment such as Eclipse, IntelliJ IDEA or Netbeans IDE. By default, effect mismatch errors are reported by the compiler on the command line, but IDEs typically intercept error messages and provide links to the erroneous locations in the source files.

The image shows a screenshot of the Scala IDE for Eclipse. The window title is 'T.scala'. The code is as follows:

```
import annotation.effects._
class C {
  /* io */
  def pureFun(): Unit @pure = ()
  def add(i: Int, j: Int): Int @pure = {
    pureFun()
    i + j
  }
  def subBad(i: Int, j: Int): Int @noIo = {
    println()
    i - j
  }

  /* purity / state modification */
  private var x = 0
  def get(): Int @pure = x
  def inc(): Unit @mod(this) = x = x + 1
  def incBad(): Unit @mod() = x...x+1

  /* exceptions */
  class E1 extends Exception
  def mult(x: Int, y: Int): Int @pure = x * y
  def div(x: Int, y: Int): Int @throws[E1] = if (y == 0) throw new E1 else x / y
  def bad(): Int @throws[Nothing] = throw new E1
}
```

Figure 5.1: Scala IDE for Eclipse Running the Effects Plugin

The second mode of operation is used in the Scala interpreter, a classical read-eval-print loop, and in Scala worksheets in Eclipse and IntelliJ IDEA. These tools are typically used for exploring libraries and trying out ideas. Effect checking works in the same way as in the batch compiler; effects are checked whenever an expression is compiled and evaluated.

Finally, the resident mode of the Scala compiler is used within some IDEs to support advanced editing features such as code completion, hyperlinking, displaying the type of an expression and, error highlighting as you type. When the compiler plugin for effects is enabled in the resident mode of the compiler, effect errors are reported to the programmer immediately when writing a statement which has a larger effect than allowed by the signature of the method. This feedback is displayed sooner than the errors reported by the batch compiler when the project is built. Figure 5.1 shows a screenshot of the Scala IDE for Eclipse with the effects plugin activated in the presentation compiler. There are a number of other tools that use the resident mode of the Scala compiler for interactive editing, for instance the Netbeans IDE<sup>1</sup> and the Ensime plugin for Emacs<sup>2</sup> or Sublime Text<sup>3</sup>. Effect checking can be enabled in these tools in the same way as in the Scala IDE for Eclipse.

The current implementation supports effect checking for three effect domains: IO, exceptions and purity with respect to state changes. The following section shows how programmers can annotate and check effects from these domains in their programs.

<sup>1</sup><http://wiki.netbeans.org/Scala>

<sup>2</sup><https://github.com/aemoncannon/ensime>

<sup>3</sup><https://github.com/sublimescala/sublime-ensime>



### 5.1.1 Annotating Effects in Multiple Domains

The effects of a method are specified in Scala as a sequence of annotations on the method's return type. All effect annotations are defined as ordinary annotation classes and no changes to the language syntax are required to implement effect checking.

To annotate multiple effect domains, the annotations of each domain are placed on the return type of a method in arbitrary order. For example, the following method has no IO effect and might throw an exception of type `E`:

```
def m(x: T): Unit @noIo @throws[E] = ...
```

Effects are inferred when the result type of a method is inferred. Methods that have an explicit result type but no effect annotations are impure, they have the largest possible effect in every effect domain. Our experience suggests that this simple annotation scheme works well in practice. However, the ability to infer effects of methods with an explicit return type could be easily added, for example by introducing an `@infer` annotation.

The effects in a method signature can be narrowed down by adding explicit effect annotations, as explained in Section 2.4. In all effect domains which are *not* annotated, the system assumes the method to be impure. For example, the method `m` above has the unknown effect `@mod(any)` in the domain of state modifications.

The `@pure` annotation marks a method as pure across all domains, it changes the default effect for non-annotated domains to the bottom effect. By adding specific annotations to the return type the programmer can allow some effects while the method remains pure in the other domains. For example, a method that might throw an exception but is otherwise pure is annotated as follows:

```
def m(x: Int, y: Int): Int @pure @throws[E] =
  if (y == 0) throw new E else x / y
```

Section 3.3 shows that the `@pure` annotation is also used for annotating effect-polymorphism. Specifically, relative effects are expressed as argument expressions of the purity annotation, for example `@pure(a.m)` denotes the effect of method `m` of parameter `a`. Given that effect-polymorphic methods always have the `@pure` annotation, their default effect for non-annotated domains is the bottom effect. This decision is consistent with the formal type system with relative effects that is presented in Section 3.2.

#### Annotations for IO, Exceptions and Purity

In the IO domain, there are only two effect annotations: `@io` for methods that have an IO effect and `@noIo` for pure methods.

## Chapter 5. Effect Checking in Scala

---

The exceptions that a method may throw are described with the `@throws[E]` annotation. This annotation specifies that exceptions of any type conforming to `E` might be thrown: for example, the effect annotation `@throws[Exception]` allows throwing exceptions of type `IOException`. Methods that might throw multiple types of exceptions can either have multiple `@throws` annotations, or the effect types can be combined with the predefined type operator “|”, i.e., `@throws[E1 | E2]`. Pure methods that do not throw any exceptions are annotated `@throws[Nothing]`.

The effect system for exceptions is equivalent to checked exceptions in the Java programming language. In Scala it is however not always possible to infer the effect of a `try` expression at compile-time, because the case statements guarding exception handlers can be arbitrary patterns, not just lists of types like in Java:

```
try {
  dubiousOperation()
} catch {
  case e if isFullMoon() => ()
}
```

In this example the exceptions of `dubiousOperation` are not always caught; the effect system assumes that the handler does not match and the exceptions are propagated.

Specifically, the system recognizes exception handling patterns of the form “case `e`: `E`” that catch one specific exception type, patterns that catch all exceptions (“case `e`”) and alternatives of these patterns, for instance “case `_`: `E1` | `_`: `E2`”. All other patterns, including patterns with a non-empty guard, do not match, which can lead to an over-approximation of the actual effect. In the cases where the type system fails to infer the exceptions precisely the programmer can override the effect system using an effect cast as explained later in this chapter.

The effect annotations for the purity domain are explained in Chapter 4: Section 4.2 introduces purity, ownership and result locality annotations, and Section 4.4.1 defines effect annotations for assignments to local variables.

### 5.1.2 Ascriptions and Effect Casts

As in ordinary type checking, effect mismatch errors are reported with respect to the location in the source code where the mismatch occurs. Nevertheless it is sometimes helpful for programmers to be able to check or document the effect of a statement within a larger piece of code individually, which can be done with an annotation ascription:

```
def f(): Unit @pure @io = {
  (somePureMethod() : @pure)
  someIoMethod()
}
```

```
}

```

The ascription allows the programmer to ensure that the invocation of `somePureMethod` does not have any effects.

There are a number of situations in which the effect inferred by the type system is larger than the actual effect that might occur at run-time. For example, the method `get` of class `Option` throws a `NoSuchElementException` if the optional value is empty. If a program tests the optional value for emptiness before invoking `get` the exception cannot occur, but the effect system does not take control flow into account and includes the effect:

```
def f(o: Option[Int]): Int @pure = {
  if (o.isEmpty) 0
  else (o.get: @unchecked @pure)
}
```

In order to type check this definition the effect of method `get` needs to be eliminated with an effect cast. Syntactically, effect casts are annotation ascriptions with an additional `@unchecked` annotation.

In Java, exceptions like `NoSuchElementException` or `NullPointerException` that are caused by client code are *unchecked*: the effect system allows them to occur in pure contexts [Gosling et al., 2013, Section 11.1.1]. For practical reasons, adopting this mechanism might also be desirable in the effect system for Scala.

### 5.1.3 Annotating Constructors and Default Arguments

The type system with relative effects from Chapter 3 and the effect system for purity from Chapter 4 both use dependent types: effect annotations can depend on a method's parameters. In Scala, the scope of a parameter includes the successive parameter lists, the return type and the method body. The fact that some effect annotations need to refer to the method's parameters is one of the reasons why effects are part of the return type.

Unfortunately, this is not possible for constructors because the syntax of Scala does not allow the return type of a constructor to be specified explicitly. There are two possibilities for annotating the effect of a constructor. If the effect annotation does not require the constructor parameters to be in scope, it can be placed on the constructor definition, or on the class or object definition for the primary constructor:

```
class C {
  var x = 0

  @pure @mod(this) def this(i: Int) = {
    this()
    x = i
  }
}
```

```
    }  
  }  
  
  @pure object C {  
    def fromString(s: String) = new C(s.toInt)  
  }
```

Both effect annotations in the above example are optional and can be inferred by the effect system. Indeed, the effect of the primary constructor of class `C` is not annotated, but inferred to be `@mod(this)` since it initializes the field `x` of the constructed object. This effect annotation is however not valid outside the class `C`: the reference `this` would be invalid for top-level classes or point to the outer object for nested classes.

To overcome this issue, constructor effect annotations that refer to parameters can be placed on a type definition named `constructorEffect` within the constructor body (or the class template for primary constructors):

```
class C {  
  @pure @mod(this) private[this] type constructorEffect = Nothing  
  var x = 0  
}
```

The type chosen on the righthand side of the type definition is not relevant. The advantage of this solution is its compatibility with the current Scala syntax. If future Scala versions allow constructor type annotations, migrating existing code will be straightforward.

Another special case for annotating effects are default argument expressions. The Scala compiler translates default arguments to member methods of the enclosing class. Method invocations that use defaults invoke these *default getters*. The return type of a default getter is identical to the corresponding parameter type. To specify the effect of a default expression the programmer has to annotate the parameter type.

```
def m(i: Int @pure = 1) = 10 * i
```

In the current implementation, default getters are impure by default if there are no effect annotations on the parameter type, which implies that pure default argument expressions need to be annotated. This is consistent with ordinary methods which are impure if they have an explicit return type but no effect annotations.

### 5.1.4 Singleton Objects, Lazy Values and By-Name Parameters

The introduction to type-and-effect systems in Section 2.1 shows that function (or method) abstractions delay the effect of an expression to the point where the function is invoked. In the Scala language there are three additional constructs that can delay execution, and therefore

effects: constructors of singleton objects, initializers of lazy values and by-name parameters.

Singleton objects and lazy values are initialized on their first access. Because the type-and-effect system for Scala does not have advanced features like flow-sensitivity or type state, the initialization status of lazy objects is not known statically. The effect system includes the initializer effects for lazy values and singleton objects whenever such objects are accessed.

The effect of a singleton object can be either inferred or annotated as explained in the previous section. For lazy values the syntax is the same as for method definitions: the initializer effect can be annotated on the type of the value, and it is inferred if also the type is inferred.

A by-name parameter type “ $\Rightarrow T$ ” in Scala is essentially syntactic sugar for the nullary function type “ $() \Rightarrow T$ ”. When invoking a function with a by-name parameter, the Scala compiler creates a thunk from the corresponding argument expression. Within the body of the method, accesses to the by-name parameter are transformed to invocations of the `apply` method of the function.

In terms of side effects, each access to a by-name parameter has the unknown effect. However, functions with by-name parameters are typically effect-polymorphic, hence the effect for the parameter is computed at call site:

```
def twice[T](op: => T): T @pure(op) = { op; op }

def t: Int @pure = {
  var x = 0
  twice { x += 1 }
  x
}
```

Since `twice` is effect-polymorphic, the invocation in method `t` has the effect of assigning to the local variable `x`, which can be masked when the variable gets out of scope.

### 5.1.5 Effects Affect Typing and Subtyping

The primary goal of an effect system is to verify that the body of a method can only have side effects that are allowed by the method signature. To achieve this goal, the effect system needs to ensure that function values are not propagated in an unsound manner: for example, if a higher-order function only accepts pure functions as argument, then the type system needs to reject invocations that pass an impure function as argument. This is achieved by the subtyping rules.

Since effect annotations are part of method result types in Scala, function types that declare a specific effect are expressed as refinement types:

```
type PureFun = (Int => Int) { def apply(x: Int): Int @pure }
```

```
def appOne(f: PureFun): Int @pure = f(1)
```

When invoking the function `appOne`, the `apply` method of the argument object is checked to be `pure`:

```
object f1 extends Int => Int {
  def apply(i: Int): Int @pure @throws[E] = throw new E
}
appOne(f1) // produces a type mismatch error
```

The fact that effect annotations need to be taken into account when comparing two method types is an important reason for placing effects on the result type in Scala. Method types are characterized by the parameter symbols and types and by the result type. In subtyping, the result types are compared in covariant order, which is exactly what is required for effect annotations. Section 5.4 explains how the compiler plugin integrates into subtype checking in the Scala compiler.

When computing the subtyping relation between two types in a nominal type system, the algorithm usually does not need to compare individual methods of the two types, but instead just looks at the inheritance structure. For example, if a class `Sub` extends the superclass `Parent`, then `Sub <: Parent` holds by definition. To ensure soundness, the type system requires all method overrides to be *sound*, which implies that the type and effect of an overriding method is required to conform to the type and effect of the method in the supertype [Odersky, 2013, Section 5.1.4].

For this reason, the definition of method `i` in trait `B` in the following example is rejected:

```
trait A {
  def i: Int @pure
}
class B extends A {
  def i: Int @throws[E] = throw new E
}
```

Since the effect system for Scala is implemented as a compiler plugin and effect annotations have no semantics in the Scala language itself, effect checking can be seen as a pluggable type system as described by Bracha [2004]. If the compiler plugin is disabled, effect annotations are simply ignored. Enabling the plugin on the other hand can lead to additional error messages, i.e., the effect system can reject more programs, but it does not change the semantics of the compiled code.

Unfortunately, this statement is not entirely true, because the Scala programming language itself does *not* have a pluggable type system: there are multiple language features for which the semantics depends on the type system, for instance overloading resolution and implicit search. Since the effect system affects the subtyping rules, it can influence the outcome of an

implicit search and change the semantics of a program:

```
def doApply(implicit f: { def m(): Object @pure }) = m()

class A {
  implicit val vA: { def m(): String } = ...
}

class B extends A {
  implicit val vB: { def m(): Object @pure } = ...

  doApply
}
```

This example does not compile when the compiler plugin is disabled. The invocation of `doApply` triggers implicit search for an object of type “`{def m(): Object @pure}`”, and both values `vA` and `vB` match this type (annotations are ignored since the plugin is disabled). The Scala type system chooses the more specific of the two values according to the procedure for overloading resolution defined in [Odersky, 2013, Section 6.26.3]: the value `vA` obtains one point for having a more specific return type, but also `vB` gets one point because it is defined in a subclass. Compilation therefore fails with the error message “ambiguous implicit values”.

When the compiler plugin is enabled, compilation succeeds: the type of value `vA` no longer matches the required argument type because the implicit parameter requires a pure method `m`. There remains only one matching implicit value which is therefore inserted by the compiler as argument in the invocation of `doApply`.

It is possible to construct similar examples which *do* compile in plain Scala but fail to compile when the effects plugin is enabled. We discovered one such case when applying the effects plugin on the Scala standard library<sup>4</sup>.

## 5.2 Effect Checking in the Scala Collections Library

The main benchmark we used for evaluating the effect system is the Scala collections library. In this section, we highlight the most interesting aspects in terms of effect checking for the core of the immutable collections.

### 5.2.1 Option

The effect annotations for class `Option` in Figure 5.2 are straightforward. In the body of `getOrElse` the effect of invoking `get` is eliminated with an effect cast, since effect checking

---

<sup>4</sup>The issue is recorded at <https://github.com/lrytz/efftp/issues/1>

does not take control flow into account as explained in Section 5.1.2. Note that the example uses effect-inference for all constructors and method definitions in the subclasses `Some` and `None`.

```
trait Option[+A] {
  def isEmpty: Boolean @pure
  def get: A @pure @throws[NoSuchElementException]
  def getOrElse[B >: A](default: => B): B @pure(default) =
    if (isEmpty) default else (get: @unchecked @pure)
}
case class Some[+A](a: A) extends Option[A] {
  def isEmpty = false
  def get = a
}
case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

Figure 5.2: Type Option

### 5.2.2 Breaks

Unlike Java, the Scala language does not provide a `break` statement to break out of a `while` loop. There are situations in the collections library where breaks are desired, e.g., when searching for an element in a collection. The object `Breaks` in the Scala library defines operators that allow breaking out of loops by throwing and catching an exception:

```
object Breaks {
  private val breakException = new BreakControl
  def breakable(op: => Unit): Unit @unchecked @pure(op) =
    try { op }
    catch { case ex if (ex eq breakException) => () }
  def break(): Nothing @unchecked @pure = throw breakException
}
```

When invoking the `break` method within a “`breakable { ... }`” block, execution resumes after that block. In the definition presented above, both operators are pure and `breakable` is effect-polymorphic, i.e., it has the effect of its argument block.

To make the use of breaks safer, the effect system can enforce the invariant that invoking `break` is only valid in a `breakable` context. For this, the `break` operation would have an effect, for instance `@throws[BreakException]`, which is masked by `breakable`. As explained in Section 3.4, currently the effect system does not support annotations for effect masking, so the behavior of `breakable` has to be implemented as a specific effect typing rule in the compiler plugin.



### 5.2.3 Core Collection Classes

This section presents the core elements of the collections library and shows how effects in multiple domains are annotated and checked<sup>5</sup>. The basic design of the collections library is based on trait `TraversableLike` and uses builder objects to implement operations. It is introduced in Sections 3.3.3 and 4.4.4.

```

trait Builder[-Elem, +To] {
  def +=(elem: Elem): Unit @pure @mod(this)
  def result: To @pure
}

trait TraversableLike[+A, +Rep] { self: Rep =>
  def newBuilder: Builder[A, Rep] @pure @loc()
  def foreach[U](f: A => U): Unit @pure(f)
  def tail: Rep @pure @throws[NoSuchElementException] = {
    if (isEmpty) throw new NoSuchElementException("tail of empty collection")
    drop(1)
  }
  def takeWhile(p: A => Boolean): Rep @pure(p) = {
    val b = newBuilder
    breakable {
      for (x <- this) {
        if (!p(x)) break
        b += x
      }
    }
    b.result
  }
}

```

Figure 5.3: Base Trait `TraversableLike`

Figure 5.3 presents the `Builder` and `TraversableLike` traits with effect annotations for multiple domains. The implementation of method `takeWhile` makes use of effect-polymorphism, modifications of a local builder object and loop-breaking.

Figure 5.4 shows the most important classes in the parent hierarchy of class `List`. Trait `IterableLike` is extended by all collections that support iteration, and trait `SeqLike` defines the `apply` method for element access in indexed sequences.

The method `foreach` in trait `Iterator` uses an effect cast in line 5 to eliminate the exception effect `@throws[NoSuchElementException]` for the invocation of `next` - the exception cannot be raised because `hasNext` is true at this point.

<sup>5</sup>The full example is available on <https://github.com/lrytz/efftp/blob/master/tests/src/test/resources/scala/tools/nsc/effects/multi/Colls-files/colls.scala>

```
1 trait Iterator[+A] {
2   def hasNext: Boolean @pure
3   def next(): A @pure @mod(this) @throws[NoSuchElementException]
4   def foreach[U](f: A => U): Unit @pure(f) @mod(this) = {
5     while (hasNext) f(next(): @unchecked @pure @mod(this))
6   }
7 }
8
9 trait IterableLike[+A, +Rep] extends TraversableLike[A, Rep] { self: Rep =>
10  def iterator: Iterator[A] @pure @loc()
11  def foreach[U](f: A => U): Unit @pure(f) = iterator.foreach(f)
12 }
13
14 trait SeqLike[+A, +Rep <: SeqLike[A, Rep]] extends IterableLike[A, Rep] {
15  self: Rep =>
16
17  def apply(idx: Int): A @pure @throws[IndexOutOfBoundsException]
18  def iterator: Iterator[A] @pure @loc() = new Iterator[A] {
19    var these = self
20    def hasNext = !these.isEmpty
21    def next() =
22      if (hasNext) {
23        val result = these.head
24        these = these.tail
25        result
26      } else throw new NoSuchElementException("next on empty iterator")
27  }
28 }
29 sealed abstract class List[+A] extends SeqLike[A, List[A]] {
30  def apply(n: Int) = {
31    val rest = drop(n)
32    if (n < 0 || rest.isEmpty) throw new IndexOutOfBoundsException("'" + n)
33    rest.head
34  }
35 }
36 final case class ::[+A](override val head: A,
37                          override val tail: List[A]) extends List[A]
38 case object Nil extends List[Nothing]
```

Figure 5.4: Iterators, Sequences and Lists

For most method definitions that implement abstract methods from a parent type, the return type and effect annotations are omitted, e.g., the methods `hasNext` and `next` in line 20 or method `apply` in line 30. This does not mean that the effects of these methods are not verified. For methods without a return type, the effect system infers the effect of the method body and attaches it to its signature. As explained in Section 5.1.5, override checking ensures that the effect of a method's implementation in a subclass conforms to the annotated effect in the

parent.

In line 36 in class `::` the methods `head` and `tail` defined in the parent trait `TraversableLike` are overridden with stable values. These overrides are sound since getter methods do not have any side effects, so the effects in the subclass conform to the `@throws[NoSuchElementException]` effects in the parent trait.

As a general pattern, we can observe that effect annotations are required in interface declarations and for defining effect-polymorphic methods. In implementation classes the effects can often be inferred. Note that the entire example did not require any constructor effect annotations.

### 5.3 Implementing Effect Domains

The compiler plugin for effect checking is not only a tool provided to Scala programmers, but also an extensible framework which facilitates the implementation of new effect domains. In order to define a new effect domain, the framework has to be provided with a representation of the effect lattice, the corresponding source code annotations, method definitions which translate between the two effect representations and, if applicable, effect inference rules for specific language features or predefined methods. Effect domain definitions are lightweight: for example, the entire implementation of the exceptions domain is in one single source file of roughly 200 lines of code.

In the current design of the compiler plugin, the source code for each effect domain is part of the source code of the plugin. In order to add a new domain to the system, the compiler plugin has to be extended and recompiled. Even though it is possible to activate domains individually using command line arguments, enabling effect domains to be implemented *outside* the compiler plugin would make the system more flexible. We plan to address this restriction in our future work.

#### 5.3.1 Effect Lattice

Effect lattices are defined as implementations of trait `EffectLattice` which declares an abstract type `Effect`, the top and bottom elements, methods to compute joins and meets, and a method `lte` to compare effects.

In the case of exceptions presented in Figure 5.5, effects are represented as a list of exception types that might be thrown, wrapped in an object of type `Throws` for convenience. The fact that the compiler-internal representation of types, `global.Type`, is *path-dependent* makes the definition slightly more involved. The lattice for exceptions is defined with respect to an instance of the Scala compiler, represented by the abstract field `global`. When instantiating

```
abstract class ExceptionsLattice extends EffectLattice {
  val global: Global

  case class Throws(tps: List[Global.Type] = Nil) { ... }
  type Effect = Throws

  lazy val top: Effect = Throws(throwableType)

  def lte(a: Effect, b: Effect): Boolean = { ... }
  def join(a: Effect, b: Effect): Effect = { ... }

  def mask(orig: Effect, mask: Effect): Effect = { ... }
}
```

Figure 5.5: Effect Lattice for Exceptions

the effect lattice in the compiler plugin, the field is defined to be the current compiler, which makes the type `global.Type` compatible with type representations that originate in that specific compiler instance.

In addition to the mandatory lattice operators, the lattice for exceptions defines a method `mask` which subtracts the parameter effect `mask` from `orig` and is used to compute the effect of try expressions as explained in the following section.

### 5.3.2 Domain Definition

Effect domain definitions are implemented as subclasses of the abstract class `EffectDomain` which provides functionality shared across domains, like effect inference and effect checking for methods, or the implementation of relative effects. Each effect domain has to specify its lattice and conversion methods between elements of the lattice and effect annotations represented as `AnnotationInfo` instances in the compiler.

The effect domain for exceptions is presented in Figure 5.6. Its effect lattice is defined in line 2 as an instance of `ExceptionsLattice` with a refined type for the field `global`. This singleton type ensures that the `Global` instances in the effect lattice and in the `EffectDomain` are the same object, which makes the path-dependent type `global.Type` compatible across the two classes.

To implement custom effect inference rules for specific language elements, effect domain authors can override the method `computeEffectImpl` and `pattern match` on the abstract syntax tree. The main information stored in the effect context parameter `ctx` is the relative effect environment of the current expression and the expected effect which enables reporting effect mismatches at the precise location in the source file.

```

1 abstract class ExceptionsDomain extends EffectDomain {
2   lazy val lattice = new ExceptionsLattice {
3     val global: ExceptionsDomain.this.global.type =
4       ExceptionsDomain.this.global
5   }
6
7   def parseAnnotationInfos(annots: List[AnnotationInfo],
8     default: => Effect): Effect = { ... }
9   def toAnnotation(eff: Effect): List[AnnotationInfo] = { ... }
10
11  override def computeEffectImpl(tree: Tree, ctx: EffContext) = tree match {
12    case Throw(expr) =>
13      val exprEff = super.computeEffect(expr, ctx)
14      exprEff u Throws(expr.tpe)
15
16    case Try(body, catches, finalizer) =>
17      val (mask, maskIsPrecise, catchEff) = typesMatchingCases(catches, ctx)
18      val expMasked = if (maskIsPrecise) ctx.expected.map(_ u mask)
19                      else None
20      val bodyEff = super.computeEffect(body, ctx.copy(expected = expMasked))
21      val bodyEffMasked = lattice.mask(bodyEff, mask)
22      val finEff = super.computeEffect(finalizer, ctx)
23      bodyEffMasked u catchEff u finEff
24
25    case _ =>
26      super.computeEffectImpl(tree, ctx)
27  }
28 }

```

Figure 5.6: Effect Domain for Exceptions

In the domain of exceptions, custom inference rules are required for `throw` and `try` expressions. The effect of an expression “`throw expr`” consist of the effect of evaluating `expr`, computed using `super.computeEffect` in line 13, and the exception type of expression `expr`. The operation “`e1 u e2`” is an alias for `lattice.join(e1, e2)`.

To compute the effect of a `try` expression in line 17, the effect handlers are analyzed using the helper method `typesMatchingCases` which returns the handled exception types, a boolean indicating if the computed mask is precise and the effect of the handler expressions. As explained in Section 5.1.1, the effect mask of an exception handler cannot always be computed precisely in Scala.

To compute the effect of the body of the `try` expression, the expected effect is adjusted to allow those exceptions which are caught by a handler. For example, in the expression “`try expr catch {case _: E =>}`”, exceptions of type `E` are allowed to be thrown in expression `expr`, so `E` is added to the expected effect.

After computing the effect of the `try` body, the effect mask is applied to obtain the uncaught exceptions that might occur in the body. The overall effect of the `try` expression is the join of the masked body effect, the effect of the handlers and the effect of the finalizer.

### 5.4 Internals of the Compiler Plugin

The implementation of the effect system for Scala is built on top of the plugin infrastructure provided by the Scala compiler, which enables deep integration into the type checking process. This section explains the internals of the compiler plugin and discusses the principal aspects of its design. The alternative of implementing effect checking as a separate compilation phase after type checking is discussed in Section 5.4.5.

#### 5.4.1 Compiler Plugins for Scala

Compiler plugins for Scala are created by writing an implementation of the abstract class `Plugin` defined in the compiler API. The compiled code of the plugin is packed into a JAR archive together with an XML file which describes the entry point into the plugin. This JAR file is passed as a command-line argument to the Scala compiler which loads and instantiates the classes using reflection.

There are three possibilities for a compiler plugin to extend the Scala compiler:

1. Compiler plugins can register new phases which can be inserted at any stage of the compilation pipeline. Each compiler phase can define tree traversals or tree transformations, and it can register a type transformation which is automatically applied to the type of each symbol. The compiler plugin for effect checking does not define new compilation phases and is implemented using the other two extension mechanisms provided by the compiler.
2. Extensions to the Scala type system that make use of type annotations can be implemented in a compiler plugin by registering a so-called *annotation checker*. Annotation checkers are invoked by the Scala compiler on type operations such as subtype tests or least upper bound calculations whenever one of the involved types has some annotations. For example, to implement a type system extension which tracks integer signs, an annotation checker can define the subtype relation between positive integers “`Int @pos`” and integers of unknown sign `Int`, which prevents negative integers to be passed to methods that only accept positive numbers.
3. To implement a type system extension, adding an annotation checker alone is typically not sufficient. For example a the type system for tracking integer signs, type inference for integer literals needs to be adjusted to assign an annotated type based on the value: the

literal 7 should have type “Int @pos” instead of only Int. Extensions to various aspects of the type checking process can be written in compiler plugins by registering so called *analyzer plugins*, which are invoked by the compiler when a type is assigned to a symbol or when a tree is type checked.

In order to be able to explain the annotation checker and the analyzer plugin for effect checking in more detail, the following section gives an overview on the type checking process in the Scala compiler.

### 5.4.2 Naming and Typing in the Scala Compiler

Even though the “typer” phase in the Scala compiler is scheduled to execute *after* the “namer” phase, the two components are actually inter-dependent and invoke each other recursively. On a high level, the typer phase is a tree transformation which takes as input an untyped tree and produces a typed tree. Before type checking a block of code, for instance a method body or a class template, the typer invokes the namer which creates symbols for all definitions nested directly in that block, assigns types to those new symbols and enters them into the current scope.

The current scope of the type checker is stored as a context object shared between typer and namer. By entering the symbols into the scope, the namer makes symbols available to the type checker. The namer does not descend into nested definitions: after creating and entering the symbols for the current scope it returns to the typer, which will invoke the namer again when type checking the nested definitions.

Note that the abstract syntax trees passed to the namer are not typed: source-level types are typically represented as identifiers or selections. For example in a method with a parameter “x: Int”, the type Int is represented as a tree of the form `Ident("Int")`. In order to obtain the symbolic representation of that type, the tree is passed to the type checker which resolves the name "Int", creates a `Type` instance representing the type `scala.Int` and assigns that type instance to the `tpe` field of the resulting tree.

The type assigned to a freshly created symbol by the namer is always a *lazy type*, i.e., a type that computes itself the first time it is accessed. The way the actual type is computed upon completion depends on the kind of definition that the symbol represents, but it always invokes methods of the typer and type checks certain trees.

For instance, a method type consists of parameter symbols and a return type. The types of the parameter symbols are available as type trees in the abstract syntax tree of the method, so their types are obtained by invoking “`typer.typedType(paramTypeTree).tpe`”. If the method definition has an explicit return type, the same procedure is applied to obtain the result type of the method. For method definitions where the return type is inferred, computing the type of the method symbol triggers a full type check of the method body using `typer.computeType(body)`,

and the result type is extracted from the resulting tree.

Due to lazy completion of the types assigned to symbols, the type checking phase is not a sequential transformation of the abstract syntax tree. For example, type checking a method selection `a.m` triggers completion of the type of symbol `a`. If the definition of `a` has an inferred result type, computing its type requires type checking the body of its definition, which might again lead to type checking other parts of the program in the same manner.

In order to enable compiler plugins to interact with the type checking process outlined above, the Scala compiler invokes registered analyzer plugins before or after the execution of important operations and allows them to modify the involved trees, types or symbols. More concretely, analyzer plugins can be used to modify the expected type before type checking a tree, to modify the type inferred by the type checker for a tree, or to change the type assigned to a symbol upon completion. The following section explains how the compiler plugin for effect checking uses those hooks in its implementation.

### 5.4.3 Implementation of the Effects Plugin

In this section, we discuss in detail the annotation checker and the analyzer plugin that our compiler plugin registers in the Scala compiler.

#### The Annotation Checker

The annotation checker refines the subtyping relation in the presence of effect annotations. For example, the type of a pure function conforms to the type of a function with an `@io` effect, but not vice versa:

```
val f: (() => Int) { def apply(): Int @pure } = () => { println("f"); 7 }
```

The subtype test in this example compares the type of method `apply` in the refinement with the `apply` method of the function literal. The comparison of the result types “`Int @pure`” and “`Int @io`” invokes the annotation checker which rejects the subtype test.

Annotation checkers also allow compiler plugins to refine the least upper bound (lub) and the greatest lower bound (glb) of a list of types. Least upper bounds are used amongst others to compute the type of an `if` expression: its type is the lub of the types of the two branches. The annotation checker for effects defines the least upper bound of two effects as their join. Therefore, the type of an `if` expression that returns either a pure function or a function performing IO is a function type with the IO effect.

The Scala compiler uses subtype tests, and therefore annotation checkers, not only for checking conformance of assignments or method arguments to an expected type, but also for



override checking and for type inference. As explained in Section 5.1.5, override checking ensures that the result type of an overriding method is more specific than the result type of the method in the superclass. Since override checks use normal subtype tests, the annotation checker for effects ensures that an overriding method cannot have a larger effect than the method in the superclass.

When inferring the type of an anonymous class instantiation, the type checker uses subtype tests to decide whether or not to keep detailed type information for the definitions. The following example shows a Scala REPL session which illustrates refinement type inference:

```
scala> new Function0[String] { def apply() = "" }
res0: () => String = <function0>

scala> new Function0[Object] { def apply() = "" }
res1: () => Object{def apply(): String} = <function0>
```

In the first expression the method `apply` in the anonymous class body has the same type as the `apply` method in the parent type `Function0[String]`, therefore no information is lost by assigning the expression the parent type. In the second expression, the `apply` method in the anonymous class specializes the return type of the method in the parent type from `Object` to `String`. Therefore, the method type is kept in the form of a refinement.

The same mechanism is used for refinement type inference in case the anonymous class defines a more precise effect for one of its members than in the parent type. This is shown in the following example:

```
1 scala> abstract class C { def m: Int }
2 defined class C
3
4 scala> def test(c: C { def m: Int @pure }): Int @pure = c.m
5 test: (c: C{def m: Int @pure})Int @pure
6
7 scala> test(new C { def m = 1 })
8 res5: Int = 1
9
10 scala> test(new C { def m = {println("m"); 1} })
11 <console>:13: error: type mismatch;
12   found   : C
13   required: C{def m: Int @pure}
14         test(new C { def m = {println("m"); 1} })
15         ^
```

The first invocation of `test` in line 7 type checks because the compiler infers a refined type for the anonymous class which specifies method `m` to be pure. In line 10 on the other hand, method `m` is impure like in the parent type `C`, therefore the inferred type is `C` without a refinement, which does not conform to the parameter type of method `test`.

In Scala, a function literal such as “`(x: Int) => x + 1`” is in principle equivalent to an anonymous class instantiation “`new Function1[Int, Int]{def apply(x: Int) = x + 1}`”. However function literals are more concise and, depending on the expected type, allow parameter and return types to be omitted. For this reason, function literals are handled separately by the Scala compiler and the procedure of refinement type inference described in this section is not applied. To ensure that the type assigned to a function literal also precisely describes the function’s effect, the compiler plugin for effects refines type inference for function literals using the analyzer plugin described in the following section.

### The Analyzer Plugin

The analyzer plugin is used for multiple tasks in the process of effect checking: it infers effects for methods with an inferred result type, it checks effects for all other methods, it assigns refined types to function trees and it prevents effect annotations from flowing to undesired places and causing spurious errors.

As explained in Section 5.1.1, the effects of a method are inferred whenever the result type of the method is inferred. The method `pluginsTypeSig` of the analyzer plugin is invoked after the lazy type of a symbol is completed, but before the type is assigned to the symbol. For methods with an inferred result type, this method infers the effect of the method body and attaches the corresponding effect annotations to the result type computed by the compiler.

Effect verification for methods with an explicit result type is performed during type checking using the method `pluginsTyped` of the analyzer plugin, invoked after type checking each subtree of the program. After type checking a method definition tree with an explicit return type, the effects plugin computes the effect of the method body and issues an effect typing error if it does not conform to the effect allowed by the type signature.

One problem when implementing an effect system using type annotations is that effects and types do not propagate in the same manner. In a block of code or a in sequence of selections, the type is defined by the last expression, but the effects depend on all of the expressions in the code. This is illustrated by the following example:

```
class C { def m(): Int @pure = 1 }
def makeC: C @io = { println("In Factory"); new C }
def test = makeC.m()
```

Before the analyzer plugin is invoked, the Scala type checker infers the type of method `test` to be “`Int @pure`”, which is simply the return type of method `m`. This is however not a valid type for method `test`. The effect annotation is unsound because the invocation of `makeC` has an `@io` effect and is not pure. As explained above, the analyzer plugin does infer the correct effect and assign it to the symbol of method `test`, however the plugin additionally prevents effect annotations from propagating incorrectly in the first place. This is achieved by the

analyzer plugin in method `pluginsTyped` which is invoked after type checking each subtree of the program: after type checking a term tree, all effect annotations on the type inferred by the compiler are removed.

The problem of incorrect propagation of effect annotations also exists for the expected type used for type checking by the Scala compiler:

```
def n: Int @pure = 7
```

When type checking the body of method `n`, using the annotated result type “`Int @pure`” as expected type leads to a spurious type mismatch error because the type of the number literal, `Int`, does not conform to the expected type. To fix this issue, the effects plugin eliminates all effect annotations from the expected type before type checking a term. This is implemented in method `pluginsPt`, which allows analyzer plugins to adjust the expected type.

Finally, the analyzer plugin is also responsible for effect inference for function literals. After type checking a function literal, the method `pluginsTyped` computes the effect of the function body and assigns it to the function type inferred by the compiler in the form of a refinement:

```
scala> val f = () => 7
f: () => Int{def apply(): Int @noIo} = <function0>

scala> val g = () => { println("g"); 7 }
g: () => Int{def apply(): Int @io} = <function0>
```

In this REPL session where only the IO domain is enabled, the effects of both function literals `f` and `g` are inferred and represented in the type using a refinement.

### 5.4.4 Propagation of Type Annotations in the Scala Compiler

The previous section shows that the annotation checker is not only used to infer and verify effects of functions and methods, but it also eliminates effect annotations from terms and expected types in order to prevent them from propagating incorrectly. This raises the question if the propagation rules for type annotations in the Scala compiler could be adjusted to accommodate the needs of the the compiler plugin for effect checking.

According to the Scala language specification [Odersky, 2013], there are no special rules for propagation of annotated types, they are treated the same as any other type. In particular, this implies that annotated types are conserved by the type projection operation called “*type  $T$  in class  $C$  seen from some prefix type  $S$* ” described in Chapter 3.4 of the specification.

The notion of a type “seen from” a prefix is most prominently used to obtain the correct type of a member selection in the context of subclassing as illustrated in the following REPL session:

```
scala> class C[T] { def f(x: T): T = x }
```

```
scala> object 0 extends C[Int]

scala> 0.f(1)
res0: Int = 1

scala> 0.f("s")
<console>:12: error: type mismatch;
 found   : String("s")
 required: Int
       0.f("s")
         ^
```

To compute the type of method `0.f`, the compiler first obtains the parent class of the prefix type `0.type` which defines method `f`, i.e., class `C` in this example. When accessed through object `0`, the type of method `f` is defined by replacing references to the type variable `T` within the declared method type  $(x:T)T$  by “`T` in class `C` *seen from* the prefix type `0.type`”. Since `T` is the first type parameter of class `C`, and the prefix type `0.type` has parent type `C[Int]`, the compiler substitutes `Int` for the type parameter `T` and computes the method type  $(x: Int)Int$  for method `0.f`, which explains the types displayed in the above example.

Because annotated types are not different from other types according to the language specification, we would expect that the example is analogous when attaching an annotation to the return type in the declaration of method `f`. However, the implementation in the Scala compiler does not fulfill that expectation and eliminates the annotation when computing the type projection:

```
scala> class ann extends annotation.Annotation
scala> class C[T] { def f(x: T): T @ann = x }
scala> object 0 extends C[Int]

scala> 0.f(1)
res1: Int = 1
```

The return type of method `f` seen from the prefix type `0.type` is just `Int` instead of “`Int @ann`”, the type annotation is not propagated through the selection. Note that the annotation is *not* eliminated in cases where type propagation does not require computing a projection:

```
scala> { val x: Int @ann = 1; val y = x; y }
res2: Int @ann = 1
```

In this example the inferred type for the local value `y` is simply the type of `x`, which is “`Int @ann`”.

These examples show that the current implementation of the type system in the Scala compiler does not exactly follow the language specification<sup>6</sup>. Internally, the compiler represents types as immutable data types. Type transformations are implemented as subclasses of the abstract

---

<sup>6</sup>The observations relate to the current release of the Scala compiler at the time of writing, which is 2.10.2

TypeMap transformation which maps a type to a new type by applying a given transformation function to each element of the original type. By default, type transformations preserve type annotations, but the type transformation that implements the “seen from” projection drops type annotations deliberately.

There is a special case in the “seen from” transformation which *does* conserve type annotations, namely, if the annotation class is a subtype of the trait `TypeConstraint` defined in the Scala standard library. The following example illustrates that behavior:

```
scala> import annotation.{Annotation, TypeConstraint}
scala> class constraintAnn extends Annotation with TypeConstraint
scala> class C[T] { def f(x: T): T @constraintAnn = x }
scala> object O extends C[Int]

scala> O.f(1)
res0: Int @constraintAnn = 1
```

The “seen from” transformation conserves the `@constraintAnn` annotation in the result type of method `O.f` when replacing the type parameter `T` by `Int`.

### Effects Are Type Constraints

Section 5.4.3 shows that the Scala type checker propagates effect annotations like ordinary types, which is unsound with respect to the semantics of an effect system. The compiler plugin for effect checking eliminates the annotations inserted by the type checker later on and computes the correct effects for methods and functions.

The reason why effect annotations are propagated through type projections is that their definitions extend the `TypeConstraint` trait, whose functionality is explained above. One might think that a more systematic way to prevent unsound propagation of effect annotations would be to define effect annotations as ordinary annotations instead of type constraints. This would lead to two different problems that we explain below, from which the second one cannot be solved in a straightforward manner.

The first problem is that ordinary type annotations are still propagated incorrectly in some cases, even though type inference does not require computing a projection. In the following example the compiler plugin for effects is not enabled. It shows how annotations are propagated by the type checker:

```
scala> class pure extends annotation.Annotation

scala> {
  | def f: Int @pure = 1
  | def g = { println("running g"); f }
  | g
}
```

```
| }  
running g  
res0: Int @pure = 1
```

The Scala compiler infers the result type “Int @pure” for method `g`, which is not sound according to the semantics of the effect system and would need to be corrected by the compiler plugin. This shows that defining effect annotations as ordinary annotations instead of type constraints does not prevent unsound propagation of effect annotations in all cases.

The second problem is related to the fact that effect annotations are not only used for verifying the effects of methods, but they affect subtyping as explained in Section 5.1.5. In order to enable the effect system to compute the subtype relation correctly, it is essential that effect annotations are *not* eliminated during the computation of a type projection.

In the following example, the definition of the annotation class `@pure` does not extend the trait `TypeConstraint`. The “seen from” transformation eliminates the effect annotation when computing the type of the field selection `0.a` and the type of the local value `x` does not specify the effect of method `m`:

```
scala> class pure extends annotation.Annotation  
scala> trait A { def m: Int }  
scala> object 0 {  
  |   val a: A { def m: Int @pure } = new A { def m = 1 }  
  | }  
  
scala> val x = 0.a  
x: A{def m: Int} = anon1@22f4bf02
```

This example shows the propagation of annotations without any compiler plugins enabled. The type annotation is eliminated early by the type checker and a compiler plugin cannot influence this process. In the case of the compiler plugin for effect checking, this would lead to spurious type mismatch errors: assume a method `f` takes a parameter of type `A{def m: Int @pure}`. The invocation `f(x)` would be rejected by the effect system because method `f` requires an instance of `A` with a pure method `m`, while the type of `x` permits method `m` to have arbitrary effects.

For this reason, all effect annotation classes defined in the effect system’s implementation extend the trait `TypeConstraint`, which ensures that effect annotations are not eliminated by the type checker:

```
scala> import annotation.{Annotation, TypeConstraint}  
scala> class pure extends Annotation with TypeConstraint  
scala> trait A { def m: Int }  
scala> object 0 {  
  |   val a: A { def m: Int @pure } = new A { def m = 1 }  
  | }
```

```
scala> val x = 0.a
x: A{def m: Int @pure } = anon1@45485026
```

There are a few issues in the implementation of the effect system related to the fact that effect annotations propagate incorrectly in the type checker. For example, when the compiler plugin for effects is disabled, propagated effect annotations are never eliminated even though in some cases the resulting annotations are not correct, as explained earlier in this section.

In the future, we plan to specify precisely the propagation of type annotations in the type system and to allow compiler plugins to customize this process.

### 5.4.5 Implementing Effect Checking as a Separate Compilation Phase

The previous section discusses a number of difficulties in the implementation of the compiler plugin for effect checking. The problems are caused by the interaction between the ordinary type checking process in the Scala compiler and the requirements for effect checking. Given these issues, would it be possible and simpler to implement effect checking as a separate phase in the compilation process, instead of extending the type checking process? In one of our prototypes we tried to implement effect checking as a separate compilation phase. Based on the problems encountered while writing this prototype, we answer the above question negatively.

First, it is essential to integrate the effect checking process into subtype checking of the Scala type system. Subtyping ensures that functions with side effects are rejected in places where a pure function is expected. Because such a subtype test can appear in any assignment or method invocation, an effect checker that does *not* integrate with the ordinary subtyping process would need check each of these cases manually. Furthermore, subtyping is not only used for comparing types but also for inferring type refinements, as explained in Section 5.4.3. This implies that the compiler plugin requires an annotation checker either way.

A second issue is that effect checking is only possible if effect annotations in refinements are propagated through type projections, as explained in Section 5.4.4. Thus, effect annotation classes need to extend the trait `TypeConstraint` as explained in Section 5.4.4, even if effect checking happens in a later compilation phase. The compiler plugin also needs to handle the incorrect propagation of effect annotations.

Finally, a major issue we encountered is that that some parts of the program need to be type checked a second time in order to correctly compute effects. We illustrate this by the following example:

```
def m: Int @pure = {
  val f = (x: Int) => x + 1
  f.apply(x)
```

```
}
```

During type checking, the Scala compiler assigns type “`Int => Int`” to the value `f` and the selection `f.apply` is resolved to the method `apply` defined in trait `Function1`, whose signature permits arbitrary side effects. When computing the side effects of method `m`, the effect checking process assigns a more precise type to the function literal, namely the refined function type “`(Int => Int){def apply(x: Int): Int @pure}`”. The difficulty is that the selection `f.apply` has to be type checked again so that it resolves to the pure method `apply` defined in the refinement type. The process of eliminating types and symbols from syntax trees and type checking them again is brittle and difficult to achieve with the current implementation of the Scala compiler. By integrating effect checking into the type checking process, all selections are resolved correctly in the first place.

### 5.5 Future Work

In this section, we discuss practical issues that we plan to work on in the future in order to make the effect system easier to use in both existing and new projects.

#### 5.5.1 Effect Annotations for Existing Libraries

The most important problem not yet solved in our effect system for Scala is the interaction of effect-annotated code with existing libraries that are lacking effect information. To ensure soundness, the effect system assumes that methods without effect annotations might have arbitrary effects.

In our system, the only tool which allows programmers to specify the effect of a method defined in an external library is to introduce an effect cast at each call site, for instance:

```
def log(msg: String): Unit @io = {  
  java.lang.System.out.println(msg): @unchecked @io  
}
```

Because this approach is error-prone and leads to boilerplate code, it is not a solution that should be recommended to the users of the effect system.

We currently focus on two solutions for this. The first one is used in the Checker Framework by Dietl et al. [2011], a framework for implementing type system extensions for Java. To solve the same problem of interfacing existing libraries, they provide annotations for existing libraries in the form of “stub” files which contain annotated member declarations for existing classes. Stub files are ordinary source files, but without any implementations. They can be generated from binary libraries so that programmers only have to fill in the necessary effect annotations.



In addition to supporting stub files, the Checker Framework also defines a file format for external storage of annotations and provides a bytecode rewriting tool which can extract or insert external annotations from or into a binary library.

A second solution would be to provide effect specifications for existing libraries in the form of a program that uses the Scala reflection API. Specifically, the effect checking framework would allow users to register descriptor functions of type “`Symbol => Option[Effect]`” which take as argument a symbol representing the invoked method and optionally return the pre-defined effect of the corresponding method. These descriptor functions have the advantage of being potentially more concise than stub files. For example the function

```
sym => if (sym.owner == StringClass) Some(pure)
      else None
```

would define the default effect for all methods of class `String` to be `@pure`.

### 5.5.2 Effect Inference for Existing Libraries

To streamline or automate the process of declaring effect annotations for existing libraries we plan to work on an inter-procedural, whole-program effect inference tool. Ideally, such a tool would be able to compute effect summaries for methods in existing code and save them either as stub files or write them directly into the original source code.

The purity system `JPure` by Pearce [2011] consists not only of a type system for verifying purity but also of an inference algorithm for existing code. Effect inference was successfully applied to the Java standard library and overall, 40% of the analyzed methods could be inferred as pure.

In the case of Scala, global effect inference is more involved due to the prevalence of effect-polymorphism and higher-order functions. The work by Kneuss et al. [2013] on effect analysis for programs with callbacks uses heuristics to automatically infer effect-polymorphism: for example, for higher-order methods like `map` from the collections library, there is a large number of call sites with argument functions of varying types and effects. In such cases the inference algorithm *delays* the effect of the parameter function and treats the method as effect-polymorphic. Their inference algorithm has been implemented for Scala and we expect to be able to transform the results of its effect analysis into effect annotations which are compatible with our framework.

### 5.5.3 External Effect Domain Definitions

Section 5.3 shows that new effect domains can be added to the compiler plugin by adding source files containing the effect annotation classes and the effect domain definition, and

re-compiling the plugin with the additional domain. While the API for implementing effect domains is lightweight, requiring effect domain authors to compile and distribute custom builds of the compiler plugin is an overhead and forces the users of the effect system to choose among the available versions of the plugin.

In the future, we will work on allowing effect domains to be implemented and compiled outside the compiler plugin and enable the users of the effect system to include effect domains modularly.

### 5.6 Conclusion

The type-and-effect systems presented in this dissertation have been implemented in the form of a compiler plugin for Scala. Specifically, the implementation is a generic framework for effect checking based on the effect system in Chapter 2, it supports relative effect annotations as described in Chapter 3 and it includes the type system for purity from Chapter 4. The compiler plugin provides effect domain implementations for verifying IO effects, checked exceptions and purity. The system is designed as an extensible framework: adding new effect domains is straightforward and requires a small amount of code. The implementation did not require any changes to the Scala language and the compiler plugin works with the current release of the Scala compiler.

We applied the effect system to the core of the Scala collections library which mixes higher-order code and locally scoped side effects in various ways. We observed that effect annotations are typically required on interface definitions, but can be inferred in implementation classes. The required effect annotations are lightweight and easy to write and understand.

We explained the process of naming and type checking in the Scala compiler and showed how the compiler plugin for effect checking integrates into that process. We showed that the main difficulties are due to the fact that side effects propagate differently than types. For example, the effects of a block of code depend on the entire block, while the type only depends on the last expression.

## Chapter 6

# Conclusion

In this dissertation, we designed a practical type-and-effect system for Scala that uses intuitive and lightweight effect annotations to verify multiple kinds of side effects simultaneously. For each of the components of our system, we aimed to find a pragmatic balance between notational and conceptual overhead, simplicity and expressive power. We believe that such a compromise is crucial for the wide-spread adoption of effect systems and that the ideas presented in this thesis are a significant step towards that goal.

In recent years, there is a clear trend in mainstream programming languages towards the integration of object-oriented and functional programming. While Scala is a pioneer in this domain, other languages such as C# 3.0, C++ 11 and Java 8 followed by adding support for lambda expressions. The availability of lambda expressions naturally leads to higher-order functions, in which the side effects of a function usually depend on the effects of its argument. Therefore, support for effect-polymorphic functions is essential for the expressiveness of an effect system.

The traditional method to express effect-polymorphism is by using effect type parametrization, e.g., the effect system for checked exceptions in Java. In Chapters 2 and 3, we identified several issues related to the expressiveness and verbosity of checked exceptions in Java and we showed that a lightweight syntax for expressing effect-polymorphism is crucial to solve these issues. The verbosity of effect-polymorphism in Java leads to practical limitations. In Java 8, the interface `Iterable` defines a method `forEach` that applies its parameter function to all elements of a collection. To simplify the type signature, method `forEach` is not effect-polymorphic and only accepts argument functions that do not throw any exceptions [Oracle, 2013b].

A type-and-effect system is the most useful if it can verify multiple kinds of side effects. For example, dead code elimination can be applied only if an expression does not perform IO, does not modify any state and cannot throw an exception. In Chapter 2, we introduced a generic type-and-effect system where multiple effect domains can be integrated modularly. By embedding a lightweight syntax to express effect-polymorphic functions, we showed that

## Chapter 6. Conclusion

---

effect-polymorphism can be expressed independently of any specific effect domains. We proved that the effects computed by the generic system are sound. When adding one or multiple effect domains, showing soundness only requires to prove two lemmas for each effect domain, but it does not require a new inductive proof.

In object-oriented languages, every object carries a potentially large number of member functions. A method that takes an object as argument can be seen as taking a large number of functions as argument, just like a higher-order function. In Chapter 3, we introduced *relative effect annotations*, an intuitive and lightweight annotation scheme for effect-polymorphism that is compatible with both object-oriented and functional languages. We studied the properties and limitations of relative effect annotations using a lambda calculus with dependent types. Using the implementation of relative effect annotations for Scala, we showed that they can express common patterns involving higher-order code such as those found in the Scala collections library.

In Chapter 4 we presented a type-and-effect system for verifying purity of methods which do not modify program state that existed before their execution. State modification is one of the most widely used side effects, but it is difficult to track because of aliasing. The purity system identifies state that is freshly allocated within a method and masks modification effects on that fresh state. The system does not make an attempt at tracking or controlling aliasing in a program globally. This restriction allows building an effect system with effect annotations that are easy to understand and lightweight enough for programmers to use. We implemented the effect system for purity as an effect domain of the generic framework for effect checking in Scala and showed that it can express purity of common patterns, e.g., using an iterator or copying a collection using a mutable builder.

In Chapter 5 we discussed the implementation of the effect system for Scala, which is in the form of a compiler plugin. Our implementation includes the effect domains for IO, exceptions and state modification effects. Because implementing new effect domains is straightforward and requires a small amount of code, the framework can be used to verify various properties that can be modeled as effects. For example, effects can track blocking operations in Scala's futures library (cf. Section 2.6.2) or in an UI library [Gordon et al., 2013]. We applied the effect system to a core of the Scala collections library and observed that the annotation overhead is manageable and that the required effect annotations are easy to write and understand. We also discussed the implementation details of the compiler plugin and explained how it integrates into the type checking process of the Scala compiler.

Overall, we believe the work in this thesis shows that a carefully designed effect system is practical and beneficial and can be incorporated into mainstream programming.

# Appendix A

## Soundness Proof for LPE

This chapter presents the proofs for the preservation and soundness theorems for LPE that are stated in Section 2.8.

### A.1 Lemmas

This section introduces two additional lemmas used later in the preservation and soundness proofs.

#### A.1.1 Canonical Forms

The canonical forms lemma states that if a value has a function type, then it can only be a function.

**Lemma A.1.** Canonical forms.

1. If  $\Gamma; f \vdash v : T! \perp$  and  $T <: T_1 \xrightarrow{e} T_2$ , then  $v = (x : T'_1) \Rightarrow t$ .
2. If  $\Gamma; f \vdash v : T! \perp$  and  $T <: T_1 \xrightarrow{e} T_2$ , then  $v = (x : T'_1) \rightarrow t$ .

*Proof.* In the first case, the type  $T$  is a subtype of a monomorphic function type. The subtyping rule is only defined within function types of the same kind, therefore  $T$  cannot be a polymorphic function type.

There are only two kinds of values in the language: monomorphic and effect-polymorphic function abstractions. Since the type of  $v$  cannot be a polymorphic function type, this restricts the value  $v$  to be a monomorphic abstraction.

The argument for the second case is equivalent.  $\square$

### A.1.2 Value Typing Environment

Lemma A.2 states that the polymorphism context  $f$  of a typing environment  $\Gamma; f$  is not relevant when type checking values, parameters or an error term.

**Lemma A.2.** Environment for type checking values.

1. If  $\Gamma; f \vdash v : T! \perp$ , then  $\Gamma; f' \vdash v : T! \perp$  for an arbitrary  $f'$ .
2. If  $\Gamma; f \vdash x : T! \perp$  for a parameter  $x \in \Gamma$ , then  $\Gamma; f' \vdash x : T! \perp$  for an arbitrary  $f'$ .
3. If  $\Gamma; f \vdash \text{throw}(p) : \text{Nothing! } e$ , then  $\Gamma; f' \vdash \text{throw}(p) : \text{Nothing! } e$  for an arbitrary  $f'$ .

*Proof.* There are two kinds of values: monomorphic and polymorphic function abstractions. Both of the corresponding typing rules, T-ABS-MONO and T-ABS-POLY, do not make use of the effect-polymorphism environment  $f$  in any way, changing it therefore does not change the inferred type  $T$ . For parameters and error terms, the conclusion also follows immediately from the corresponding typing rules.  $\square$

**Monotonicity and Consistency** The proofs for the monotonicity and consistency lemmas (2.1 and 2.2) are given in Chapter 2.

### A.1.3 Substitution Lemmas

This section presents the proofs for Lemma 2.3 (Preservation under substitution for monomorphic abstractions) and Lemma 2.4 (Preservation under substitution for polymorphic abstractions).

*Proof (Lemma 2.3).* The preconditions of the lemma are:

- $\Gamma, x : T_1; f \vdash t : T! e_l$  with  $f \neq x$
- $\Gamma; g \vdash v : T_2! \perp$
- $T_2 <: T_1$

Proof of  $\Gamma; f \vdash [v/x]t : T'! e'_l$  with  $T' <: T$  and  $e'_l \sqsubseteq e_l$  by induction on the typing derivations for term  $t$ .

♪ Case T-PARAM: We have  $t = z$  and  $z : T \in \Gamma, x : T_1$ . There are two sub-cases:

- $z = x$ , then  $[v/x]z = v$  and  $T_1 = T$ . Since we have  $\Gamma; g \vdash v : T_2 ! \perp$ , by Lemma A.2, we obtain  $\Gamma; f \vdash v : T_2 ! \perp$ . The requirements  $T_2 <: T$  and  $\perp \sqsubseteq e_l$  are immediate.
- $z \neq x$ , then  $[v/x]z = z$ . We have  $\Gamma, x : T_1; f \vdash z : T ! e_l$ . Since we know that  $x$  does not appear freely,  $x \notin FV(z)$ , we can remove its binding from the typing environment and obtain the result.

♪ Case T-ABS-MONO: We have  $t = (y : T_a) \Rightarrow t_1$ . From the typing rule we get  $T = T_a \xRightarrow{e} T_b$  and  $\Gamma, x : T_1, y : T_a; \epsilon \vdash t_1 : T_b ! e$ . We can assume that  $x \neq y$  and  $y \notin FV(v)$  by applying  $\alpha$ -renaming if necessary.

- $\Gamma, y : T_a, x : T_1; \epsilon \vdash t_1 : T_b ! e$       permutation of the environment (1)
- $\Gamma, y : T_a; g \vdash v : T_2 ! \perp$       precondition, environment extended by  $y$  (2)
- $\Gamma, y : T_a; \epsilon \vdash [v/x]t_1 : T'_b ! e'$       with  $T'_b <: T_b \wedge e' \sqsubseteq e$ , by 1,2, induction (3)

Since  $[v/x]t = (y : T_a) \Rightarrow [v/x]t_1$  we can apply T-ABS-MONO using (3) and obtain

$$\Gamma; f \vdash [v/x]t : T_a \xRightarrow{e'} T'_b ! \perp$$

Verifying  $T_a \xRightarrow{e'} T'_b <: T$  and  $\perp \sqsubseteq e_l$  is straightforward.

♪ Case T-APP-MONO:  $t = t_1 t_2$ . From the typing rule we have:

- $\Gamma, x : T_1; f \vdash t_1 : T_a \xRightarrow{e} T ! e_1$
- $\Gamma, x : T_1; f \vdash t_2 : T_b ! e_2$
- $T_b <: T_a$  and  $e_l = \text{eff}(\text{APP}, e_1, e_2, e)$

We apply the induction hypothesis to the subterms  $t_1$  and  $t_2$  to obtain

- $\Gamma; f \vdash [v/x]t_1 : T_c ! e'_1$  with  $T_c <: T_a \xRightarrow{e} T$  and  $e'_1 \sqsubseteq e_1$       (1)
- $\Gamma; f \vdash [v/x]t_2 : T_d ! e'_2$  with  $T_d <: T_b$  and  $e'_2 \sqsubseteq e_2$       (2)

There are two subtyping rules that match  $T_c <: T_a \xRightarrow{e} T$ :

- case S-NOTHING, then  $T_c = \text{Nothing}$ . By applying T-APP-E with (1) and (2) we obtain  $\Gamma; f \vdash [v/x]t : \text{Nothing} ! e'_l$  where  $e'_l = \text{eff}(\text{APP}, e'_1, e'_2, \perp)$ . We have  $\text{Nothing} <: T$  by S-NOTHING.
- case S-FUN-MONO,  $T_c = T'_a \xRightarrow{e'} T'$ . By S-TRANS we obtain  $T_d <: T'_a$ . Applying T-APP-MONO yields  $\Gamma; f \vdash [v/x]t : T' ! e'_l$  where  $e'_l = \text{eff}(\text{APP}, e'_1, e'_2, e')$ . The result  $T' <: T$  is immediate.

In both cases we have  $\text{eff}(\text{APP}, e'_1, e'_2, e') \sqsubseteq \text{eff}(\text{APP}, e_1, e_2, e)$  for the resulting effect by monotonicity of  $\text{eff}$ .

♪ Case T-ABS-POLY: similar to the case T-ABS-MONO.

♪ Case T-APP-PARAM:  $t = f t_2$ . From the typing rule we have

- $f : T_a \xRightarrow{e} T \in \Gamma, x : T_1$ , and  $f : T_a \xRightarrow{e} T \in \Gamma$  since  $f \neq x$
- $\Gamma, x : T_1; f \vdash t_2 : T_b ! e_2$ , and  $T_b <: T_a$
- $e_l = \text{eff}(\text{APP}, \perp, e_2, \perp)$

By induction hypothesis  $\Gamma; f \vdash [v/x]t_2 : T'_b ! e'_2$  with  $T'_b <: T_b$ ,  $e'_2 \sqsubseteq e_2$ . Since  $f \neq x$  we

## Appendix A. Soundness Proof for LPE

---

have  $[\nu/x]f = f$ . Therefore applying T-APP-PARAM yields

$$\Gamma; f \vdash [\nu/x]t : T ! e'_1 \text{ with } e'_1 = \text{eff}(\text{APP}, \perp, e'_2, \perp)$$

By monotonicity of  $\text{eff}$  we obtain  $e'_1 \sqsubseteq e_1$ .

♪ Case T-APP-POLY: similar to the case T-APP-MONO. An additional lemma is required: if  $T' <: T$  then  $\text{latent}(T') \sqsubseteq \text{latent}(T)$ . The proof is straightforward.

♪ Case T-APP-E: similar to T-APP-MONO. Note that  $T <: \text{Nothing}$  implies  $T = \text{Nothing}$ , which is used in the proof.

♪ Case T-THROW: straightforward.

♪ Case T-TRY:  $t = \text{try } t_1 \text{ catch}(\overline{p}) t_2$ . From the typing rule we have:

$$\Gamma, x : T_1; f \vdash t_1 : T_a ! e_1 \text{ and } T_a <: T$$

$$\Gamma, x : T_1; f \vdash t_2 : T_b ! e_2 \text{ and } T_b <: T$$

$$e_1 = \text{eff}(\text{CATCH}(\overline{p}), \text{eff}(\text{TRY}, e_1), e_2)$$

By applying the induction hypothesis on  $t_1$  and  $t_2$  we obtain:

$$\Gamma; f \vdash [\nu/x]t_1 : T'_a ! e'_1 \text{ with } T'_a <: T_a \text{ and } e'_1 \sqsubseteq e_1$$

$$\Gamma; f \vdash [\nu/x]t_2 : T'_b ! e'_2 \text{ with } T'_b <: T_b \text{ and } e'_2 \sqsubseteq e_2$$

Applying T-TRY we get:

$$\Gamma; f \vdash [\nu/x]t : T ! e'_1 \text{ with } e'_1 = \text{eff}(\text{CATCH}(\overline{p}), \text{eff}(\text{TRY}, e'_1), e'_2)$$

By monotonicity of  $\text{eff}$  we obtain  $e'_1 \sqsubseteq e_1$ .

□

*Proof (Lemma 2.4).* The preconditions of the lemma are:

- $\Gamma, x : T_1; x \vdash t : T ! e_1$
- $\Gamma; g \vdash \nu : T_2 ! \perp$
- $T_2 <: T_1$

Proof of  $\Gamma; \epsilon \vdash [\nu/x]t : T' ! e'_1$  with  $T' <: T$  and  $e'_1 \sqsubseteq e_1 \sqcup \text{latent}(T_2)$  by induction on the typing derivations for term  $t$ .

♪ Case T-PARAM: Similar to case T-PARAM in the proof of Lemma 2.3.

♪ Case T-ABS-MONO:  $t = (y : T_a) \Rightarrow t_1$ . From the typing rule we get:

$$\Gamma, x : T_1, y : T_a; \epsilon \vdash t_1 : T_b ! e$$

$$T = T_a \stackrel{e}{\Rightarrow} T_b$$

We can assume that  $x \neq y$  and  $y \notin FV(v)$ .



$\Gamma, y : T_a, x : T_1; \epsilon \vdash t_1 : T_b ! e$	permutation of the environment
$\Gamma, y : T_a; g \vdash v : T_2 ! \perp$	precondition, $y$ added to environment
$\Gamma, y : T_a; \epsilon \vdash [v/x]t_1 : T'_b ! e'$	with $T'_b <: T_b \wedge e' \sqsubseteq e$ , by Lemma 2.3
$\Gamma; \epsilon \vdash [v/x]t : T_a \xrightarrow{e'} T'_b ! \perp$	by T-ABS-MONO

Verifying  $T_a \xrightarrow{e'} T'_b <: T$  and  $\perp \sqsubseteq e_l$  is straightforward.

♪ Case T-APP-MONO:  $t = t_1 t_2$ . From the typing rule we have:

$\Gamma, x : T_1; x \vdash t_1 : T_a \xrightarrow{e} T ! e_1$
$\Gamma, x : T_1; x \vdash t_2 : T_b ! e_2$
$T_b <: T_a$ and $e_l = \text{eff}(\text{APP}, e_1, e_2, e)$

We apply the induction hypothesis to the subterms  $t_1$  and  $t_2$  to obtain

$$\Gamma; \epsilon \vdash [v/x]t_1 : T_c ! e'_1 \text{ with } T_c <: T_a \xrightarrow{e} T \text{ and } e'_1 \sqsubseteq e_1 \sqcup \text{latent}(T_2) \quad (1)$$

$$\Gamma; \epsilon \vdash [v/x]t_2 : T_d ! e'_2 \text{ with } T_d <: T_b \text{ and } e'_2 \sqsubseteq e_2 \sqcup \text{latent}(T_2) \quad (2)$$

There are two subtyping rules that match  $T_c <: T_a \xrightarrow{e} T$ :

- case S-NOTHING, then  $T_c = \text{Nothing}$ . By applying T-APP-E with (1) and (2) we obtain  $\Gamma; \epsilon \vdash [v/x]t : \text{Nothing} ! e'_l$  where  $e'_l = \text{eff}(\text{APP}, e'_1, e'_2, \perp)$ . We have  $\text{Nothing} <: T$  by S-NOTHING.
- case S-FUN-MONO, then  $T_c = T'_a \xrightarrow{e'} T'$ . By S-TRANS we obtain  $T_d <: T'_a$ . Applying T-APP-MONO yields  $\Gamma; \epsilon \vdash [v/x]t : T' ! e'_l$  where  $e'_l = \text{eff}(\text{APP}, e'_1, e'_2, e')$ . The result  $T' <: T$  is immediate.

We need to verify  $\text{eff}(\text{APP}, e'_1, e'_2, e') \sqsubseteq \text{eff}(\text{APP}, e_1, e_2, e) \sqcup \text{latent}(T_2)$ . By (1), (2) and monotonicity of  $\text{eff}$  we obtain:

$$\text{eff}(\text{APP}, e'_1, e'_2, e') \sqsubseteq \text{eff}(\text{APP}, e_1 \sqcup \text{latent}(T_2), e_2 \sqcup \text{latent}(T_2), e)$$

Using the second property of the monotonicity Lemma 2.1 we conclude:

$$\dots \sqsubseteq \text{eff}(\text{APP}, e_1, e_2, e) \sqcup \text{latent}(T_2)$$

♪ Case T-ABS-POLY: similar to the case T-ABS-MONO.

♪ Case T-APP-PARAM:  $t = x t_2$ . From the typing rule:

$x : T_a \xrightarrow{e} T \in \Gamma, x : T_1$ , therefore $T_1 = T_a \xrightarrow{e} T$
$\Gamma, x : T_1; x \vdash t_2 : T_b ! e_2$ , and $T_b <: T_a$
$e_l = \text{eff}(\text{APP}, \perp, e_2, \perp)$

By applying the induction hypothesis on  $t_2$  we have:

$$\Gamma; \epsilon \vdash [v/x]t_2 : T'_b ! e'_2 \text{ with } T'_b <: T_b \wedge e'_2 \sqsubseteq e_2 \sqcup \text{latent}(T_2) \quad (1)$$

$$\Gamma; \epsilon \vdash v : T_2 ! \perp \quad \text{by Lemma A.2 (2)}$$

We have  $[v/x]t = v [v/x]t_2$ . Since  $T_2 <: T_1 \wedge T_1 = T_a \xrightarrow{e} T$ , there are two matching subtyping rules:

- case S-NOTHING, then  $T_2 = \text{Nothing}$ . Applying T-APP-E with (1) and (2) yields  $\Gamma; \epsilon \vdash [v/x]t : \text{Nothing} ! e'_l$  where  $e'_l = \text{eff}(\text{APP}, \perp, e'_2, \perp)$ . We have  $\text{Nothing} <: T$  by S-NOTHING.

## Appendix A. Soundness Proof for LPE

---

- case S-FUN-MONO,  $T_2 = T'_a \stackrel{e'}{\Rightarrow} T'$ . Using S-TRANS we obtain  $T'_b <: T'_a$ . Applying T-APP-MONO yields  $\Gamma; \epsilon \vdash [v/x]t : T' ! e'_1$  where  $e'_1 = \text{eff}(\text{APP}, \perp, e'_2, e')$ . The result  $T' <: T$  is immediate.

We need to verify  $\text{eff}(\text{APP}, \perp, e'_2, e') \sqsubseteq \text{eff}(\text{APP}, \perp, e_2, \perp) \sqcup \text{latent}(T_2)$ . Using (1) and the monotonicity of  $\text{eff}$  we obtain:

$$\text{eff}(\text{APP}, \perp, e'_2, e') \sqsubseteq \text{eff}(\text{APP}, \perp, e_2 \sqcup \text{latent}(T_2), e')$$

Since  $e' \sqsubseteq \perp \sqcup e'$ , monotonicity gives:

$$\dots \sqsubseteq \text{eff}(\text{APP}, \perp, e_2 \sqcup \text{latent}(T_2), \perp \sqcup e')$$

Using the second property of the monotonicity Lemma 2.1:

$$\dots \sqsubseteq \text{eff}(\text{APP}, \perp, e_2, \perp) \sqcup e' \sqcup \text{latent}(T_2)$$

And finally, since  $e' = \text{latent}(T_2)$ , we obtain:

$$\dots \sqsubseteq \text{eff}(\text{APP}, \perp, e_2, \perp) \sqcup \text{latent}(T_2).$$

♪ Case T-APP-POLY: similar to the case T-APP-MONO.

♪ Case T-APP-E: similar to T-APP-MONO.

♪ Case T-THROW: straightforward.

♪ Case T-TRY:  $t = \text{try } t_1 \text{ catch}(\overline{p}) t_2$ . By the typing rules:

$$\Gamma, x : T_1; x \vdash t_1 : T_a ! e_1 \text{ and } T_a <: T$$

$$\Gamma, x : T_1; x \vdash t_2 : T_b ! e_2 \text{ and } T_b <: T$$

$$e_1 = \text{eff}(\text{CATCH}(\overline{p}), \text{eff}(\text{TRY}, e_1), e_2)$$

By applying the induction hypothesis to  $t_1$  and  $t_2$ :

$$\Gamma; \epsilon \vdash [v/x]t_1 : T'_a ! e'_1 \text{ with } T'_a <: T_a \text{ and } e'_1 \sqsubseteq e_1 \sqcup \text{latent}(T_2)$$

$$\Gamma; \epsilon \vdash [v/x]t_2 : T'_b ! e'_2 \text{ with } T'_b <: T_b \text{ and } e'_2 \sqsubseteq e_2 \sqcup \text{latent}(T_2)$$

Applying T-TRY, we obtain  $\Gamma; \epsilon \vdash [v/x]t : T' ! e'_1$  with  $e'_1 = \text{eff}(\text{CATCH}(\overline{p}), \text{eff}(\text{TRY}, e'_1), e'_2)$ .

By the monotonicity Lemma 2.1 we obtain  $e'_1 \sqsubseteq e_1$ .

□

## A.2 Soundness Theorems

### A.2.1 Preservation

*Proof (Theorem 2.1).* The preconditions of the preservation theorem are:

- $\Gamma, f \vdash t : T ! e$
- $t \downarrow \langle r, S \rangle$

Proof of  $\Gamma, f \vdash r : T' ! e'$  with  $T' <: T$  by induction on the evaluation rules for term  $t$ .

♪ Case E-APP-E1:  $t = t_1 t_2$ . We have

$$t_1 \downarrow \langle \text{throw}(p), S_1 \rangle \wedge t_2 \downarrow \langle \text{throw}(p), S \rangle \quad \text{where } S = \text{dynEff}(S_1, \emptyset, \emptyset)$$

By typing rule T-THROW, we obtain  $\Gamma; f \vdash r : \text{Nothing} ! \text{eff}(\text{THROW}(p))$ . The result  $\text{Nothing} <: T$  is immediate by S-NOTHING.

♪ Case E-APP-E2: similar.

♪ Case E-APP:  $t = t_1 t_2$ . We have:

$$t_1 \downarrow \langle (x : T_1) \mapsto t_r, S_1 \rangle$$

$$t_2 \downarrow \langle \nu_2, S_2 \rangle$$

$$[\nu_2/x]t_r \downarrow \langle r, S_1 \rangle$$

We distinguish the following sub-cases for the various typing rules for application expressions:

– case T-APP-E. We have  $\Gamma; f \vdash t_1 : \text{Nothing} ! e_1$ . By applying the induction hypothesis to  $t_1$  we obtain  $\Gamma; f \vdash (x : T_1) \mapsto t_r : T'_1 ! e'_1$  such that  $T'_1 <: \text{Nothing}$ . Since the type of a function abstraction cannot be *Nothing*, this case is impossible.

– case T-APP-PARAM. We have  $t_1 = p$  for some parameter  $p$ . This is impossible, since we have  $t_1 \downarrow \langle (x : T_1) \mapsto t_r, S_1 \rangle$ , but there is no evaluation rule for parameters.

– case T-APP-MONO. We have

$$\Gamma; f \vdash t_1 : T_a \xRightarrow{e'_1} T ! e_1$$

$$\Gamma; f \vdash t_2 : T_b ! e_2 \text{ with } T_b <: T_a$$

Applying the induction hypothesis to  $t_1$  and  $t_2$ :

$$\Gamma; f \vdash (x : T_1) \mapsto t_r : T_c ! e'_1 \text{ with } T_c <: T_a \xRightarrow{e'_1} T$$

$$\Gamma; f \vdash \nu_2 : T'_b ! \perp \text{ with } T'_b <: T_b$$

According to the subtyping rules,  $T_c$  can either be *Nothing* or a monomorphic function type. But since  $T_c$  is the type of a function abstraction, it cannot be *Nothing*, therefore we have  $T_c = T_d \xRightarrow{e'_1} T_e$ . By the canonical forms Lemma A.1, the corresponding function abstraction is a monomorphic one. We obtain:

$$\Gamma; f \vdash (x : T_1) \mapsto t_r : T_1 \xRightarrow{e'_1} T_e ! e'_1, \text{ with } T_d = T_1$$

Therefore:

$$\Gamma, x : T_1; \epsilon \vdash t_r : T_e ! e'_1$$

By transitivity of subtyping, we have  $T'_b <: T_b <: T_a <: T_1$ , and we can apply substitution Lemma 2.3 to obtain:

$$\Gamma; \epsilon \vdash [\nu_2/x]t_r : T'_e ! e''_1 \text{ with } T'_e <: T_e$$

Now we apply the induction hypothesis on  $[\nu_2/x]t_r$  to obtain:

$$\Gamma; \epsilon \vdash r : T_r ! e_r \text{ with } T_r <: T'_e$$

By Lemma A.2 we get  $\Gamma; f \vdash r : T_r ! e_r$ , and by transitivity of subtyping  $T_r <: T'_e <: T_e <: T$ .

– case T-APP-POLY. Similar.

## Appendix A. Soundness Proof for LPE

---

♪ Case E-THROW:  $t = \text{throw}(p)$ . By T-THROW, we have

$$\Gamma; f \vdash \text{throw}(p) : \text{Nothing} ! \text{eff}(\text{THROW}(p))$$

The same typing rule is also applied to the result  $r$ , and we verify  $\text{Nothing} <: \text{Nothing}$  by S-REFL.

♪ Case E-TRY-E:  $t = \text{try } t_1 \text{ catch}(\overline{p}) t_2$ . We have

$$t_1 \downarrow \langle \text{throw}(p), S_1 \rangle$$

$$t_2 \downarrow \langle r_2, S_2 \rangle$$

From the typing rule T-TRY:

$$\Gamma; f \vdash t_1 : T_1 ! e_1 \text{ with } T_1 <: T$$

$$\Gamma; f \vdash t_2 : T_2 ! e_2 \text{ with } T_2 <: T$$

Applying the induction hypothesis to  $t_1$ , we obtain  $\Gamma; f \vdash r_2 : T'_2 ! e'_2$  with  $T'_2 <: T_2$ . Since  $r = r_2$ , it remains to verify using S-TRANS that  $T'_2 <: T_2 <: T$ .

♪ Case E-TRY: similar.

□

### A.2.2 Effect Soundness

*Proof (Theorem 2.2).* The preconditions of the soundness theorem are:

- $\Gamma, f \vdash t : T ! e$
- $t \downarrow \langle r, S \rangle$

Proof of  $S \leq e \sqcup \text{latent}(\Gamma(f))$  by induction on the evaluation rules for term  $t$ .

♪ Case E-APP-E1:  $t = t_1 t_2$ . We have

$$t_1 \downarrow \langle \text{throw}(p), S_1 \rangle$$

$$t \downarrow \langle \text{throw}(p), \text{dynEff}(\text{APP}, S_1, \emptyset, \emptyset) \rangle$$

We look at the sub-cases corresponding to the typing rules for applications:

- case T-APP-E. The typing rule gives  $\Gamma; f \vdash t_1 : \text{Nothing} ! e_1$  and  $\Gamma; f \vdash t_2 : T_2 ! e_2$  and  $e = \text{eff}(\text{APP}, e_1, e_2, \perp)$ .  
By induction hypothesis, we have  $S_1 \leq e_1 \sqcup \text{latent}(\Gamma(f))$ .  
Note that trivially,  $\emptyset \leq e_x$  for any  $e_x$ . Therefore we can apply the consistency Lemma 2.2 to obtain  $\text{dynEff}(\text{APP}, S_1, \emptyset, \emptyset) \leq \text{eff}(\text{APP}, e_1, e_2, \perp) \sqcup \text{latent}(\Gamma(f))$ .
- case T-APP-PARAM:  $t_1 = p$  for some parameter  $p$ . This case is not possible because there is no evaluation rule for parameters.

- case T-APP-MONO:
$$\Gamma; f \vdash t_1 : T_1 \xRightarrow{e_1} T ! e_1$$

$$\Gamma; f \vdash t_2 : T_2 ! e_2 \text{ with } T_2 <: T_1$$

$$e = \text{eff}(\text{APP}, e_1, e_2, e_1)$$
The induction hypothesis on  $t_1$  gives
$$S_1 \leq e_1 \sqcup \text{latent}(\Gamma(f))$$
Since  $\emptyset \leq e_x$  for any  $e_x$ , we can apply Lemma 2.2 to obtain
$$\text{dynEff}(\text{APP}, S_1, \emptyset, \emptyset) \leq \text{eff}(\text{APP}, e_1, e_2, e_1) \sqcup \text{latent}(\Gamma(f))$$
- case T-APP-POLY: similar.

♪ Case E-APP-E2: similar

♪ Case E-APP: Again,  $t = t_1 t_2$ . We have the following preconditions

$$t_1 \downarrow \langle (x : T_a) \mapsto t_r, S_1 \rangle$$

$$t_2 \downarrow \langle v_2, S_2 \rangle n$$

$$[v_2/x]t_r \downarrow \langle r, S_l \rangle$$

$$t \downarrow \langle r, S \rangle \text{ with } S = \text{dynEff}(\text{APP}, S_1, S_2, S_l)$$

There is a sub-case for every possible typing rule for the application expression.

- case T-APP-E: We have  $\Gamma; f \vdash t_1 : \text{Nothing} ! e_1$ .  
By preservation we obtain  $\Gamma; f \vdash (x : T_a) \mapsto t_r : T_1 ! e'_1$  with  $T_1' <: \text{Nothing}$ , which cannot be derived with any typing rule. Therefore, this case is impossible.
- case T-APP-PARAM:  $t_1 = p$  for some parameter  $p$ . This case is not possible because there is no evaluation rule for parameters.
- case T-APP-MONO:
$$\Gamma; f \vdash t_1 : T_1 \xRightarrow{e_1} T ! e_1$$

$$\Gamma; f \vdash t_2 : T_2 ! e_2 \text{ with } T_2 <: T_1$$

$$e = \text{eff}(\text{APP}, e_1, e_2, e_1)$$
By applying the induction hypothesis on  $t_1$  and  $t_2$ , we obtain:
$$S_1 \leq e_1 \sqcup \text{latent}(\Gamma(f))$$

$$S_2 \leq e_2 \sqcup \text{latent}(\Gamma(f))$$

By preservation we have  $\Gamma; f \vdash (x : T_a) \mapsto t_r : T_1' ! \perp$  with  $T_1' <: T_1 \xRightarrow{e_1} T$ . Since the term is a function abstraction,  $T_1' = \text{Nothing}$  is not possible, which implies  $T_1' = T_a \xRightarrow{e'_1} T'$ . By the canonical forms Lemma A.1, the term is a monomorphic function abstraction:

$$\Gamma; f \vdash (x : T_a) \Rightarrow t_r : T_a \xRightarrow{e'_1} T' ! \perp$$

Therefore, we have  $\Gamma, x : T_a; \varepsilon \vdash t_r : T' ! e'_1$ . Applying preservation on  $t_2$  gives us  $\Gamma; f \vdash v_2 : T_2' ! \perp$  with  $T_2' <: T_2 <: T_1 <: T_a$ . We can apply substitution Lemma 2.3 to obtain

$$\Gamma; \varepsilon \vdash [v_2/x]t_r : T'' ! e''_1 \text{ with } T'' <: T' \text{ and } e''_1 \sqsubseteq e'_1$$

The induction hypothesis on  $[v_2/x]t$  gives  $S_l \leq e''_1 \sqcup \text{latent}(\Gamma(\varepsilon))$ .

Since  $\text{latent}(\Gamma(\varepsilon)) = \perp$ , and by  $e''_1 \sqsubseteq e'_1 \sqsubseteq e_1$ , we have  $S_l \leq e_1$ . Together with the

## Appendix A. Soundness Proof for LPE

---

induction hypotheses, we apply the consistency Lemma 2.2 to obtain

$$S \leq \text{eff}(\text{APP}, e_1, e_2, e_i) \sqcup \text{latent}(\Gamma(f))$$

– case T-APP-POLY: similar, shown in Section 2.8.2.

♪ Case E-THROW:  $t = \text{throw}(p)$  and  $t \downarrow \langle \text{throw}(p), \text{dynEff}(\text{THROW}(p)) \rangle$ .

By typing rule T-THROW, we obtain:

$$\Gamma; f \vdash \text{throw}(p) : \text{Nothing} ! \text{eff}(\text{THROW}(p))$$

By Lemma 2.2, we conclude

$$\text{dynEff}(\text{THROW}(p)) \leq \text{eff}(\text{THROW}(p)) \sqcup \text{latent}(\Gamma(f))$$

♪ Case E-TRY-E:  $t = \text{try } t_1 \text{ catch}(\overline{p}) t_2$ . We have

$$t_1 \downarrow \langle \text{throw}(p), S_1 \rangle$$

$$t_2 \downarrow \langle r_2, S_2 \rangle$$

$$S_t = \text{dynEff}(\text{TRY}, S_1)$$

$$S = \text{dynEff}(\text{CATCH}(\overline{p}), S_t, S_2)$$

From the typing rule T-TRY:

$$\Gamma; f \vdash t_1 : T_1 ! e_1 \text{ with } T_1 <: T$$

$$\Gamma; f \vdash t_2 : T_2 ! e_2 \text{ with } T_2 <: T$$

$$e_t = \text{eff}(\text{TRY}, e_1)$$

$$e = \text{eff}(\text{CATCH}(\overline{p}), e_t, e_2)$$

The induction hypotheses are

$$S_1 \leq e_1 \sqcup \text{latent}(\Gamma(f))$$

$$S_2 \leq e_2 \sqcup \text{latent}(\Gamma(f))$$

Applying Lemma 2.2 to  $S_t$  and  $e_t$  yields  $S_t \leq e_t \sqcup \text{latent}(\Gamma(f))$ . Now we can apply the same lemma to  $S$  and  $e$  and conclude  $S \leq e \sqcup \text{latent}(\Gamma(f))$ .

♪ Case E-TRY: similar.

□

# Bibliography

- Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 63–74, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328449. URL <http://doi.acm.org/10.1145/1328438.1328449>.
- Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 378–400. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39037-1. doi: 10.1007/978-3-642-39038-8\_16. URL [http://dx.doi.org/10.1007/978-3-642-39038-8\\_16](http://dx.doi.org/10.1007/978-3-642-39038-8_16).
- D. Aspinall and A. Compagnoni. Subtyping dependent types. In *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 86–97, 1996. doi: 10.1109/LICS.1996.561307.
- Philip Bagwell and Tiark Rompf. RRB-Trees: Efficient Immutable Vectors. Technical report, 2011.
- Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-70591-8. doi: 10.1007/978-3-540-70592-5\_17. URL [http://dx.doi.org/10.1007/978-3-540-70592-5\\_17](http://dx.doi.org/10.1007/978-3-540-70592-5_17).
- Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In Rudrapatna K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51. Springer Berlin Heidelberg, 1993. ISBN 978-3-540-57529-0. doi: 10.1007/3-540-57529-4\_42. URL [http://dx.doi.org/10.1007/3-540-57529-4\\_42](http://dx.doi.org/10.1007/3-540-57529-4_42).
- Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.
- Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A

## Bibliography

---

- type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, OOPSLA '09*, pages 97–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640097. URL <http://doi.acm.org/10.1145/1640089.1640097>.
- Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '03*, pages 213–223, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5. doi: 10.1145/604131.604156. URL <http://doi.acm.org/10.1145/604131.604156>.
- Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.
- Edwin Brady. Programming and reasoning with algebraic effects and dependent types. To Appear in *Proceedings of the ACM SIGPLAN international conference on Functional programming*, 2013. URL <http://edwinb.wordpress.com/2013/03/28/programming-and-reasoning-with-algebraic-effects-and-dependent-types/>.
- Sigmund Cherm and Radu Rugina. A practical escape and effect analysis for building lightweight method summaries. In *Proceedings of the 16th international conference on Compiler construction, CC'07*, pages 172–186, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71228-2. URL <http://dl.acm.org/citation.cfm?id=1759937.1759953>.
- Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '02*, pages 292–310, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1. doi: 10.1145/582419.582447. URL <http://doi.acm.org/10.1145/582419.582447>.
- David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '98*, pages 48–64, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8. doi: 10.1145/286936.286947. URL <http://doi.acm.org/10.1145/286936.286947>.
- William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, OOPSLA '09*, pages 557–572, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640133. URL <http://doi.acm.org/10.1145/1640089.1640133>.
- Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, ICFP '07*, pages 315–326, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291199. URL <http://doi.acm.org/10.1145/1291151.1291199>.



- Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73588-5. doi: 10.1007/978-3-540-73589-2\_3. URL [http://dx.doi.org/10.1007/978-3-540-73589-2\\_3](http://dx.doi.org/10.1007/978-3-540-73589-2_3).
- Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 681–690, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985889. URL <http://doi.acm.org/10.1145/1985793.1985889>.
- Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '13*, pages 287–300, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429104. URL <http://doi.acm.org/10.1145/2429069.2429104>.
- Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in Java. In *Proceedings of the 15th ACM conference on Computer and communications security, CCS '08*, pages 161–174, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-810-7. doi: 10.1145/1455770.1455793. URL <http://doi.acm.org/10.1145/1455770.1455793>.
- Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, PLDI '93*, pages 237–247, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: 10.1145/155090.155113. URL <http://doi.acm.org/10.1145/155090.155113>.
- Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, PLDI '02*, pages 234–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512558. URL <http://doi.acm.org/10.1145/512529.512558>.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.
- David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming, LFP '86*, pages 28–38, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319848. URL <http://doi.acm.org/10.1145/319838.319848>.
- David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Report on the FX programming language. Technical report, MIT/LCS/TR-531, 1992.

## Bibliography

---

- Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. Java ui : Effects for controlling ui object access. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 179–204. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39037-1. doi: 10.1007/978-3-642-39038-8\_8. URL [http://dx.doi.org/10.1007/978-3-642-39038-8\\_8](http://dx.doi.org/10.1007/978-3-642-39038-8_8).
- James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013. ISBN 0133260224, 9780133260229.
- Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February 2009. ISSN 0304-3975. doi: 10.1016/j.tcs.2008.09.019. URL <http://dx.doi.org/10.1016/j.tcs.2008.09.019>.
- Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and promises (Scala documentation), 2012. URL <http://docs.scala-lang.org/overviews/core/futures.html>.
- Anders Hejlsberg. The trouble with checked exceptions, 2003. URL <http://www.artima.com/intv/handcuffs.html>.
- Anders Hejlsberg. Introducing async – simplifying asynchronous programming, 2010. URL <http://channel9.msdn.com/Blogs/Charles/Anders-Hejlsberg-Introducing-Async>.
- My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 176–185, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199481. URL <http://doi.acm.org/10.1145/199448.199481>.
- Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: checking and inference of reference immutability and method purity. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 879–896, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384680. URL <http://doi.acm.org/10.1145/2384616.2384680>.
- Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-37215-8. doi: 10.1007/11813040\_19. URL [http://dx.doi.org/10.1007/11813040\\_19](http://dx.doi.org/10.1007/11813040_19).
- Etienne Kneuss, Viktor Kunčak, and Philippe Suter. Effect analysis for programs with callbacks. In *Verified Software: Theories, Tools, Experiments*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.
- John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*,

- PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: 10.1145/178243.178246. URL <http://doi.acm.org/10.1145/178243.178246>.
- Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3. doi: 10.1145/337449.337465. URL <http://doi.acm.org/10.1145/337449.337465>.
- Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006. ISSN 0163-5948. doi: 10.1145/1127878.1127884. URL <http://doi.acm.org/10.1145/1127878.1127884>.
- Daan Leijen. Koka: A language with effect inference. <http://research.microsoft.com/en-us/projects/koka/2012-overviewkoka.pdf>, April 2012.
- K. Rustan M. Leino and Peter Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In Peter Müller, editor, *Advanced Lectures on Software Engineering*, pages 91–139. Springer-Verlag, Berlin, Heidelberg, 2010. ISBN 3-642-13009-7, 978-3-642-13009-0. URL <http://dl.acm.org/citation.cfm?id=2167938.2167942>.
- K.RustanM. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-17510-7. doi: 10.1007/978-3-642-17511-4\_20. URL [http://dx.doi.org/10.1007/978-3-642-17511-4\\_20](http://dx.doi.org/10.1007/978-3-642-17511-4_20).
- K.RustanM. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–515. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22159-3. doi: 10.1007/978-3-540-24851-4\_22. URL [http://dx.doi.org/10.1007/978-3-540-24851-4\\_22](http://dx.doi.org/10.1007/978-3-540-24851-4_22).
- Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199528. URL <http://doi.acm.org/10.1145/199448.199528>.
- Ben Lippmeier. *Type Inference and Optimisation for an Impure World*. PhD thesis, Australian National University, 2010.
- J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73564. URL <http://doi.acm.org/10.1145/73560.73564>.

## Bibliography

---

- Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-420-1. doi: 10.1145/1481861.1481868. URL <http://doi.acm.org/10.1145/1481861.1481868>.
- Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4. URL [http://dx.doi.org/10.1016/0890-5401\(91\)90052-4](http://dx.doi.org/10.1016/0890-5401(91)90052-4).
- Flemming Nielson and HanneRiis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66624-0. doi: 10.1007/3-540-48092-7\_6. URL [http://dx.doi.org/10.1007/3-540-48092-7\\_6](http://dx.doi.org/10.1007/3-540-48092-7_6).
- Martin Odersky. The Scala language specification, 2013. URL <http://www.scala-lang.org/docu/files/ScalaReference.pdf>.
- Martin Odersky and Adriaan Moors. Fighting bit rot with types (experience report: Scala collections). In Ravi Kannan and K Narayan Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 427–451, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-13-2. doi: <http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2009.2338>. URL <http://drops.dagstuhl.de/opus/volltexte/2009/2338>.
- Oracle. Java Platform, Standard Edition 7, API Specification, 2013a. URL <http://docs.oracle.com/javase/7/docs/api/index.html>.
- Oracle. Java Platform, Standard Edition 8, API Specification, 2013b. URL <http://download.java.net/jdk8/docs/api/>.
- Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 75–86, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328451. URL <http://doi.acm.org/10.1145/1328438.1328451>.
- David J. Pearce. JPure: A modular purity system for Java. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 104–123. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19860-1. doi: 10.1007/978-3-642-19861-8\_7. URL [http://dx.doi.org/10.1007/978-3-642-19861-8\\_7](http://dx.doi.org/10.1007/978-3-642-19861-8_7).
- Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158524. URL <http://doi.acm.org/10.1145/158511.158524>.

- Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.
- Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 773–789. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8\_54. URL [http://dx.doi.org/10.1007/978-3-642-39799-8\\_54](http://dx.doi.org/10.1007/978-3-642-39799-8_54).
- John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS '02*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL <http://dl.acm.org/citation.cfm?id=645683.664578>.
- Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTJP '13*, pages 4:1–4:7, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2042-9. doi: 10.1145/2489804.2489808. URL <http://doi.acm.org/10.1145/2489804.2489808>.
- Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8:1–22, 0 1998. ISSN 1469-7653. doi: 10.1017/S0956796897002943. URL [http://journals.cambridge.org/article\\_S0956796897002943](http://journals.cambridge.org/article_S0956796897002943).
- Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 148–172. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0\_8. URL [http://dx.doi.org/10.1007/978-3-642-03013-0\\_8](http://dx.doi.org/10.1007/978-3-642-03013-0_8).
- Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. In *Selected papers of the colloquium on Formal approaches of software engineering, TAPSOFT '93*, pages 197–226, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V. URL <http://dl.acm.org/citation.cfm?id=202776.202784>.
- Manu Sridharan, Satish Chandra, Julian Dolby, StephenJ. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 196–232. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-36945-2. doi: 10.1007/978-3-642-36946-9\_8. URL [http://dx.doi.org/10.1007/978-3-642-36946-9\\_8](http://dx.doi.org/10.1007/978-3-642-36946-9_8).
- Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24297-0. doi: 10.1007/978-3-540-30579-8\_14. URL [http://dx.doi.org/10.1007/978-3-540-30579-8\\_14](http://dx.doi.org/10.1007/978-3-540-30579-8_14).

## Bibliography

---

- Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In *Proceedings of the 13th international conference on Practical aspects of declarative languages, PADL'11*, pages 175–189, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-18377-5. URL <http://dl.acm.org/citation.cfm?id=1946313.1946334>.
- J. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 162–173, 1992a. doi: 10.1109/LICS.1992.185530.
- Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 6 1992b. ISSN 1469-7653. doi: 10.1017/S0956796800000393. URL [http://journals.cambridge.org/article\\_S0956796800000393](http://journals.cambridge.org/article_S0956796800000393).
- Ross Tate. The sequential semantics of producer effect systems. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '13*, pages 15–26, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429074. URL <http://doi.acm.org/10.1145/2429069.2429074>.
- Tachio Terauchi and Alex Aiken. Witnessing side-effects. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, ICFP '05*, pages 105–115, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7. doi: 10.1145/1086365.1086379. URL <http://doi.acm.org/10.1145/1086365.1086379>.
- Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value  $\lambda$ -calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '94*, pages 188–201, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0. doi: 10.1145/174675.177855. URL <http://doi.acm.org/10.1145/174675.177855>.
- Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld, and Olesen Peter Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, January 2006.
- Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05*, pages 211–230, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094828. URL <http://doi.acm.org/10.1145/1094811.1094828>.
- Marko van Dooren and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05*, pages 455–471, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094847. URL <http://doi.acm.org/10.1145/1094811.1094847>.

Philip Wadler. Linear types can change the world! In C.B. Jones, editor, *Proceedings of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods*. North-Holland, 1990. ISBN 9780444885456.

Philip Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, September 1997. ISSN 0360-0300. doi: 10.1145/262009.262011. URL <http://doi.acm.org/10.1145/262009.262011>.

Philip Wadler. The marriage of effects and monads. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 63–74, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi: 10.1145/289423.289429. URL <http://doi.acm.org/10.1145/289423.289429>.





